Contents

Intel® Advisor User Guide ................................................................. 7
  What's New ...................................................................................... 7
  Before You Begin ........................................................................... 9
  Launch the Intel® Advisor ............................................................. 12
    Intel® Advisor GUI and Microsoft Visual Studio® Integration .... 13
    Create a Project Dialog Box .......................................................... 14
    Project Navigator Pane ................................................................. 15
    Intel Advisor Menus and Toolbars ............................................... 16
    Corresponding advixe-cl Command Options Dialog Box ........ 20
    Intel Advisor CLI .......................................................................... 21
    Setting and Using Intel® Advisor Environment Variables .......... 21
    Launch Intel® Advisor Beta from a Docker® Container for Linux® OS ... 23
  Configuration .................................................................................. 24
    Project Properties Dialog Box ....................................................... 25
      Analysis Target Tab .................................................................. 27
      Binary/Symbol Search Tab ......................................................... 34
      Source Search Tab .................................................................. 35
      Binary/Symbol Search and Source Search Locations .............. 36
    Options Dialog Box ..................................................................... 38
      Editor Tab ................................................................................ 38
      General Tab ........................................................................... 40
      Result Location Tab ................................................................. 42
      Assembly Tab ........................................................................ 42
      Choose External Editor Dialog Box ........................................... 43
    Target Application Configuration .................................................. 43
      Build Settings for Supported Languages ............................... 43
      Limiting the Number of Threads Used by Parallel Frameworks .... 44
      Choosing a Small, Representative Data Set ............................ 44
  Workflows ....................................................................................... 45
    Vectorization Workflow Diagram .................................................. 47
    Vectorization Workflow Tab .......................................................... 48
    Threading Workflow Diagram ......................................................... 49
    Threading Workflow Tab ................................................................ 50
  Survey, Trip Counts, FLOPS, and Roofline Analyses ................... 52
    Survey Analysis ........................................................................ 53
      Next Steps: After Running Survey Analysis ........................ 57
      Advanced View Pane (Survey Report) ..................................... 59
      Filters Pane ............................................................................. 65
      Loop Information Pane (Survey Report) ................................ 67
      Survey Source Window ............................................................. 69
      Configure Columns Dialog Box .............................................. 72
    Trip Counts Analysis .................................................................. 72
    FLOP Analysis .......................................................................... 73
    Roofline Analysis ...................................................................... 75
  Dependencies and Memory Access Patterns Analyses .................. 85
    Loop Information Pane (Refinement Reports) ........................... 85
    Running Multiple Analyses Automatically ............................... 86
    Dependencies Analysis ............................................................... 87
      Code Locations Pane ............................................................... 88
<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data Communication, Child Task</td>
<td>444</td>
</tr>
<tr>
<td>Inconsistent Lock Use</td>
<td>445</td>
</tr>
<tr>
<td>Lock Hierarchy Violation</td>
<td>446</td>
</tr>
<tr>
<td>Memory Reuse</td>
<td>448</td>
</tr>
<tr>
<td>Memory Reuse, Child Task</td>
<td>449</td>
</tr>
<tr>
<td>Memory Watch</td>
<td>449</td>
</tr>
<tr>
<td>Missing End Site</td>
<td>451</td>
</tr>
<tr>
<td>Missing End Task</td>
<td>451</td>
</tr>
<tr>
<td>Missing Start Site</td>
<td>452</td>
</tr>
<tr>
<td>Missing Start Task</td>
<td>453</td>
</tr>
<tr>
<td>No Tasks in Parallel Site</td>
<td>454</td>
</tr>
<tr>
<td>One Task Instance in Parallel Site</td>
<td>454</td>
</tr>
<tr>
<td>Orphaned Task</td>
<td>455</td>
</tr>
<tr>
<td>Parallel Site Information</td>
<td>456</td>
</tr>
<tr>
<td>Thread Information</td>
<td>456</td>
</tr>
<tr>
<td>Unhandled Application Exception</td>
<td>457</td>
</tr>
<tr>
<td>Troubleshooting Reference</td>
<td>458</td>
</tr>
<tr>
<td>Analysis of Debug Build</td>
<td>458</td>
</tr>
<tr>
<td>Analysis of Release Build</td>
<td>459</td>
</tr>
<tr>
<td>Debug Information Not Available</td>
<td>459</td>
</tr>
<tr>
<td>No Annotations Found</td>
<td>461</td>
</tr>
<tr>
<td>No Data</td>
<td>462</td>
</tr>
<tr>
<td>Source Not Available</td>
<td>463</td>
</tr>
<tr>
<td>Survey Tool does not Display Survey Report</td>
<td>464</td>
</tr>
<tr>
<td>Undefined Linker References to dlopen or dlsym</td>
<td>465</td>
</tr>
<tr>
<td>Unexpected C/C++ Compilation Errors After Adding Annotations</td>
<td>465</td>
</tr>
<tr>
<td>Unexpected Unmatched Annotations in the Dependencies Report</td>
<td>466</td>
</tr>
<tr>
<td>Appendix</td>
<td>466</td>
</tr>
<tr>
<td>Getting Help and Support</td>
<td>467</td>
</tr>
<tr>
<td>Glossary</td>
<td>467</td>
</tr>
<tr>
<td>Key Concepts</td>
<td>470</td>
</tr>
<tr>
<td>Notational Conventions</td>
<td>470</td>
</tr>
<tr>
<td>Related Information</td>
<td>471</td>
</tr>
<tr>
<td>Parallelism</td>
<td>472</td>
</tr>
<tr>
<td>Parallel Processing Terminology</td>
<td>472</td>
</tr>
<tr>
<td>Adding Parallelism</td>
<td>473</td>
</tr>
<tr>
<td>Common Issues When Adding Parallelism</td>
<td>473</td>
</tr>
<tr>
<td>Parallel Programming Implementations</td>
<td>475</td>
</tr>
<tr>
<td>Data Sharing Problems</td>
<td>476</td>
</tr>
<tr>
<td>Data Sharing Problem Types</td>
<td>476</td>
</tr>
<tr>
<td>Problem Solving Strategies</td>
<td>479</td>
</tr>
<tr>
<td>Adding Parallelism to Your Program</td>
<td>489</td>
</tr>
<tr>
<td>Before You Add Parallelism: Choose a Parallel Framework</td>
<td>489</td>
</tr>
<tr>
<td>Adding the Parallel Framework to Your Build Environment</td>
<td>492</td>
</tr>
<tr>
<td>Annotation Report</td>
<td>496</td>
</tr>
<tr>
<td>Replacing Annotations with Intel® Threading Building Blocks</td>
<td>497</td>
</tr>
<tr>
<td>(Intel® TBB) Code</td>
<td>497</td>
</tr>
<tr>
<td>Replacing Annotations with OpenMP* Code</td>
<td>502</td>
</tr>
<tr>
<td>Next Steps for the Parallel Program</td>
<td>514</td>
</tr>
<tr>
<td>Notices and Disclaimers</td>
<td>516</td>
</tr>
</tbody>
</table>
Intel® Advisor User Guide

Intel® Advisor is composed of a set of tools to help ensure your Fortran, C and C++ (as well as .NET on Windows®) applications realize full performance potential on modern processors:

- **Vectorization Advisor** is a vectorization optimization tool that lets you identify high-impact, under-optimized loops, what is blocking vectorization, and where it is safe to force vectorization. It also provides code-specific how-can-I-fix-this-issue recommendations. For details, see Vectorization Workflow Diagram.
- **Roofline Analysis** visualizes actual performance against hardware-imposed performance ceilings (rooflines). It provides insights into where the bottlenecks are, which loops are worth optimizing for performance, what are the likely causes of bottlenecks and what should be the next optimization steps. For details, see CPU Roofline Analysis or GPU Roofline Analysis.
- **Threading Advisor** is a fast-track threading design and prototyping tool that lets you analyze, design, tune, and check threading design options without disrupting your normal development. For details, see Threading Workflow Diagram.
- **Offload Advisor** (Intel® Advisor Beta only) allows you to identify high-impact opportunities to offload to GPU as well as the areas that are not advantageous to offload. It provides performance speedup projection on accelerators along with offload overhead estimation and pinpoints accelerator performance bottlenecks. For details, see the Offload Advisor.
- **Flow Graph Analyzer** is a visual prototyping tool that lets you represent and analyze performance for applications that use the Intel® Threading Building Blocks (Intel® TBB) flow graph interfaces. For details, see the Flow Graph Analyzer section of this guide.

Intel® Advisor is available as a standalone product and as part of the following products:

- Intel® System Studio Professional Edition
- Intel® System Studio Ultimate Edition
- Intel® Parallel Studio XE Professional Edition
- Intel® Parallel Studio XE Cluster Edition
- Intel® oneAPI Base Toolkit (Beta)

**What's New**

This topic lists at a high level new features and improvements in Intel® Advisor.

**Intel® Advisor 2021 Beta**

The following features and improvements are applicable to Intel® Advisor Beta, available as part of the Intel® oneAPI Base Toolkit (Beta).

For full list of new Intel® Advisor Beta features and known issues, refer to the Release Notes for Intel® Advisor Beta.

**Intel® Advisor 2021 Beta 09**

- Technical Preview: Added hints for data transfer optimizations to Offload Modeling report. This is implemented in the new UI.

  For details, see the Examine Data Transfers for Modeled Regions section in the Intel® Advisor Beta User Guide (PDF).

**NOTE** To enable the new preview UI, set the ADVIXE_EXPERIMENTAL=beta_gui.

**Intel® Advisor 2021 Beta 08**

- Introduced Intel® Advisor Beta UI redesign (preview feature), which includes:
New look-and-feel for **Workflow pane and Toolbars**
**Offload Modeling** and **GPU Roofline** workflows integrated in GUI

For details about the new Intel® Advisor Beta UI, see the Intel® Advisor Beta User Guide (PDF).

**NOTE** To enable the new UI, set the `ADVIXE_EXPERIMENTAL=beta_gui`.

**Intel® Advisor 2021 Beta 07**

Intel Advisor Beta:

- **GPU Roofline** is now publicly available without the `gpu-roofline` environment variable.
- Introduced new command-line options for GPU Roofline: `profile-gpu`, `gpu-carm`, `gpu-sampling-interval`.
- Introduced three strategies to manage kernel invocation taxes when modeling performance: do not hide invocation taxes, hide all invocation taxes except the first one, hide a part of invocation taxes. For more information, see Manage Invocation Taxes.

Flow Graph Analyzer:

- Implemented DPC++ support: You can profile Data Parallel C++ (DPC++) code on CPUs on Linux* OS. The collector is only available on Linux OS, but you can view the data on all platforms.
- Added support for visualizing DPC++/SYCL asynchronous task-graphs and connecting the executions traces with graph nodes.
- Added analytics for determining inefficiencies in thread start-up and join for DPC++ data parallel algorithms running on the CPU using `cpu_selector`.
- Added rules to the Static Rule-check engine to determine issues with unnecessary copies during the creation of buffers, host pointer accessor usage in a loop, multiple build/compilations for the same kernel when invoked multiple times.

**Intel® Advisor 2021 Beta 05**

- Offload Advisor scripts support Python* 3.6 and 3.7. If you do not have Python 3.6 or 3.7 installed, you can use internal Python using the `advixe-python` command-line tool. See the Collect Performance Metrics and Run Performance Modeling for sample command lines.

**Intel® Advisor 2021 Beta 04**

- **GPU Roofline** analysis (tech preview) is now available on Windows* OS.
- GPU Roofline report supports integer data types with the `--data-type=int` option.
- Offload Advisor can be run with the internal Python* using the `advixe-python` command-line tool. See the Collect Performance Metrics and Run Performance Modeling for sample command lines.
- A new `--no-stacks` options is introduced for `collect.py` and `analyze.py` to enforce flat data usage and minimize overhead.
- Messages in the **Why Not Offloaded** column of the Offload Advisor report are extended with reasons why a region is not profitable to offload.

**Initial Release**

- This release provides initial support for oneAPI Data Parallel C++ (DPC++) performance profiling on CPU and GPU targets.
- **Offload Advisor** tool allows you to collect performance predictor data and determine what parts of code can be offloaded to a target device (for example, GPU), accelerating the performance of your CPU-based application.
- **GPU Roofline analysis** (technical preview feature) allows you to estimate and visualize how loops perform on a GPU and identify the main limiting factor.
**Intel® Advisor 2020**

The following features and improvements are applicable to Intel® Advisor installed as part of Intel® Parallel Studio XE or Intel® System Studio, or standalone Intel® Advisor.

For a full list of present and past release notes, see Intel® Advisor Release Notes and New Features.

**Intel® Advisor 2020 Update 2**

Intel® Advisor 2020 Update 2 introduces Memory-Level Roofline feature (previously known as Integrated Roofline, tech preview feature). Memory-Level Roofline collects metrics for all memory levels and allows you to observe each loop/function at different cache levels.

**Intel® Advisor 2020 Update 1**

- Integrated Roofline chart (technical preview feature) supports highlighting the limiting memory level roof for a loop/function. Limiting roof is defined as the roof at a minimum distance from the point with given arithmetic intensities for the selected loop/function and hardware peaks for multiple levels of memory subsystem.

  Roofline guidance and recommendations are clarified with a limiting memory level roof, which improves recommendation accuracy.

- **Code Analytics pane** shows a single-kernel Roofline guidance for all memory levels with dots for multiple levels of a memory subsystem and limiting roof highlighting.

**Intel® Advisor 2020**

- Intel Advisor has been updated to include more recent versions of 3rd party components, which include functional and security updates. Users should update to the latest version.
- Intel Advisor viewer for macOS* is now notarized to run on Mac OS 10.15.

**Before You Begin**

*This section contains steps you should perform before you begin running analyses on your application with Intel® Advisor or Intel® Advisor Beta.*

- Build an optimized binary of your application in **release** mode using settings designed to produce the most accurate and complete analysis results.
- Verify the resulting executable runs before trying to analyze it with the Intel Advisor.

**Optimal C/C++ Settings**

<table>
<thead>
<tr>
<th>To Do This</th>
<th>For This Tool</th>
<th>Optimal C/C++ Settings</th>
</tr>
</thead>
<tbody>
<tr>
<td>Request full debug information (compiler and linker).</td>
<td>Vectorization Advisor</td>
<td>Linux* OS command line: -g</td>
</tr>
<tr>
<td>Threading Advisor</td>
<td>Windows* OS command line:</td>
<td></td>
</tr>
<tr>
<td>Offload Advisor</td>
<td>• /ZI</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• /DEBUG</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Microsoft Visual Studio* IDE:</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• C/C++ &gt; General &gt; Debug Information Format &gt; Program Database (/Zi)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Linker &gt; Debugging &gt; Generate Debug Info &gt; Yes (/DEBUG)</td>
<td></td>
</tr>
<tr>
<td>Request moderate optimization.</td>
<td>Vectorization Advisor</td>
<td>Linux* OS command line: -O2 or higher</td>
</tr>
<tr>
<td></td>
<td>Windows* OS command line:</td>
<td></td>
</tr>
<tr>
<td>To Do This</td>
<td>For This Tool</td>
<td>Optimal C/C++ Settings</td>
</tr>
<tr>
<td>------------------------------------------</td>
<td>---------------------</td>
<td>------------------------------------------------------------</td>
</tr>
<tr>
<td>Threading Advisor</td>
<td></td>
<td>• /O2 or higher</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• /Ob1 (Threading Advisor only)</td>
</tr>
<tr>
<td>Offload Advisor</td>
<td></td>
<td>Visual Studio* IDE:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• C/C++ &gt; Optimization &gt; Optimization &gt; Maximum Optimization (Favor Speed) (/O2) or higher</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• C/C++ &gt; Optimization &gt; Inline Function Expansion &gt; Only_inline (/Ob1) (Threading Advisor only)</td>
</tr>
<tr>
<td>Disable interprocedural optimizations</td>
<td>Offload Advisor only</td>
<td>Linux* OS command line: -no-ipo</td>
</tr>
<tr>
<td>that may inhibit the ability of Intel®</td>
<td></td>
<td>Windows* OS command line: /Qipo-</td>
</tr>
<tr>
<td>Advisor Beta to collect performance data.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>For Intel® C++ Compiler Classic only.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Produce compiler diagnostics (necessary</td>
<td>Vectorization</td>
<td>Linux* OS command line: -qopt-report=5</td>
</tr>
<tr>
<td>for version 15.0 of the Intel® C++</td>
<td>Advisor only</td>
<td>Windows* OS command line: /Qopt-report:5</td>
</tr>
<tr>
<td>Compiler Classic; unnecessary for</td>
<td></td>
<td>Visual Studio* IDE: C/C++ &gt; Diagnostics [Intel C++] &gt;</td>
</tr>
<tr>
<td>version 16.0 and higher)</td>
<td></td>
<td>Optimization Diagnostic Level &gt; Level 5 (/Qopt-report: 5)</td>
</tr>
<tr>
<td>Enable vectorization.</td>
<td>Vectorization</td>
<td>Linux* OS command line: -vec</td>
</tr>
<tr>
<td>Advisor only</td>
<td></td>
<td>Windows* OS command line: /Qvec</td>
</tr>
<tr>
<td>Enable SIMD directives.</td>
<td>Vectorization</td>
<td>Linux command line: -simd</td>
</tr>
<tr>
<td>Advisor only</td>
<td></td>
<td>Windows* OS command line: /Qsimd</td>
</tr>
<tr>
<td>Enable generation of multi-threaded code</td>
<td>Vectorization</td>
<td>Linux* OS command line: -qopenmp</td>
</tr>
<tr>
<td>based on OpenMP* directives.</td>
<td>Advisor only</td>
<td>Windows* OS command line: /Qopenmp</td>
</tr>
<tr>
<td>Search additional directory related to</td>
<td>Primarily</td>
<td>Visual Studio* IDE: C/C++ &gt; General &gt; Additional Include</td>
</tr>
<tr>
<td>Intel Advisor annotation definitions.</td>
<td>Threading Advisor,</td>
<td>Directories &gt; $(ADVISOR_[product_year]_DIR) \include;%(AdditionalIncludeDirectories)</td>
</tr>
<tr>
<td></td>
<td>but could also be</td>
<td></td>
</tr>
<tr>
<td></td>
<td>useful for Vectorization Advisor refinement analyses</td>
<td></td>
</tr>
</tbody>
</table>
## Optimal C/C++ Settings

<table>
<thead>
<tr>
<th>To Do This</th>
<th>For This Tool</th>
<th>Optimal C/C++ Settings</th>
</tr>
</thead>
<tbody>
<tr>
<td>Search for unresolved references in multithreaded,</td>
<td>Threading Advisor only</td>
<td>Linux* OS command line: -Bdynamic</td>
</tr>
<tr>
<td>dynamically linked libraries.</td>
<td></td>
<td>Windows* OS command line: /MD or /MDd</td>
</tr>
<tr>
<td>Enable dynamic loading.</td>
<td>Threading Advisor only</td>
<td>Visual Studio* IDE: C/C++ &gt; Code Generation &gt; Runtime Library &gt; Mutithread</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Linux* OS command line: -ldl</td>
</tr>
</tbody>
</table>

## Optimal Fortran Settings

<table>
<thead>
<tr>
<th>To Do This</th>
<th>For This Tool</th>
<th>Optimal Fortran Settings</th>
</tr>
</thead>
<tbody>
<tr>
<td>Request full debug information (compiler and linker).</td>
<td>Vectorization Advisor</td>
<td>Linux* OS command line: -g</td>
</tr>
<tr>
<td></td>
<td>Threading Advisor</td>
<td>Windows* OS command line:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• /debug=full</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• /DEBUG</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Visual Studio* IDE:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Fortran &gt; General &gt; Debug Information Format &gt; Full (/debug=full)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Linker &gt; Debugging &gt; Generate Debug Info &gt; Yes (/DEBUG)</td>
</tr>
<tr>
<td>Request moderate optimization.</td>
<td>Vectorization Advisor</td>
<td>Linux* OS command line: -o2 or higher</td>
</tr>
<tr>
<td></td>
<td>Threading Advisor</td>
<td>Windows* OS command line:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• /o2 or higher</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• /Ob1 (Threading Advisor only)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Visual Studio* IDE:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Fortran &gt; Optimization &gt; Optimization &gt; Maximize Speed or higher</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Fortran &gt; Optimization &gt; Inline Function Expansion &gt; Only INLINE directive (/Ob1)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(Threading Advisor only)</td>
</tr>
<tr>
<td>Produce compiler diagnostics (necessary for version</td>
<td>Vectorization Advisor</td>
<td>Linux* OS command line: -qopt-report=5</td>
</tr>
<tr>
<td>15.0 of the Intel® Fortran Compiler Classic;</td>
<td>Threading Advisor only</td>
<td>Windows* OS command line:</td>
</tr>
<tr>
<td>unnecessary for version 16.0 and higher).</td>
<td></td>
<td>• Qopt-report:5</td>
</tr>
<tr>
<td>Enable vectorization.</td>
<td>Vectorization Advisor</td>
<td>Visual Studio* IDE:</td>
</tr>
<tr>
<td></td>
<td>Threading Advisor only</td>
<td>Fortran &gt; Diagnostics &gt; Optimization Diagnostic Level &gt; Level 5 (/Qopt-report:5)</td>
</tr>
<tr>
<td>Enable SIMD directives.</td>
<td>Vectorization Advisor</td>
<td>Linux* OS command line: -vec</td>
</tr>
<tr>
<td></td>
<td>Threading Advisor only</td>
<td>Windows* OS command line:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Qvec</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Linux* OS command line: -simd</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Windows* OS command line:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Qsimd</td>
</tr>
</tbody>
</table>
### Launch the Intel® Advisor

The following sections provide simple steps to quickly ramp up with the Intel® Advisor.

- **Intel Advisor GUI and Microsoft Visual Studio® Integration** - simple steps to launch Intel® Advisor standalone GUI or from Microsoft Visual Studio
- Intel Advisor CLI - simple steps to launch with Intel® Advisor CLI.
- Launch Intel® Advisor Beta from a Docker® container (Linux® OS) - simple steps to launch the Intel Advisor Beta from a Docker container.

<table>
<thead>
<tr>
<th>To Do This</th>
<th>For This Tool</th>
<th>Optimal Fortran Settings</th>
</tr>
</thead>
<tbody>
<tr>
<td>Enable generation of multi-threaded code based on OpenMP* directives.</td>
<td>Vectorization Advisor only</td>
<td>Linux* OS command line: <code>-qopenmp</code></td>
</tr>
<tr>
<td>Search additional directory related to Intel Advisor annotation definitions.</td>
<td>Primarily Threading Advisor, but could also be useful for Vectorization Advisor refinement analyses.</td>
<td>Visual Studio* IDE: <strong>Fortran &gt; Language &gt; Process OpenMP Directives &gt; Generate Parallel Code (</strong>/Qopenmp**)</td>
</tr>
<tr>
<td>Search for unresolved references in multithreaded, dynamically linked libraries.</td>
<td>Threading Advisor only</td>
<td>Linux* OS command line:</td>
</tr>
<tr>
<td>Enable dynamic loading.</td>
<td>Threading Advisor only</td>
<td>Windows* OS command line:</td>
</tr>
</tbody>
</table>

**Launch the Intel® Advisor**

- **Intel Advisor GUI and Microsoft Visual Studio® Integration** - simple steps to launch Intel® Advisor standalone GUI or from Microsoft Visual Studio
- Intel Advisor CLI - simple steps to launch with Intel® Advisor CLI.
- **Launch Intel® Advisor Beta from a Docker® container (Linux® OS)** - simple steps to launch the Intel Advisor Beta from a Docker container.
Quick steps to ramp up with Intel® Advisor are included in Getting Started with Intel® Advisor. For more detailed steps to set up the Intel® Advisor Beta environment and run performance profiling and performance modeling, see the Offload Advisor section.

**Intel® Advisor GUI and Microsoft Visual Studio* Integration**

Steps to ramp up with Intel® Advisor GUI and get quick results.

This section provides simple steps to get quick results and grasp the overall idea of how you can use Intel® Advisor. For detailed guidelines refer to the ultimate Get Started with Intel® Advisor.

**Run the Intel Advisor GUI**

The Intel® Advisor GUI is available:

- On the Microsoft Windows* OS:
  - Intel® Advisor: From the Start menu, choose All Programs > Intel Parallel Studio XE > Intel Advisor [version].
  - Intel® Advisor Beta: From the Start menu, choose All Programs > Intel oneAPI [version] > Intel Advisor [version].
- In the Microsoft Visual Studio*: From the Tools menu, choose Intel Advisor [version] > Vectorization and Threading Advisor Analysis.
- From a command line on Windows* or Linux* OS systems. Type the advixe-gui command and optionally specify a path to one of following:
  - Full (absolute) path to a result file (*.advixe)
  - Full path to a project file (config.advixeproj)
  - Full path to a project directory. If there is no project file in the directory, the Create a Project dialog box opens and prompts you to create a new project in the given directory.

On Windows systems, if the path contains embedded spaces, enclose it in quotation marks.

**Create New Project**

NOTE In case of Microsoft Visual Studio* integration, you can skip this step, as Intel Advisor creates new project automatically when opened. Intel Advisor uses Visual Studio Project Properties to configure the Intel Advisor project.

If it is not displayed, to view the Welcome Page, click View > Welcome.

Use the Welcome Page to create a new project, view the Current project (already open), open a Recent project (or any existing project), and view the Intel Advisor Getting Started page.

**Run an Analysis**

In the Workflow tab, click Collect next to the analysis type that you need to run. For example, run Survey Analysis.

**View Result**

After the collection is over, Intel Advisor presents the collected results in Report tabs.
After investigating the vectorization blocking factors, studying Advisor recommendations, and making decisions of further optimizations, create a Snapshot.

**Create Snapshot**

Create a snapshot of the analysis results so that you can compare performance statistics before and after making changes in the target application’s code. To create a snapshot,

- In Intel Advisor GUI, go to **File > Create Data Snapshot**, and configure the snapshot parameters.
- In the Microsoft Visual Studio: go to **Tools > Intel Advisor [version] > Create Data Snapshot**.

The Snapshot opens in a separate tab with run analysis buttons unavailable and the (read only) label on the tab selector.

**Re-run the Analysis and Compare Results**

After making changes in the source code and recompiling the application, you need to re-run the analysis to see if the application performance changed as you expected.

To re-run the analysis, switch back to the result tab, and click the same **Collect** button you clicked before. Once the analysis is over, you can compare the current results with the ones collected earlier.

**See Also**
- Intel Advisor Menus and Toolbars
- Intel Advisor Workflow Tab
- Set Up Environment Variable

**Create a Project Dialog Box**

**Create a Project Dialog Box Purpose and Usage**

Intel Advisor is based on a project paradigm and requires that you create or open a project to enable analysis features. Think of a project as a reusable container for:
• The location of a compiled application
• A collection of configurable properties
• An analysis result

Create a Project Dialog Box Access
From the Intel Advisor GUI, do one of the following:
• Choose File > New > Project....
• Click the toolbar icon.
• Click the Welcome page New Project... link.

Create a Project Dialog Box Controls

<table>
<thead>
<tr>
<th>Use This</th>
<th>To Do This</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Project name</strong> field</td>
<td>Specify the name of the Intel Advisor project. This might be similar to the target executable name. The project name is used for the project directory name:</td>
</tr>
<tr>
<td></td>
<td>• A project file that identifies the target to be analyzed and a set of configurable attributes for running the target.</td>
</tr>
<tr>
<td></td>
<td>• Results that allows you to view the collected data.</td>
</tr>
<tr>
<td><strong>Location</strong> field and <strong>Browse</strong> button</td>
<td>Choose or create a directory to contain the project directory. Click the <strong>Browse</strong> button to browse to and select a directory where the project directory will be created.</td>
</tr>
<tr>
<td></td>
<td>Project files should be located in a different directory than your source directories, such as a directory above the source directories or in a separate projects directory. You must have write permission to the specified directory and its subdirectories.</td>
</tr>
<tr>
<td><strong>Create project</strong> button</td>
<td>After entering the <strong>Project name</strong> and specifying its <strong>Location</strong>, click <strong>Create project</strong> to create the project and its directory and display the Analysis Target tab of the <strong>Project Properties</strong> dialog box.</td>
</tr>
</tbody>
</table>

Project Navigator Pane
**Pane Purpose and Usage**
Use this pane to view, modify, and open existing Intel Advisor results.

**Project Navigator Pane Access**
Choose View > Project Navigator.

**Project Navigator Pane Controls**

<table>
<thead>
<tr>
<th>Use This</th>
<th>To Do This</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Title bar</strong></td>
<td>Drag to move the <strong>Project Navigator</strong> pane.</td>
</tr>
<tr>
<td></td>
<td>Drag to a window edge to dock the <strong>Project Navigator</strong> pane.</td>
</tr>
<tr>
<td><strong>Path to project directory</strong></td>
<td>View the location of the currently opened project.</td>
</tr>
<tr>
<td>Use This</td>
<td>To Do This</td>
</tr>
<tr>
<td>----------</td>
<td>------------</td>
</tr>
<tr>
<td></td>
<td>Right-click the path to access the directory context menu.</td>
</tr>
</tbody>
</table>
| Project name | Double-click to open the project.  
Right-click to access the project context menu. |
| **NOTE** | Opening a project closes the currently opened project. |
| Result name | Double-click to open the result.  
Right-click to access the result context menu. |
| **NOTE** | Opening a result opens the associated project. |

**Intel Advisor Menus and Toolbars**

After you launch the Microsoft Visual Studio* (on Windows* OS) or Intel® Advisor GUI environment and choose your target, you can begin the process of using the Intel Advisor tools to help you add parallelism into parts of your program.

To launch the Intel Advisor tools, use one of the following:

- In the Visual Studio **Tools** menu, click **Intel Advisor [version]** and choose the desired action.
- In the Intel Advisor GUI, click **File > New** and choose the desired action.
- In the Intel Advisor toolbar, select the desired action by clicking the appropriate icon in a drop-down list:

In the Intel Advisor GUI, click the downward arrow icon to display a drop-down list:
In Microsoft Visual Studio, click the downward arrow icon to display a drop-down list:

![Image of a drop-down list with various options]

To display the installed help or the Getting Started page, or view online videos or articles, click the or icon on the Intel Advisor toolbar. The Getting Started page introduces the Intel Advisor and provides links to the installed tutorials, documentation, and online internet resources.

The following table lists the actions available:

<table>
<thead>
<tr>
<th>To Do This</th>
<th>Use the Menu</th>
<th>Use the Advisor Workflow Tab</th>
<th>Use the Toolbars</th>
</tr>
</thead>
<tbody>
<tr>
<td>Run the Survey analysis on your application (target) to profile it to locate hotspots.</td>
<td>In Visual Studio, click Tools &gt; Intel Advisor [version] &gt; Start Survey Analysis In the product GUI, click File &gt; New &gt; Start Survey Analysis</td>
<td>Below Survey Target, click the Collect button or . Once you start the analysis, pause , stop , and cancel buttons appear.</td>
<td>Select Start Survey Analysis in the Advisor toolbar list.</td>
</tr>
<tr>
<td>Run the Trip Counts and FLOP analysis on your application (target) to collect loop iteration statistics.</td>
<td>In Visual Studio, click Tools &gt; Intel Advisor [version] &gt; Start Trip Counts and FLOP Analysis In the product GUI, click File &gt; New &gt; Start Trip Counts and FLOP Analysis</td>
<td>Below Find Trip Counts and FLOP click the Collect button. By default, only Trip Counts are collected. To collect FLOP data as well, select the FLOP checkbox under the Collect button. Once you start the analysis, stop</td>
<td>Select Start Trip Counts and FLOP Analysis in the Advisor toolbar list.</td>
</tr>
<tr>
<td>To Do This</td>
<td>Use the Menu</td>
<td>Use the Advisor Workflow Tab</td>
<td>Use the Toolbars</td>
</tr>
<tr>
<td>--------------------------------------------------------------------------</td>
<td>-----------------------------------------------------------------------------------------------------------------------------------------------</td>
<td>---------------------------------------------------------------------------------------------</td>
<td>-----------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>Run the Roofline analysis [ ] on your application to visualize actual performance against hardware-imposed performance ceilings.</td>
<td>In Visual Studio, click <strong>Tools &gt; Intel Advisor [version] &gt; Start Roofline Analysis.</strong>&lt;br&gt;    In the product GUI, click <strong>File &gt; New &gt; Start Roofline Analysis.</strong></td>
<td>Below Run Roofline, click the <strong>Collect</strong> button. Once you start the analysis, stop&lt;br&gt;    □ and cancel&lt;br&gt;    × buttons appear.</td>
<td>Select <strong>Start Roofline Analysis</strong> in the Advisor toolbar list.</td>
</tr>
<tr>
<td>Run the Memory Access Patterns analysis [ ] on your application (target) to collect data on memory access strides.</td>
<td>In Visual Studio, click <strong>Tools &gt; Intel Advisor [version] &gt; Start Trip Counts Analysis</strong>&lt;br&gt;    In the product GUI, click <strong>File &gt; New &gt; Start Memory Access Patterns Analysis</strong></td>
<td>In the Vectorization Workflow below Check Memory Access Patterns click the <strong>Collect</strong> button. Once you start the analysis, stop&lt;br&gt;    □ and cancel&lt;br&gt;    × buttons appear.</td>
<td>Select <strong>Start Memory Access Patterns Analysis</strong> in the Advisor toolbar list.</td>
</tr>
<tr>
<td>Run the Dependencies analysis [ ] on your application to predict its potential data sharing problems. Before running this analysis, add source annotations to define at least one parallel site and task(s) within each site.</td>
<td>In Visual Studio, click <strong>Tools &gt; Intel Advisor [version] &gt; Start Dependencies Analysis</strong>&lt;br&gt;    In the product GUI, click <strong>File &gt; New &gt; Start Dependencies Analysis</strong></td>
<td>Below Check Dependencies click the <strong>Collect</strong> button. Once you start the analysis, stop&lt;br&gt;    □ and cancel&lt;br&gt;    × buttons appear.</td>
<td>Select <strong>Start Dependencies Analysis</strong> in the Advisor toolbar list.</td>
</tr>
<tr>
<td>Run the Suitability analysis [ ] on your application to predict its approximate parallel performance. Before running this tool, add source</td>
<td>In Visual Studio, click <strong>Tools &gt; Intel Advisor [version] &gt; Start Suitability Analysis</strong>&lt;br&gt;    In the product GUI, click <strong>File &gt; New &gt; Start Suitability Analysis</strong></td>
<td>In the Threading Workflow tab below Check Suitability click the <strong>Collect</strong></td>
<td>Select <strong>Start Suitability Analysis</strong> in the Advisor toolbar list.</td>
</tr>
<tr>
<td>To Do This</td>
<td>Use the Menu</td>
<td>Use the Advisor Workflow Tab</td>
<td>Use the Toolbars</td>
</tr>
<tr>
<td>-----------</td>
<td>-------------</td>
<td>-------------------------------</td>
<td>------------------</td>
</tr>
<tr>
<td>annotations to define at least one parallel site and task(s) within each site.</td>
<td>button or ¦. Once you start the analysis, pause ¦, stop ¦, and cancel ¦ buttons appear.</td>
<td>Create the data snapshot (read-only version) © of the current project.</td>
<td>Select Create Data Snapshot in the Advisor toolbar list.</td>
</tr>
<tr>
<td>Create the data snapshot (read-only version) © of the current project.</td>
<td>In Visual Studio, click Tools &gt; Intel Advisor [version] &gt; Create Data Snapshot</td>
<td>In the product GUI, find the button at the top of the Report window next to Where should I add vectorization and/or threading parallelism?.</td>
<td></td>
</tr>
<tr>
<td>Open the Project Properties dialog to view and modify project properties.</td>
<td>In Visual Studio, click Tools &gt; Intel Advisor [version] &gt; Project Properties</td>
<td>In the product GUI, click File &gt; Project Properties...</td>
<td>Click (if available) in the Intel Advisor toolbar.</td>
</tr>
<tr>
<td>In the product GUI, open or close the Project navigator.</td>
<td>In the product GUI, click View &gt; Project Navigator</td>
<td></td>
<td>Click in the Intel Advisor toolbar.</td>
</tr>
<tr>
<td>In the product GUI, create a new project.</td>
<td>In the product GUI, click File &gt; New &gt; Project...</td>
<td></td>
<td>Click in the Intel Advisor toolbar.</td>
</tr>
<tr>
<td>In the product GUI, open an existing project.</td>
<td>In the product GUI, choose either:</td>
<td></td>
<td>Click the button in the Intel Advisor toolbar.</td>
</tr>
<tr>
<td>• File &gt; Recent Projects&gt; name</td>
<td>• File &gt; Open &gt; Project...</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Temporarily stop data collection while the target continues to run.</td>
<td>Click the button for the Survey Report, Suitability Report, or Dependencies Report in the Workflow tab.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>To Do This</td>
<td>Use the Menu</td>
<td>Use the Advisor Workflow Tab</td>
<td>Use the Toolbars</td>
</tr>
<tr>
<td>---------------------------------------------------------------------------</td>
<td>------------------------------------------------------------------------------</td>
<td>------------------------------------------------------------</td>
<td>--------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>Resume data collection that you previously paused or that was started as paused.</td>
<td></td>
<td>Click the button in the workflow tab.</td>
<td></td>
</tr>
<tr>
<td>Stop target execution and data collection, and display the result collected thus far. This button is enabled when the collection is in progress.</td>
<td></td>
<td>Click the button for the Survey Report, Suitability Report, or Dependencies Report in the Workflow tab.</td>
<td></td>
</tr>
<tr>
<td>Cancel the target execution and data collection. Discards the data collected so far. This button is enabled when the collection is in progress.</td>
<td></td>
<td>Click the button for the Survey Report, Suitability Report, or Dependencies Report in the Workflow tab.</td>
<td></td>
</tr>
<tr>
<td>Cancel data finalization. Retains the data collected. This button is enabled when data finalization is in progress.</td>
<td></td>
<td>Click the Cancel button in the Survey Report or Suitability Report window's command toolbar.</td>
<td></td>
</tr>
</tbody>
</table>

**Corresponding advixe-cl Command Options Dialog Box**

**Dialog Box Purpose and Usage** | Dialog Box Access | Dialog Box Controls

**Corresponding advixe-cl Command Options Dialog Box Purpose and Usage**
Use this dialog box to get the command-line for generating report with specified options.

**Corresponding advixe-cl Command Options Dialog Box Access**
To access the command-line options dialog box, in Intel® Advisor Workflow Tab click the Get Command Line button next to the needed analysis name.

**Corresponding advixe-cl Command Options Dialog Box Controls**

<table>
<thead>
<tr>
<th>Use This</th>
<th>To Do This</th>
</tr>
</thead>
<tbody>
<tr>
<td>Command Line text box</td>
<td>View the full command line for running the required analysis type with the selected options.</td>
</tr>
</tbody>
</table>
**Intel Advisor CLI**

*Steps to ramp up with Intel® Advisor CLI and get quick results.*

This section provides simple steps to get quick results and grasp the overall idea of how you can use Intel® Advisor.

For detailed information on using the command line to predict performance with the Intel® Advisor Beta, refer to the Offload Advisor.

**Run the Intel® Advisor Beta CLI**

The Intel® Advisor Beta CLI is available from a command line on Windows® or Linux® OS systems.

**Run Analysis**

To run an analysis from the CLI, you need to run the following command:

```
advixe-cl --collect=survey -- <target>
```

This command creates new project and runs the Survey analysis for the target application.

**View Result**

To view the analysis result, use the following command:

```
advixe-cl --report=survey
```

This command prints the analysis result into the terminal window.

**Create Snapshot**

Create a snapshot of the analysis results so that you can compare performance statistics before and after making changes in the target application’s code. To create a snapshot run the following command:

```
advixe-cl --snapshot
```

The command creates snapshot in the current project directory with auto-generated name.

**Re-run the Analysis**

After making changes in the source code and recompiling the application, you need to re-run the analysis to see if the application performance changed as you expected. To re-run the analysis,

```
advixe-cl --collect=survey -- <target>
```

**Setting and Using Intel® Advisor Environment Variables**

Intel® Advisor:
Provides the ADVISOR_<version>_DIR environment variable that you can use to specify additional include directories when compiling, so the compiler can find the Intel® Advisor include file that defines annotations. <version> is the Advisor year version (for example, 2019 for Intel® Advisor 2019 or 2021 for Intel® Advisor Beta).

To specify the environment variable:

- **On Windows* OS** define the environment variable when you use the predefined command prompt or execute the environment script file.
- **On Linux* OS and MacOS* define the environment variable by using an export command.
- Tests the ADVIXE_EXPERIMENTAL environment variable to determine whether evaluation features have been activated.
- With C++ parallel Intel® Threading Building Blocks (Intel® TBB) programs, the TBBROOT environment variable must be defined, so your compiler can locate the installed Intel TBB include directory. See the help topic Defining the TBBROOT Environment Variable.
- **On Linux* OS and MacOS* Intel® Advisor uses the BROWSER environment variable on Linux* systems to locate an installed HTML browser. This enables the display of Getting Started Tutorials or Help from the Intel® Advisor GUI Help menu.
- **On Linux* OS and MacOS*, in addition to the File > Option> Editor settings, you can use the VISUAL or EDITOR environment variable to specify the external editor to launch when you double-click a line in a Source window. (VISUAL takes precedence over EDITOR). For example, to set the VISUAL environment variable, use an export command: export VISUAL="/usr/bin/vi -n"

To view and set the ADVISOR_<version>_DIR environment variable:

### Setting Environment Variables

In the instructions below, be sure to replace any values in brackets, such as <version> or <install-dir>. The default installation path for the application, <install-dir>, can be found inside the following:

- **For Linux* OS and macOS*, <install-dir> is inside:
  - /opt/intel/ or /opt/intel/oneapi/ (Intel® Advisor Beta) for root users
  - $HOME/intel/ or $HOME/intel/oneapi/ (Intel® Advisor Beta) for non-root users
  For example, <install-dir> may be /opt/intel/advisor_2019 for Intel Advisor 2019 or /opt/intel/oneapi/advisor/<version> for Intel® Advisor Beta.
- **For Windows* OS, <install-dir> is inside:
  - C:\Program Files (x86)\IntelSWTools\ for Intel Advisor.
  - C:\Program Files (x86)\Intel\oneAPI\ for Intel® Advisor Beta.
  For 32-bit systems, the Program files (x86) folder is Program files.
  For example, <install-dir> may be C:\Program Files (x86)\IntelSWTools\Advisor 2019 for Intel® Advisor 2019 or C:\Program Files (x86)\Intel\oneAPI\advisor\<version> for Intel® Advisor Beta.

### On Linux* OS / MacOS* Manually

1. View the current definition of the environment variable. For example, with the bash shell, type: env | grep ADVISOR_<version>_DIR
2. Use an export command to set the environment variable. Type: export ADVISOR_<version>_DIR="<install-dir>"
   For example: export ADVISOR_2019_DIR="/opt/intel/advisor_2019" for Intel Advisor 2019 or export ADVISOR_2021_DIR="/opt/intel/oneapi/advisor/<version>" for Intel® Advisor Beta
3. To always set this variable on the current system, add this definition to your .login or similar shell initialization file.
4. To test the definition of an environment variable, type an `export` command.

**On Linux* OS / MacOS* via Script**

Run the shell script, as follows:

- For Intel Advisor: `source <install-dir>/advixe-vars.sh` or `source <install-dir>/advixe-vars.csh`
- For Intel® Advisor Beta: `source <install-dir>/env/vars.sh`

**On Windows* OS Manually**

1. Open a command line window.
2. View the current definition of the environment variable. For example, type: `set ADVISOR_<version>_DIR`
3. Use a `set` command to set the environment variable. Type: `set ADVISOR_<version>_DIR="<install-dir>"`
   
   For example: `set ADVISOR_2019_DIR="C:\Program files (x86)\IntelSWTools\Advisor 2019"` or `set ADVISOR_2021_DIR="C:\Program Files (x86)\Intel\oneAPI\advisor\<version>"` (for Intel® Advisor Beta)
4. To always set this variable on the current system, add this definition to your system or user environment variables using Control Panel > System and Security > System > Advanced system settings > Environment Variables....

**On Windows* OS via Script**

Run the batch script, as follows:

- `<install-dir>/advixe-vars.bat`
- `<install-dir>/env/vars.bat` (for Intel® Advisor Beta)

**See Also**

Limiting the Number of Threads Used by Parallel Frameworks
Intel Advisor Annotation Definitions File
Defining the TBBROOT Environment Variable (for C++ programs using Intel TBB )

**Launch Intel® Advisor Beta from a Docker* Container for Linux* OS**

This section contains steps to run Intel® Advisor Beta in a Docker* container. Containers allow you to set up and configure environments and distribute them using images:

- You can install an image containing an environment pre-configured with all the tools you need, then develop within that environment.
- You can save an environment and use the image to move that environment to another machine without additional setup.
- You can prepare containers with different sets of compilers, tools, libraries, or other components, as needed.

This is a feature of Intel® Advisor Beta installed as part of the Intel® oneAPI Base Toolkit

**Set up the Docker container**

1. Pull the Docker image from the Intel oneAPI Containers Repository with the following commands:

   ```bash
   image=amr-registry.caas.intel.com/oneapi/oneapi:base-dev-ubuntu18.04
docker pull "$image"
   ```

2. Run the Docker container using the following command:

   ```bash
   docker run --cap-add=SYS_PTRACE -it "$image"
   ```
3. For the rest of the steps in this section, run any commands from the command line prompt inside the Docker container.
   For example, to set up the Mandelbrot sample, you can run:

   ```
   cd /one-api-code-samples/HPC/mandelbrot
   make
   ./main -d1
   ./main -t gpu # run on gpu
   ./main -t cpu # run on cpu
   make clean
   ```

4. Run the following commands to source Advisor variables:

   ```
   source /opt/intel/oneapi/advisor/latest/advixe-vars.sh
   ```

5. Now that your Docker container is running, you can run Advisor from the command line as you would without a container. For example:

   ```
   /opt/intel/oneapi/advisor/latest/bin64# advixe-cl --collect survey /bin/ls
   ```

   • For steps to run Intel® Advisor using the command line interface, see the Intel Advisor CLI section.
   • For steps to run the Offload Advisor of Intel® Advisor Beta, see the Intel® Advisor Beta: Offload Advisor.

## Configuration

### Configuration Windows Purpose and Usage

Intel Advisor configuration is available thru several windows:

- **Project Properties** dialog box - Configure how the Intel Advisor analyses process your target executable.
- **Options** dialog box - Configure the Intel Advisor user interface behavior.
- **Project Navigator** pane - Select projects, results, and snapshots that the Intel Advisor shows in the user interface.
- **Create a Project** dialog box - Configure properties of a new Intel Advisor project.
- **Create a Result Snapshot** dialog box - Configure and save your Intel Advisor project’s result snapshot.
- **Build Settings for Supported Languages** dialog box - Configure your application for the Intel Advisor analyses.
- **Corresponding advixe-cl Command Options** dialog box - Get command line to collect data in console. This dialog generates command line with the set of options that you define in Project Properties and Workflow Tab.
Configuration Menus and Toolbars Locations

<table>
<thead>
<tr>
<th>Icon</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image" alt="Edit" /></td>
<td>Toggle the <strong>Project Navigator</strong> pane.</td>
</tr>
<tr>
<td><img src="image" alt="Options" /></td>
<td>Open the <strong>Project Properties</strong> dialog box.</td>
</tr>
<tr>
<td><img src="image" alt="Options" /></td>
<td>Open the <strong>Create Result Snapshot</strong> dialog box.</td>
</tr>
<tr>
<td><img src="image" alt="Options" /></td>
<td>Open the <strong>Get Command Line</strong> dialog box.</td>
</tr>
<tr>
<td><img src="image" alt="Options" /></td>
<td>Open the <strong>Options</strong> dialog box.</td>
</tr>
</tbody>
</table>

**Project Properties Dialog Box**

**Dialog Box Purpose and Usage**
**Dialog Box Access**
**Dialog Box Controls**

**Project Properties Purpose and Usage**

The **Project Properties** dialog box consists of the following tabs:

- **Analysis Target Tab** - Specify the target executable, set important project properties, and review current project properties.
- **Binary/Symbol Search Tab** - Specify non-standard directories for the supporting files needed to execute and analyze the target. With Visual Studio* on Windows* OS, you can instead use the Visual Studio solution and project capabilities to search for specific directories.
- **Source Search Tab** - Specify the source search locations needed to execute and analyze the target. With Visual Studio, some source locations are pre-populated from the Visual Studio startup project into the internal representation of Intel Advisor project properties, so you may not need to add new row(s).

**Tip**
Always check project property values before analyzing a new target.
**Project Properties Dialog Box Access**

To access this dialog box:

- From the Intel Advisor GUI, choose **Project > Project Properties.**
- From the Visual Studio® menu, choose **Project > Intel Advisor [version] Project Properties...**

**Project Properties Dialog Box Controls**

1. **Tab Selector** - Select among the **Project Properties** tabs. The following tabs are available:
   - Analysis Target Tab
   - Binary/Symbol Search Tab
   - Source Search Tab

2. **Tab Settings** - Configure settings for the selected tab.
See Also
- Binary/Symbol Search and Source Search Locations dialog box

Analysis Target Tab

Analysis Target Tab Purpose and Usage
Use this tab in the Project Properties dialog box to specify the target executable, set important project properties, and review current project properties.

Tip
Always check project property values before analyzing a new target.

Analysis Target Tab Location
One of the tabs in the Project Properties dialog box, which also includes:
- Binary/Symbol Search tab
- Source Search tab

To access this tab:
- From the Intel Advisor GUI, choose Project > Project Properties.
- From the Visual Studio* menu, choose Project > Intel Advisor [version] Project Properties...

Analysis Target Tab Overview
On the Analysis Target tab, select an analysis type from list (on the left) to display and set project properties.

| Analysis Type selector | Select an analysis type to configure. Different project properties are available in the Analysis Properties region depending on the analysis type selected. The following analysis types are available:
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>• Survey Analysis Types</td>
</tr>
<tr>
<td></td>
<td>• Survey Hotspots Analysis</td>
</tr>
<tr>
<td></td>
<td>• Trip Counts and FLOP analysis</td>
</tr>
<tr>
<td></td>
<td>• Suitability Analysis</td>
</tr>
<tr>
<td></td>
<td>• Refinement Analysis Types</td>
</tr>
<tr>
<td></td>
<td>• Memory Access Patterns Analysis</td>
</tr>
<tr>
<td></td>
<td>• Dependencies Analysis</td>
</tr>
</tbody>
</table>

| Analysis Properties | Set project properties for the analysis type selected in the Analysis Type region. |

Analysis Target Tab Controls
The following table covers project properties applicable to all analysis types. To view controls applicable only to a specific analysis type, use the links immediately below:
- Survey Analysis Controls
- Trip Counts and FLOPS Controls
- Suitability Analysis Controls
- MAP Analysis Controls
- Dependencies Analysis Controls

### Common Controls

<table>
<thead>
<tr>
<th>Use This</th>
<th>To Do This</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Target type</strong> drop-down</td>
<td>• Analyze an executable or script (choose <strong>Launch Application</strong>).</td>
</tr>
<tr>
<td></td>
<td>• Analyze a process (choose <strong>Attach to Process</strong>).</td>
</tr>
<tr>
<td></td>
<td>If you choose <strong>Attach to Process</strong>, you can either inherit settings from the <strong>Survey Hotspots Analysis Type</strong> or specify the needed settings.</td>
</tr>
<tr>
<td><strong>Inherit settings from Visual Studio project</strong> checkbox and field</td>
<td>Inherit Intel Advisor project properties from the Visual Studio* startup project (enable).</td>
</tr>
<tr>
<td>(Visual Studio* IDE only)</td>
<td>If enabled, the <strong>Application</strong>, <strong>Application parameters</strong>, and <strong>Working directory</strong> fields are pre-filled and cannot be modified.</td>
</tr>
<tr>
<td><strong>Application</strong> field and <strong>Browse...</strong> button</td>
<td>Select an analysis target executable or script.</td>
</tr>
<tr>
<td></td>
<td>If you specify a script in this field, consider specifying the executable in the <strong>Advanced &gt; Child application</strong> field (required for Dependencies analysis).</td>
</tr>
<tr>
<td><strong>Application parameters</strong> field and <strong>Modify...</strong> button</td>
<td>Specify runtime arguments to use when performing analysis (equivalent to command line arguments).</td>
</tr>
<tr>
<td><strong>Use application directory as working directory</strong> checkbox</td>
<td>Automatically use the value in the <strong>Application directory</strong> to pre-fill the <strong>Working directory</strong> value (enable).</td>
</tr>
<tr>
<td><strong>Working directory</strong> field and <strong>Browse...</strong> button</td>
<td>Select the working directory.</td>
</tr>
<tr>
<td><strong>User-defined environment variables</strong> field and <strong>Modify...</strong> button</td>
<td>Specify environment variables to use during analysis.</td>
</tr>
<tr>
<td><strong>Managed code profiling mode</strong> drop-down</td>
<td>• Automatically detect the type of target executable as Native or Managed, and switch to that mode (choose <strong>Auto</strong>).</td>
</tr>
<tr>
<td></td>
<td>• Collect data for native code and do not attribute data to managed code (choose <strong>Native</strong>).</td>
</tr>
<tr>
<td></td>
<td>• Collect data for both native and managed code, and attribute data to managed code as appropriate (choose <strong>Mixed</strong>). Consider using this option when analyzing a native executable that makes calls to managed code.</td>
</tr>
<tr>
<td></td>
<td>• Collect data for both native and managed code, resolve samples attributed to native code, and attribute data to managed source only (choose <strong>Managed</strong>). The call stack in the analysis result displays data for managed code only.</td>
</tr>
<tr>
<td><strong>Child application</strong> field</td>
<td>Analyze a file that is not the starting application. For example: Analyze an executable (identified in this field) called by a script (identified in the <strong>Application</strong> field).</td>
</tr>
<tr>
<td></td>
<td>Invoking these properties could decrease analysis overhead.</td>
</tr>
<tr>
<td>Use This</td>
<td>To Do This</td>
</tr>
<tr>
<td>----------</td>
<td>------------</td>
</tr>
<tr>
<td><strong>NOTE</strong></td>
<td>For the <strong>Dependencies Analysis Type</strong>: If you specify a script file in the <strong>Application</strong> field, you must specify the target executable in the <strong>Child application</strong> field.</td>
</tr>
</tbody>
</table>
| **Modules** radio buttons, field, and **Modify...** button | • Analyze specific modules and disable analysis of all other modules (click the **Include only the following module(s)** radio button and choose the modules).  
• Disable analysis of specific modules and analyze all other modules (click the **Exclude only the following module(s)** radio button and choose the modules). |
| **Use MPI launcher** checkbox | Generate a command line (enable) that appears in the **Get command line** field based on the following parameters:  
• **Select MPI Launcher** - Intel or another vendor  
• **Number of ranks** - Number of instances of the application  
• **Profile ranks** - All or a range of ranks to profile | |
| **Automatically stop collection after (sec)** checkbox and field | Stop collection after a specified number of seconds (enable and specify seconds).  
Invoking this property could minimize analysis overhead. | |

**Survey Analysis-Specific Controls**

<table>
<thead>
<tr>
<th>Use This</th>
<th>To Do This</th>
</tr>
</thead>
</table>
| **Automatically resume collection after (sec)** checkbox and field | Start running your target application with collection paused, then resume collection after a specified number of seconds (enable and specify seconds).  
Invoking this property could decrease analysis overhead. |
| **Tip** | The corresponding CLI action option is `--resume-after=<integer>`, where the integer argument is in milliseconds, not seconds. |
| **Sampling Interval** selector | Set the wait time between each analysis collection CPU sample while your target application is running.  
Increasing the wait time could decrease analysis overhead. |
| **Collection data limit, MB** selector | Set the amount of collected raw data if exceeding a size threshold could cause issues. Not available for hardware event-based analyses.  
Decreasing the limit could decrease analysis overhead. |
<table>
<thead>
<tr>
<th>Use This</th>
<th>To Do This</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Stack unwinding mode</strong></td>
<td><strong>Set to After collection</strong> if:</td>
</tr>
<tr>
<td>drop-down list</td>
<td>• Survey analysis runtime overhead exceeds 1.1x.</td>
</tr>
<tr>
<td></td>
<td>• A large quantity of data is allocated on the stack, which is a common case for Fortran applications or applications with a large number of small, parallel, OpenMP* regions.</td>
</tr>
<tr>
<td></td>
<td>Otherwise, set to <strong>During Collection</strong>.</td>
</tr>
<tr>
<td><strong>Stitch stacks</strong> checkbox</td>
<td>Restore a logical call tree for Intel® Threading Building Blocks (Intel® TBB) or OpenMP* applications by catching notifications from the runtime and attaching stacks to a point introducing a parallel workload (enable).</td>
</tr>
<tr>
<td></td>
<td>Disable if Survey analysis runtime overhead exceeds 1.1x.</td>
</tr>
<tr>
<td><strong>Analyze MKL Loops and</strong></td>
<td>Show Intel® Math Kernel Library (Intel® MKL) loops and functions in Intel Advisor reports (enable).</td>
</tr>
<tr>
<td><strong>Functions</strong> checkbox</td>
<td>Enabling could increase analysis overhead.</td>
</tr>
<tr>
<td><strong>Analyze Python loops and</strong></td>
<td>Show Python* loops and functions in Intel Advisor reports (enable).</td>
</tr>
<tr>
<td><strong>functions</strong> checkbox</td>
<td>Enabling could increase analysis overhead.</td>
</tr>
<tr>
<td><strong>Analyze loops that reside</strong></td>
<td>Collect a variety of data during analysis for loops that reside in non-executed code paths, including loop assembly code, instruction set architecture (ISA), and vector length (enable).</td>
</tr>
<tr>
<td><strong>in non-executed code</strong></td>
<td>Enabling could increase analysis overhead.</td>
</tr>
<tr>
<td><strong>paths</strong> checkbox</td>
<td></td>
</tr>
<tr>
<td><strong>NOTE</strong></td>
<td>Analyzing non-executed code paths in binaries that target multiple ISAs (contain multiple code paths) is available only for binaries compiled using the -ax (Linux* OS) / Qax (Windows* OS) option with an Intel compiler.</td>
</tr>
<tr>
<td><strong>Enable registry spill/fill</strong></td>
<td>Calculate the number of consecutive load/store operations in registers and related memory traffic (enable).</td>
</tr>
<tr>
<td><strong>analysis</strong> checkbox</td>
<td>Enabling could increase analysis overhead.</td>
</tr>
<tr>
<td><strong>Enable static instruction</strong></td>
<td>Statically calculate the number of specific instructions present in the binary (enable).</td>
</tr>
<tr>
<td><strong>mix analysis</strong> checkbox</td>
<td>Enabling could increase analysis overhead.</td>
</tr>
<tr>
<td><strong>Source caching</strong> drop-down</td>
<td>• Delete source code cache from a project with each analysis run (default; choose Clear cached files).</td>
</tr>
<tr>
<td><strong>list</strong></td>
<td>• Keep source code cache within the project (choose Keep cached files).</td>
</tr>
</tbody>
</table>
## Trip Counts and FLOP Analysis-Specific Controls

<table>
<thead>
<tr>
<th>Use This</th>
<th>To Do This</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Inherit settings from the Survey Hotspots Analysis Type</strong> checkbox</td>
<td>Copy similar settings from Survey analysis properties (enable). When enabled, this option disables application parameters controls.</td>
</tr>
<tr>
<td><strong>Automatically resume collection after (sec)</strong> checkbox and field</td>
<td>Start running your target application with collection paused, then resume collection after a specified number of seconds (enable and specify seconds). Invoking this property could decrease analysis overhead.</td>
</tr>
<tr>
<td></td>
<td><strong>Tip</strong> The corresponding CLI action option is --resume-after=&lt;integer&gt;, where the integer argument is in milliseconds, not seconds.</td>
</tr>
<tr>
<td><strong>Collect information about Loop Trip Counts</strong> checkbox</td>
<td>Measure loop invocation and execution (enable).</td>
</tr>
<tr>
<td><strong>Collect information about FLOP, L1 memory traffic, and AVX-512 mask usage</strong> checkbox</td>
<td>Measure floating-point operations, integer operations, and memory traffic (enable).</td>
</tr>
<tr>
<td><strong>Collect callstacks</strong> checkbox</td>
<td>Collect call stack information when performing analysis (enable). Enabling could increase analysis overhead.</td>
</tr>
<tr>
<td><strong>Capture metrics for dynamic loops and functions</strong> checkbox</td>
<td>Collect metrics for dynamic Just-In-Time (JIT) generated code regions.</td>
</tr>
<tr>
<td><strong>Capture metrics for stripped binaries</strong> checkbox</td>
<td>Collect metrics for stripped binaries. Enabling could increase analysis overhead.</td>
</tr>
<tr>
<td><strong>Enable Memory-Level Roofline with cache simulation</strong> checkbox</td>
<td>Model multiple levels of cache for data, such as counts of loaded or stored bytes for each loop, to plot the Roofline chart for all memory levels (enable). Enabling could increase analysis overhead.</td>
</tr>
<tr>
<td><strong>Cache simulator configuration</strong> field</td>
<td>Specify a cache hierarchy configuration to model (enable and specify hierarchy). The hierarchy configuration template is: <code>[num_of_level1_caches]:[num_of_ways_level1_connected]:[level1_cache_size]:[level1_cacheline_size]/ [num_of_level2_caches]:[num_of_ways_level2_connected]:[level2_cache_size]:[level2_cacheline_size]/ [num_of_level3_caches]:[num_of_ways_level3_connected]:[level3_cache_size]:[level3_cacheline_size]</code> For example: <code>4:8w:32k:64l/4:4w:256k:64l/1:16w:6m:64l</code> is the hierarchy configuration for:</td>
</tr>
<tr>
<td>Use This</td>
<td>To Do This</td>
</tr>
<tr>
<td>-----------------------------------------------</td>
<td>-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
</tbody>
</table>
| • Four eight-way 32-KB level 1 caches with line size of 64 bytes  
• Four four-way 256-KB level 2 caches with line size of 64 bytes  
• One sixteen-way 6-MB level 3 cache with line size of 64 bytes |                                                                                                                                                                                                          |

**Data transfer simulation mode** drop-down

Select a level of details for data transfer simulation:

• **Off** - Disable data transfer simulation analysis.
• **Light** - Model data transfers between host and device memory.
• **Full** - Model data transfers, attribute memory objects to loops that accessed the objects, and track accesses to stack memory.

---

### Suitability Analysis-Specific Controls

<table>
<thead>
<tr>
<th>Use This</th>
<th>To Do This</th>
</tr>
</thead>
</table>
| **Inherit settings from the Survey Hotspots Analysis Type** checkbox | Copy similar settings from Survey analysis properties (enable).  
When enabled, this option disables application parameters controls.                                                                                      |

**Automatically resume collection after (sec)** checkbox and field

Start running your target application with collection paused, then resume collection after a specified number of seconds (enable and specify seconds).

Invoking this property could decrease analysis overhead.

---

**Tip**

The corresponding CLI action option is `--resume-after=<integer>`, where the integer argument is in milliseconds, not seconds.

<table>
<thead>
<tr>
<th>Use This</th>
<th>To Do This</th>
</tr>
</thead>
</table>
| **Sampling Interval** selector               | Set the wait time between each analysis collection sample while your target application is running.  
Increasing the wait time could decrease analysis overhead.                                                                                      |

<table>
<thead>
<tr>
<th>Use This</th>
<th>To Do This</th>
</tr>
</thead>
</table>
| **Collection data limit, MB** selector        | Set the amount of collected raw data if exceeding a size threshold could cause issues. Not available for hardware event-based analyses.  
Decreasing the limit could decrease analysis overhead.                                                                                          |

---

### Memory Access Patterns Analysis-Specific Controls

<table>
<thead>
<tr>
<th>Use This</th>
<th>To Do This</th>
</tr>
</thead>
</table>
| **Inherit settings from the Survey Hotspots Analysis Type** checkbox | Copy similar settings from Survey analysis properties (enable).  
When enabled, this option disables application parameters controls.                                                                                      |

**Suppression mode** group box

• Report possible memory issues in system modules (choose the **Show problems in system modules** radio button).
• Do not report possible memory issues in system modules (choose the **Suppress problems in system modules** radio button).
<table>
<thead>
<tr>
<th>Use This</th>
<th>To Do This</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Loop call count limit</strong></td>
<td>Choose the maximum number of instances each marked loop is analyzed. 0 = analyze all loop instances. Supplying a non-zero value could decrease analysis overhead.</td>
</tr>
<tr>
<td><strong>Instance of interest</strong></td>
<td>Analyze the nth child process, where 1 = the first process of the specified name in the application process tree. 0 = analyze all processes. Supplying a non-zero value could decrease analysis overhead.</td>
</tr>
<tr>
<td><strong>Report stack variables</strong></td>
<td>Report stack variables for which memory access strides are detected (enable). Enabling could increase analysis overhead.</td>
</tr>
<tr>
<td><strong>Report heap allocated variables</strong> checkbox</td>
<td>Report heap-allocated variables for which memory access strides are detected (enable). Enabling could increase analysis overhead.</td>
</tr>
<tr>
<td><strong>Enable CPU cache simulation</strong> checkbox</td>
<td>Model cache misses, cache misses and cache line utilization, or cache misses and loop footprint (enable and select desired options). Enabling could increase analysis overhead.</td>
</tr>
<tr>
<td><strong>Cache associativity</strong> drop-down list</td>
<td>Set the cache associativity for modeling CPU cache behavior. You can set the value to the following power-of-two integers: 1, 2, 4, 8, 16.</td>
</tr>
<tr>
<td><strong>Cache sets</strong> drop-down list</td>
<td>Set the cache set size (in bytes) for modeling CPU cache behavior. You can set the value to the following power-of-two integers: 256, 512, 1024, 2048, 4096, 8192.</td>
</tr>
<tr>
<td><strong>Cache line size</strong> drop-down list</td>
<td>Set the cache line size (in bytes) to model CPU cache behavior. You can set the value to the following power-of-two integers: 4, 8, 16, 32, ..., up to 65536.</td>
</tr>
</tbody>
</table>
| **Cache simulation mode** drop-down list      | Set the focus for modeling CPU cache behavior:  
  - **Model cache misses only.**  
  - **Model cache misses and memory footprint of a loop.** Calculation: Cache line size x Number of unique cache lines accessed during simulation.  
  - **Model cache misses and cache line utilization.**                                                                                                                                                                                                 |

**Dependencies Analysis Controls**

<table>
<thead>
<tr>
<th>Use This</th>
<th>To Do This</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Inherit settings from the Survey Hotspots Analysis Type</strong> checkbox</td>
<td>Copy similar settings from Survey analysis properties (enable). When enabled, this option disables application parameters controls.</td>
</tr>
<tr>
<td>Use This</td>
<td>To Do This</td>
</tr>
<tr>
<td>---------------------------------------</td>
<td>------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td><strong>Loop call count limit selector</strong></td>
<td>Choose the maximum number of instances each marked loop is analyzed. 0 = analyze all loop instances. Supplying a non-zero value could decrease analysis overhead.</td>
</tr>
<tr>
<td><strong>Instance of interest selector</strong></td>
<td>Analyze the nth child process, where 1 = the first process of the specified name in the application process tree. 0 = analyze all processes. Supplying a non-zero value could decrease analysis overhead.</td>
</tr>
<tr>
<td><strong>Analyze stack variables checkbox</strong></td>
<td>Analyze parallel data sharing for stack variables (enable). Enabling could increase analysis overhead.</td>
</tr>
</tbody>
</table>
| **Filter stack variables by scope checkbox** | Enable to report:  
• Variables initiated inside the loop as potential dependencies (warning)  
• Variables initialized outside the loop as dependencies (error)  
Enabling could increase analysis overhead. |
| **Filter reduction variables checkbox** | Mark all potential reductions by a specific diagnostic (enable). Enabling could increase analysis overhead.                                |

**Binary/Symbol Search Tab**

**Binary/Symbol Search Tab Purpose**

Use this tab in the Project Properties dialog box to specify non-standard directories for the supporting files needed to execute and analyze the target. With Visual Studio® on Windows® OS, you can instead use the Visual Studio solution and project capabilities to search for specific directories.

**Binary/Symbol Search Tab Location**

One of the tabs in the Project Properties dialog box, which also includes:

- Analysis Target tab
- Source Search tab

To access this tab:

- From the Intel Advisor GUI, choose Project > Project Properties. Then click the Binary/Symbol Search tab.
- From the Visual Studio® menu, choose Project > Intel Advisor [version] Project Properties.... Then click the Binary/Symbol Search tab.

**Binary/Symbol Search Tab Controls**

<table>
<thead>
<tr>
<th>Use This</th>
<th>To Do This</th>
</tr>
</thead>
<tbody>
<tr>
<td>[button]</td>
<td>On a row containing Add new search location, click to browse for directories to include in the search list. You can also type directly in the row. In addition to local directories, you can specify a symbol server URL.</td>
</tr>
<tr>
<td>[↑ and ↓ buttons]</td>
<td>Change the search order of the selected directory by moving it up or down. To select multiple rows, use the Ctrl or Shift keys.</td>
</tr>
<tr>
<td>Use This</td>
<td>To Do This</td>
</tr>
<tr>
<td>---------------------</td>
<td>---------------------------------------------------------------------------</td>
</tr>
<tr>
<td>❌ button</td>
<td>Delete a selected directory row(s).</td>
</tr>
<tr>
<td><strong>Search recursively</strong> checkbox</td>
<td>Enable to search the specified location subdirectories. To use recursive search, the lines must provide only a directory name and omit a file name. Using a recursive search for multiple directories may slow processing and could lead to unexpected results.</td>
</tr>
</tbody>
</table>

**See Also**
- Binary/Symbol Search and Source Search Locations

**Source Search Tab**

**Source Search Tab Purpose**
Use this tab in the *Project Properties* dialog box to specify the source search locations needed to execute and analyze the target. With Visual Studio, some source locations are pre-populated from the Visual Studio startup project into the internal representation of Intel Advisor project properties, so you may not need to add new row(s).

**Tip**
For Threading Advisor only: Intel® Advisor does not automatically populate source locations after you create a project using the Intel® Advisor GUI, so you must specify one or more locations to find application annotations. View the **Annotation Report** to verify all project annotations are found.

**Source Search Tab Location**
One of the tabs in the *Project Properties* dialog box, which also includes:
- Analysis Target tab
- Source Search tab

To access this tab:
- From the Intel® Advisor GUI, choose **Project > Project Properties**. Then click the **Source Search** tab.
- From the Visual Studio® menu, choose **Project > Intel Advisor [version] Project Properties...**. Then click the **Source Search** tab.

**Source Search Tab Controls**

<table>
<thead>
<tr>
<th>Use This</th>
<th>To Do This</th>
</tr>
</thead>
<tbody>
<tr>
<td>⋮ button</td>
<td>On a row containing <strong>Add new search location</strong>, click to browse for directories to include in the search list. You can also type directly in the row.</td>
</tr>
<tr>
<td>↑ and ↓ buttons</td>
<td>Change the search order of the selected directory by moving it up or down. To select multiple rows, use the Ctrl or Shift keys.</td>
</tr>
<tr>
<td>❌ button</td>
<td>Delete a selected directory row(s).</td>
</tr>
</tbody>
</table>
Use This | To Do This
---|---
**Search recursively checkbox** | Enable to search the specified location subdirectories. To use recursive search, the lines must provide only a directory name and omit a file name. Using a recursive search for multiple directories may slow processing and could lead to unexpected results.

**Mask text box** | Specify the file name mask pattern(s) to ignore (skip) using wildcard characters, such as an asterisk (*). For example, you can skip certain file suffixes.

**File text box** | Specify the file(s) to ignore (skip) using an absolute path.

To delete a row, use the button.

---

**See Also**
- Binary/Symbol Search and Source Search Locations

**Binary/Symbol Search and Source Search Locations**

When using the Intel Advisor Standalone GUI:

- Specify the locations of binary and symbol locations to be searched by using the **Project Properties** dialog box, **Binary/Symbol Search** tab. These locations will be searched in addition to the default binary and symbol locations.
- Specify the source locations to be searched by using the **Project Properties** dialog box, **Source Search** tab. These locations will be searched in addition to the default source search locations.

**Binary/Symbol Search Locations**

Certain default binary and symbol locations are used in addition to the locations specified in the **Binary/Symbol Search** tab. With Intel Advisor, you can use this tab to indicate whether or not the default binary and symbol locations (listed below) will be searched.

The following lists describe the order and default locations that are searched. As indicated below, some directory searches examine the specified directory and its subdirectories, while other searches do not examine its subdirectories.

The search order on Windows* OS systems is the following

1. Search for binary or symbol files in the directories specified in the **Project Properties**, **Binary/Symbol Search** tab. With Intel Advisor, you can indicate whether the subdirectories of these directories will be searched: select the **Search recursively** checkbox to include the subdirectories.

2. Search for symbol files in the directories near the related (corresponding) binary file(s) just found, such as a library:
   - Check in the directory of the corresponding binary file, using the corresponding name.
   - Check in the directory of the corresponding binary file, using a related name. For example, for app.dll where a file app_x86.pdb is present, also search for file app.pdb.

3. When using an integrated Visual Studio project, the directories provided by the Visual Studio project pre-populate the corresponding directories in the internal representation of the **Binary/Symbol Search** tab (for example, Visual Studio binary locations pre-populate the Project Properties binary locations). So, the Visual Studio project’s directories are searched and are specific to the selected configuration.

For symbol files, also search using symbol server paths specified in the **Project Properties**, **Binary/Symbol Search** tab in the following notation: srv*C:\localsymbols*http://msdl.microsoft.com/download/symbols and/or provided in Visual Studio Tools > Options > Debugging > Symbols.
4. Search for binary files in this standard Windows OS system directory:
   %SYSTEMROOT%\system32\drivers (subdirectories are not searched)

5. Search for symbol files in these standard Windows OS system directories:
   - All directories specified in the environment variable _NT_SYMBOL_PATH (subdirectories are not searched)
   - srv%SYSTEMROOT%\symbols (symbol downstream or cache path)
   - %SYSTEMROOT%\symbols\dll (subdirectories are not searched)

The search order on Linux* OS systems is the following:

1. Search for binary or symbol files in the directories specified in the **Project Properties, Binary/ Symbol Search** tab. With Intel Advisor, you can indicate whether the subdirectories of these directories will be searched: select the **Search recursively** checkbox to include the subdirectories.

2. Search for binary files in directories from the collected result that provide an absolute path name. If the file name vmlinux is present, search these directories:
   - /usr/lib/debug/lib/modules/`uname -r`/vmlinux
   - /boot/vmlinuz-`uname -r`

3. Search for symbol files in the directories near the related (corresponding) binary file(s) just found, such as a library:
   - Check in the directory of the corresponding binary file, using the corresponding name.
   - Check in the directory of the corresponding binary file, using a related name. For example, for app.dll where a file app_x86.pdb is present, also search for file app.pdb.
   - Search in the .debug subdirectory.

4. Search for binary files in these standard Linux OS system directories:
   - /lib/modules (subdirectories are not searched)
   - /lib/modules/`uname -r`/kernel (subdirectories are searched)

5. Search for symbol files in these standard Linux OS system directories:
   - usr/lib/debug (subdirectories are not searched)
   - /usr/lib/debug with appended path to the corresponding binary file, such as /usr/lib/ debug/usr/bin/ls.debug

**Source Search Locations**

A limited set of default source locations are used in addition to the locations specified in the **Source Search** tab. With Intel Advisor, you can use this tab to indicate whether or not the default source locations (listed below) will be searched.

**NOTE**
When using the Intel Advisor GUI, you must specify one or more new rows (locations) in the **Source Search** tab so Intel Advisor tools can find your application’s annotations.

The following list describes the order and default locations that are searched. As indicated below, some directory searches examine the specified directory and its subdirectories, while other searches do not examine its subdirectories.

1. Search for source files in the directories specified in the **Project Properties, Source Search** tab. With Intel Advisor, you can indicate whether the subdirectories of these directories should be searched.

2. Search for source files in directories from the collected result that provide an absolute path name.
3. When using an integrated Visual Studio project, the source directories provided by the Visual Studio project pre-populate the corresponding source directories in the internal representation of the Source Search tab. So, the Visual Studio project's source directories are searched for source files, and they apply to all configurations. When using Microsoft Visual Studio*: Search for source files in Visual Studio project directories.

4. On Linux OS systems: Search for source files in these standard Linux locations (does not search subdirectories):

   /usr/src
   /usr/src/linux-headers-`uname -r`

See Also
Binary/Symbol Search Tab
Source Search Tab

Options Dialog Box
Dialog Box Purpose and Usage | Dialog Box Purpose and Usage | Dialog Box Purpose and Usage

Options Dialog Box Purpose and Usage
The Options dialog box consists of the following tabs:

- **Editor tab** (Linux* OS only) - Specify the editor in which the Intel Advisor displays source files when you double-click a line in an Intel Advisor source region.
- **General tab** - Configure default behavior, enable/disable warning messages, and set modeling assumptions, result locations, and assembly code style.
- **Result Location tab** - Specify the storage directory for future Intel Advisor result files.
- **Assembly tab** - Specify the way the Intel Advisor displays application assembly code.

Options Dialog Box Access
To access this dialog box:

- From the Intel Advisor GUI, choose File > Options...
- From the Visual Studio* menu (Windows* OS only), choose Tools > Options. In the Options dialog box, expand the Intel Advisor program folder.

Options Dialog Box Controls

<table>
<thead>
<tr>
<th>Tab Selector</th>
<th>Select among the Options tabs:</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>• <strong>General tab</strong></td>
</tr>
<tr>
<td></td>
<td>• <strong>Result Location tab</strong></td>
</tr>
<tr>
<td></td>
<td>• <strong>Editor tab</strong> (Linux* OS only)</td>
</tr>
<tr>
<td></td>
<td>• <strong>Assembly tab</strong></td>
</tr>
</tbody>
</table>

| Tab Options | Specify the properties and fields you need to run analyses. |

Editor Tab
Tab Purpose and Usage | Tab Location | Tab Controls
Editor Tab Purpose and Usage

NOTE
The Editor tab is available on Linux* OS only.

Use this tab in the Options dialog box to specify the editor in which the Intel Advisor displays source files when you double-click a line in an Intel Advisor source region.

Editor Tab Location
One of the tabs in the Options dialog box, which also includes:

- Assembly tab
- General tab
- Result Location tab

To access this tab: From the Intel Advisor GUI, choose File > Options > Editor.

Editor Tab Controls

<table>
<thead>
<tr>
<th>Use This</th>
<th>To Do This</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. External editor for this language:</td>
<td>Select the language for which you will choose an editor: C/C++, Fortran, or</td>
</tr>
<tr>
<td>drop-down menu</td>
<td>Other.</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>
General Tab
Tab Purpose and Usage | Tab Location | Tab Controls

General Tab Purpose and Usage
Use this tab in the Options dialog box to configure default behavior, enable/disable warning messages, and set modeling assumptions, result locations, and assembly code style.

General Tab Location
One of the tabs in the Options dialog box, which also includes:
- Assembly tab
- Result Location tab
- Editor tab (Linux* OS only)

To access this tab:
- From the Intel Advisor GUI, choose File > Options > General.
- From the Visual Studio* menu, choose Tools > Options.... In the Options dialog box, expand the Intel Advisor program folder and choose the General page.

General Tab Controls

<table>
<thead>
<tr>
<th>Use This</th>
<th>To Do This</th>
</tr>
</thead>
<tbody>
<tr>
<td>When displaying a window, show its explanation tip checkbox</td>
<td>Determine whether a help snippet explanation for the current window appears when the Result is opened.</td>
</tr>
<tr>
<td>Show the Advisor Workflow tab when a new collection starts checkbox</td>
<td>Control whether the Advisor Workflow tab automatically opens when you run a tool analysis.</td>
</tr>
</tbody>
</table>
| Show build settings warning before a new collection starts checkbox | Control whether a message appears when the build settings do match the suggested settings for the selected analysis:  
  - **Analysis using the Debug build settings** message appears when you run Survey or Suitability tool analysis with Debug build options.  
  - **Analysis using the Release build settings** message appears when you run Dependencies tool analysis with Release build options. |
<table>
<thead>
<tr>
<th>Use This</th>
<th>To Do This</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Show missing debug information warning</strong></td>
<td>Control whether a message appears near the top of the Survey Report window when the target executable does not contain debug information after running the Survey tool.</td>
</tr>
<tr>
<td>checkbox</td>
<td></td>
</tr>
<tr>
<td><strong>Show incorrect compiler options warning</strong></td>
<td>Control whether a message appears near the top of the Survey Report window when the current compiler options are incorrect.</td>
</tr>
<tr>
<td>checkbox</td>
<td></td>
</tr>
<tr>
<td><strong>Show incorrect compiler version warning</strong></td>
<td>Control whether a message appears near the top of the Survey Report window when a higher compiler version should be installed.</td>
</tr>
<tr>
<td>checkbox</td>
<td></td>
</tr>
<tr>
<td><strong>Show higher ISA available warning</strong></td>
<td>Control whether a message appears near the top of the Survey Report window when a higher Instruction Set Architecture should be used.</td>
</tr>
<tr>
<td>checkbox</td>
<td></td>
</tr>
<tr>
<td><strong>Show inline debug information warning</strong></td>
<td>Control whether a message appears near the top of the Survey Report window when debug information is in the code.</td>
</tr>
<tr>
<td>checkbox</td>
<td></td>
</tr>
<tr>
<td><strong>Modeling Assumptions</strong> drop-down lists</td>
<td>Set the default values for Modeling Assumptions that appear in the Suitability Report window, such as the scalability graph.</td>
</tr>
<tr>
<td>• <strong>Maximum CPU Count</strong> - specify the maximum number of CPUs to model.</td>
<td></td>
</tr>
<tr>
<td>• This value limits the size you can set for the <strong>CPU Count</strong>; it also sets the size of the scalability graph <strong>CPU Count</strong> (X axis). You can set values using power-of-two integers 2, 4, 8, ... up to 8192.</td>
<td></td>
</tr>
<tr>
<td>• <strong>CPU Count</strong> - specify the number of CPUs to model for the target system(s).</td>
<td></td>
</tr>
<tr>
<td>• Set a value using power-of-two integers from 2 up to the chosen <strong>Maximum CPU Count</strong>. This value sets the default for the Suitability Report window.</td>
<td></td>
</tr>
<tr>
<td>• <strong>Threading Mode</strong> - choose either Intel® TBB, OpenMP*, Microsoft TPL*, or Other.</td>
<td></td>
</tr>
<tr>
<td><strong>Application output destination</strong> radio button</td>
<td>On Windows* OS systems, control whether console output for a program is displayed in the:</td>
</tr>
<tr>
<td>• Separate console window.</td>
<td></td>
</tr>
<tr>
<td>• Microsoft Visual Studio <strong>Output</strong> window.</td>
<td></td>
</tr>
<tr>
<td>• <strong>Application Output</strong> window (in the Intel Advisor result tab). This also enables application output to be viewed after collection by clicking a link in the <strong>Summary</strong> window to display the <strong>Application Output</strong> window.</td>
<td></td>
</tr>
<tr>
<td>The next time you run a tool analysis, the application output appears in the selected output destination.</td>
<td></td>
</tr>
<tr>
<td>On Linux* OS systems, control whether the console output from the target is displayed in the Intel Advisor <strong>Application Output</strong> window (in the Result tab) or in a separate command terminal window.</td>
<td></td>
</tr>
<tr>
<td><strong>NOTE</strong></td>
<td>If you must interact with the application during execution, choose the separate console window option or use stdin redirection on the command line.</td>
</tr>
</tbody>
</table>
Result Location Tab
Tab Purpose and Usage | Tab Location | Tab Controls

Result Location Tab Purpose and Usage
Use this tab in the Options dialog box to specify the storage directory for future Intel Advisor result files.

Result Location Tab Location
One of the tabs in the Options dialog box, which also includes:

- General tab
- Assembly tab
- Editor tab (Linux* OS only)

To access this tab:

- From the Intel Advisor GUI, choose File > Options > Result Location.
- From the Visual Studio* menu, choose Tools > Options.... In the Options dialog box, expand the Intel Advisor program folder and choose the Result Location page.

Result Location Tab Controls

<table>
<thead>
<tr>
<th>Use This</th>
<th>To Do This</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Result location</strong> radio button</td>
<td>Determine whether new result files are saved in a subdirectory within each Microsoft Visual Studio* or Intel Advisor GUI project’s directory, or in a custom, central location that you specify. If you select the <strong>Save all results in this directory</strong> option, either type the path or click the <strong>Browse</strong> button to navigate to the desired custom directory location. The subdirectory name is the result name, such as e000). When done, click <strong>OK</strong>.</td>
</tr>
</tbody>
</table>

Assembly Tab
Tab Purpose and Usage | Tab Access | Tab Controls

Assembly Tab Purpose and Usage
Use this tab in the Options dialog box to specify the way the Intel Advisor displays application assembly code.

Assembly Tab Access
One of the tabs in the Options dialog box, which also includes:

- General tab
- Result Location tab
- Editor tab (Linux* OS only)

To access this tab:

- From the Intel Advisor GUI, choose File > Options... > Assembly
• For Windows* OS: From the Visual Studio* menu, choose **Tools > Options...** In the **Options** dialog box, expand the **Intel Advisor** program folder and choose the **Assembly** page.

**Assembly Tab Controls**

<table>
<thead>
<tr>
<th>Use This</th>
<th>To Do This</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Default syntax (MASM style for Windows, GAS style for Linux) radio button</strong></td>
<td>Enable the default assembly representation style.</td>
</tr>
<tr>
<td><strong>GAS style syntax radio button</strong></td>
<td>Strictly use GAS assembly representation style.</td>
</tr>
<tr>
<td><strong>MASM style syntax radio button</strong></td>
<td>Strictly use MASM assembly representation style.</td>
</tr>
</tbody>
</table>

**Choose External Editor Dialog Box**

Use this dialog box to specify which external editor to use with Intel Advisor. This dialog box applies to Linux* OS systems only. Alternatively, you can set the project properties for multiple languages using the **Editor** tab on the **Options** dialog box.

To access this dialog box: Double-click a source line while editing a source file.

<table>
<thead>
<tr>
<th>Use This</th>
<th>To Do This</th>
</tr>
</thead>
<tbody>
<tr>
<td>A list of available editors on this system</td>
<td>Select the editor (such as gedit) to be associated with the current language, or choose to allow selection using an environment variable with <strong>Text editor set in EDITOR or VISUAL environment variable</strong>. Repeat to associate an editor with each language you use. After selecting the editor to be associated with the current language, click <strong>OK</strong>.</td>
</tr>
</tbody>
</table>

**Target Application Configuration**

*Configure your application for analysis.*

Before you start profiling your applications and applying changes that should increase performance, you need to configure the applications. Refer to the following topics:

- **Build Settings for Supported Languages** topic - To understand supported programming languages, and required configuration.
- **Limiting the Number of Threads Used by Parallel Frameworks** topic - To understand required configuration for application threading.
- **Choosing a Small, Representative Data Set** topic - To understand how you can reduce analysis overheads by reducing the amount of analyzed data.

**Build Settings for Supported Languages**

Intel Advisor supports three major languages:

- **C/C++** for Windows and Linux systems
- **C#** for Windows systems only
- **Fortran** for Windows and Linux systems

The option names for the build settings vary by language and operating system. Building native code targets require both compiler and linking options.
Tip

• For the most current information on optimal C/C++ and Fortran build settings, see User Guide: Before You Begin.
• In each C# module that contains Intel Advisor annotations, you must reference the annotation definitions using the Visual Studio IDE:

  1. Choose Properties > Add > Existing Item.
  2. Browse to the Intel Advisor installation directory and select ...

If you need to build an assembly, you must explicitly reference the ...

Limiting the Number of Threads Used by Parallel Frameworks

Intel® Advisor tools are designed to collect data and analyze serial programs. Before you use the Intel Advisor Suitability and Dependencies tools to examine a partially parallel program, modify your program so it runs as a serial program with a single thread within each parallel site.

Run Your Program as a Serial Program

To run the current version of your program as a serial program, you need to limit the number of threads to 1. To run your program with a single thread:

• With Intel® Threading Building Blocks (Intel® TBB), in the main thread create a

  tbb::task_scheduler_init init(1); object for the lifetime of the program and run the executable again. For example:

  ```c
  int main() {
    tbb::task_scheduler_init init(1);
    // ...rest of program...

    return 0;
  }
  ```

  The effect of `task_scheduler_init` applies separately to each user-created thread. So if the program creates threads elsewhere, you need to create a `tbb::task_scheduler_init init(1);` for that thread’s lifetime as well. Use of certain Intel TBB features can prevent the program from running serially. For more information, see the Intel TBB documentation.

• With OpenMP*, do one of the following:
  
  • Set the OpenMP* environment variable `OMP_NUM_THREADS` to 1 before you run the program.
  • Omit the compiler option that enables recognition of OpenMP pragmas and directives. On Windows* OS, omit `/Qopenmp`, and on Linux* OS omit `-openmp`.

  For more information, see your compiler documentation.

If you cannot remove the parallelism, you should add annotations to mark the parallel code regions and learn how parallel code will impact Intel Advisor tool reports.

See Also

Using Partially Parallel Programs with Intel Advisor Tools

Choosing a Small, Representative Data Set

When you run an analysis, the Intel® Advisor executes the target against the supplied data set. Data set size and workload have a direct impact on application execution time and analysis speed.
For example, it takes longer to process a 1000x1000 pixel image than a 100x100 pixel image. A possible reason: You may have loops with an iteration space of 1...1000 for the larger image, but only 1...100 for the smaller image. The exact same code paths may be executed in both cases. The difference is the number of times these code paths are repeated.

You can control analysis cost without sacrificing completeness by minimizing this kind of unnecessary repetition from your target's execution.

Instead of choosing large, repetitive data sets, choose small, representative data sets that fully create tasks with minimal to moderate work per task. *Minimal* to *moderate* means just enough work to demonstrate all the different behaviors a task can perform.

Your objective: In as short a runtime period as possible, execute as many paths and the maximum number of tasks (parallel activities) as you can afford, while minimizing the repetitive computation within each task to the bare minimum needed for good code coverage.

Data sets that run in about ten seconds or less are ideal. You can always create additional data sets to ensure all your code is checked.

**On Windows* OS only:** To modify the input data set in the Visual Studio IDE, do one of the following:

- Specify Properties for the project or configuration. For example, right-click the startup project's name to display the context menu:
  1. Choose **Properties > Configuration properties > Debugging**.
  2. Select the type of configuration this change will apply to by selecting the type under **Configuration**, such as **Active(Debug)**, **Debug**, **Release**, or **All Configurations**. By default, properties for **Debug** and **Release** configuration are maintained separately.
  3. Edit the **Command Arguments** to select the appropriate data set.
  4. Click **OK**.
- Specify a different startup project that already has a reduced data set.
- Modify the program's sources (perhaps using `#ifdef` directives) and rebuild the target.

To modify the input data set using the Intel Advisor GUI, do one of the following:

- Specify the project properties for the target. For example:
  1. Either click **File > Project properties...** or the icon on the Intel Advisor toolbar. This displays the **Project Properties** dialog box.
  2. If needed, click the **Analysis Target** tab.
  3. In the **Target type** drop-down list, choose **Dependencies Analysis**.
  4. In the **Application parameters**, if your target's main entry point accepts command-line arguments, specify a value in this field. Either type a value, or click the **Modify...** button.
  5. When done, click **OK**.
- Modify the program's sources (perhaps using `#ifdef` directives) and rebuild the target.

**Tip**

- **On Windows* OS only:** If you run this configuration often, consider creating a new configuration perhaps called Dependencies for this small data set.
- For the most current information on optimal C/C++ and Fortran build settings, see Before You Begin.

### Workflows

**Purpose and Usage | Report Regions**

**Workflows Purpose and Usage**

Intel Advisor suggests but does not limit you to following suggested Threading and Vectorization workflows:

- **Vectorization Workflow Diagram** - represents the workflow that you might want to follow to achieve the best performance using vectorization.
• **Threading Workflow Diagram** - represents the workflow that you might want to follow to achieve the best performance using threading parallelism.

Intel® Advisor Beta introduces a new *Offload workflow* that you might want to follow to achieve best performance using performance modeling on Intel® Advisor Beta profiles. For more information, see the *Offload Advisor* section.

### Workflow Tabs Regions

Select between workflow tabs. The following are available:

- Vectorization Workflow tab
- Threading Workflow tab

**Batch mode** toggle, which enables you to run several analyses in a row according to the specified criteria.

Workflow steps that include different analysis types and report options.
Vectorization Workflow Diagram

Follow these steps (white blocks are optional) to get started using the Vectorization Advisor in the Intel Advisor.

1. **Survey analysis** - Offers integrated compiler report data and performance data all in one place. Use it to help identify:
   - Where vectorization, or parallelization with threads, will pay off the most
   - If vectorized loops are providing benefit, and if not, why not
   - Un-vectorized loops and why they are not vectorized
   - Performance problems in general

   The **Survey Analysis** also provides code-specific recommendations for how to fix vectorization issues, and quick visibility into source code and assembly code.

2. **Trip Counts and FLOP analysis** (optional) - Dynamically identifies the number of times loops are invoked and execute (sometimes called call count/loop count and iteration count respectively); and measures the number of floating-point and integer operations, and memory traffic. Use to make better decisions about your vectorization strategy for particular loops, as well as optimize already-parallel loops.

3. **Roofline analysis** (optional) - Helps visualize actual performance against hardware-imposed performance ceilings, as well as determine the main limiting factor (memory bandwidth or compute capacity), thereby providing an ideal roadmap of potential optimization steps.

   Use the **Roofline** chart to answer the following questions:
   - What is the maximum achievable performance with your current hardware resources?
   - Does your application work optimally on current hardware resources?
   - If not, what are the best candidates for optimization?
   - Is memory bandwidth or compute capacity limiting performance for each optimization candidate?

4. **Dependencies analysis** (optional) - For safety purposes, the compiler is often conservative when assuming data dependencies. Use a Dependencies-focused **Refinement Report** to check for real data dependencies in loops the compiler did not vectorize because of assumed dependencies. If real dependencies are detected, the analysis can provide additional details to help resolve the dependencies.

   Your objective: Identify and better characterize real data dependencies that could make forced vectorization unsafe.
• **Memory Access Patterns (MAP) analysis** (optional) - Use a MAP-focused Refinement Report to check for various memory issues, such as non-contiguous memory accesses and unit stride vs. non-unit stride accesses. Your objective: Eliminate issues that could lead to significant vector code execution slowdown or block automatic vectorization by the compiler.

**Vectorization Workflow Tab**

**Tab Purpose and Usage | Tab Location | Tab Controls**

**Workflow Tab Purpose and Usage**

Use the Intel Advisor workflow tab to start analysis.

See [Getting Started with Intel\textsuperscript{\textregistered} Advisor](#) for a more information about Intel Advisor typical usage scenarios (workflows).

**Workflow Tab Location**

Left side of Intel Advisor GUI

**Workflow Tab Controls**

<table>
<thead>
<tr>
<th>To Do This</th>
<th>Use This</th>
</tr>
</thead>
<tbody>
<tr>
<td>Learn about each workflow step as you progress through the workflow.</td>
<td>Hover over the question mark icon to see the tooltip on how to use the Advisor Workflow tab components.</td>
</tr>
<tr>
<td><strong>Run Roofline</strong> - Consequently run the Survey and FLOP analyses to build the Roofline chart.</td>
<td>Below Run Roofline, click the Collect button.</td>
</tr>
<tr>
<td>To Do This</td>
<td>Use This</td>
</tr>
<tr>
<td>------------</td>
<td>----------</td>
</tr>
<tr>
<td><strong>1. Survey Target</strong> - Run the Survey tool <img src="logo.png" alt="icon" /> to profile your target program to locate hotspots.</td>
<td>Below <strong>Survey Target</strong> click the <strong>Collect</strong> button or <img src="stop.png" alt="icon" />. Once you start the analysis, pause <img src="pause.png" alt="icon" />, stop <img src="stop.png" alt="icon" />, and cancel <img src="cancel.png" alt="icon" /> buttons appear.</td>
</tr>
<tr>
<td><strong>1.1 Find Trip Counts</strong> - Run the Trip Counts tool <img src="logo.png" alt="icon" /> to collect loop iteration statistics.</td>
<td>Below <strong>Find Trip Counts</strong> click the <strong>Collect</strong> button. Once you start the analysis, stop <img src="stop.png" alt="icon" /> and cancel <img src="cancel.png" alt="icon" /> buttons appear.</td>
</tr>
<tr>
<td><strong>2. Check Memory Access Patterns</strong> tool <img src="logo.png" alt="icon" /> with your application (target) to collect data on memory access strides.</td>
<td>In the <strong>Vectorization Workflow</strong> below <strong>Check Memory Access Patterns</strong> click the <strong>Collect</strong> button. Once you start the analysis, stop <img src="stop.png" alt="icon" /> and cancel <img src="cancel.png" alt="icon" /> buttons appear.</td>
</tr>
<tr>
<td><strong>Mark Loops for Deeper Analysis (Vectorization Workflow only)</strong> - Select loops <img src="logo.png" alt="icon" /> for further analysis via Dependencies and Memory Access Patterns tools.</td>
<td>In the <strong>Result Tab</strong> switch to the <strong>Survey Report</strong> and in the <img src="check.png" alt="icon" /> column check-mark the <img src="check.png" alt="icon" /> loops that you want to analyze with the Dependencies and Memory Access Patterns tools as part of your Vectorization Workflow.</td>
</tr>
<tr>
<td><strong>2.2 Check Dependencies</strong> - Run the Dependencies tool <img src="logo.png" alt="icon" /> to predict likely data sharing problems before you add parallelism to your program. This tool uses your inserted annotations to predict data sharing problems.</td>
<td>Below <strong>2.2 Check Dependencies</strong> in Vectorization Workflow tab, click the <strong>Collect</strong> button. Once you start the analysis, stop <img src="stop.png" alt="icon" /> and cancel <img src="cancel.png" alt="icon" /> buttons appear.</td>
</tr>
</tbody>
</table>

**Threading Workflow Diagram**

*Follow these steps (white blocks are optional) to get started using the Threading Advisor in the Intel Advisor.*

- Click the ![icon](Threading Workflow.png) button.
- Click the ![icon](Vectorization Workflow.png) button.
- Use the vertical dot ![icon](vertical.png) button at the edge of the tab.
- **Survey analysis** - Shows the loops and functions where your application spends the most time. Use this information to discover candidates for parallelization with threads.

- **Trip Counts and FLOP** analysis (optional) - Shows the minimum, maximum, and median number of times a loop body will execute, as well as the number of times a loop is invoked. Use this information to make better decisions about your threading strategy for particular loops.

- **Roofline analysis** (optional) - Applies to both Vectorization Advisor and Threading Advisor, but is most useful in Vectorization Advisor.

- **Annotations** - Insert to mark places in your application that are good candidates for later replacement with parallel framework code that enables threading parallel execution. Annotations are subroutine calls or macros (depending on the programming language) that can be processed by your current compiler but do not change the computations of your application.

- **Suitability analysis** - Predicts the maximum speed-up of your application based on the inserted annotations and a variety of what-if modeling parameters with which you can experiment. Use this information to choose the best candidates for parallelization with threads.

- **Dependencies analysis** - Predicts parallel data sharing problems based on the inserted annotations. Use this information to fix the data sharing problems if the predicted maximum speed-up benefit justifies the effort.

**Threading Workflow Tab**

**Tab Purpose and Usage | Tab Location | Tab Controls**

**Workflow Tab Purpose and Usage**

Use the Intel Advisor workflow tab to start analysis.

See [Getting Started with Intel® Advisor](#) for a more information about Intel Advisor typical usage scenarios (workflows).

**Workflow Tab Location**

Left side of Intel Advisor GUI
Workflow Tab Controls

Learning about each workflow step as you progress through the workflow.

1. **Survey Target** - Run the Survey tool to profile your target program to locate hotspots.

   1.1 **Find Trip Counts** - Run the Trip Counts tool to collect loop iteration statistics.

2. **Annotate Sources** - Add annotations into your sources to mark possible parallel regions.

**To Do This**

<table>
<thead>
<tr>
<th>To Do This</th>
<th>Use This</th>
</tr>
</thead>
<tbody>
<tr>
<td>Learn about each workflow step as you progress through the workflow.</td>
<td>Hover over the question mark icon to see the tooltip on how to use the Advisor Workflow tab components.</td>
</tr>
<tr>
<td><strong>1. Survey Target</strong> - Run the Survey tool to profile your target program to locate hotspots.</td>
<td>Below Survey Target click the Collect button or . Once you start the analysis, pause , stop , and cancel buttons appear.</td>
</tr>
<tr>
<td><strong>1.1 Find Trip Counts</strong> - Run the Trip Counts tool to collect loop iteration statistics.</td>
<td>Below Find Trip Counts click the Collect button. Once you start the analysis, stop and cancel buttons appear.</td>
</tr>
<tr>
<td><strong>2. Annotate Sources</strong> - Add annotations into your sources to mark possible parallel regions.</td>
<td>View instructions under 2. Annotate Sources and click the button to view the steps. On Windows* OS, you can use the Visual Studio code editor and the Intel Advisor Annotation Wizard to add annotations into your sources. Use the Survey Report and its Survey Source window to help you locate the code regions where you will add annotations. To help you add annotations and build settings, use the annotation assistant in the Survey windows.</td>
</tr>
</tbody>
</table>
To Do This | Use This
---|---
**3. Check Suitability** - Run the Suitability tool to predict approximate performance improvements by sampling your program. This tool uses your inserted annotations to predict site and task performance characteristics. **In the Threading Workflow tab below Check Suitability** click the button or . Once you start the analysis, pause , stop , and cancel buttons appear.

Below **4. Check Dependencies** in the Threading Workflow tab, click the button. Once you start the analysis, stop and cancel buttons appear.

Switch to the Threading Workflow Tab while the Vectorization Workflow Tab is active.

Switch to the Vectorization Workflow Tab while the Threading Workflow Tab is active.

Hide and unhide the workflow tab

### Survey, Trip Counts, FLOPS, and Roofline Analyses

#### Purpose and Usage | Report Regions

**Survey Report Purpose and Usage**

Run a Survey analysis to generate a Survey Report that offers integrated compiler report data and performance data for your target application all in one place. Optionally run a Trip Counts analysis and/or FLOP analysis to add data to the Survey Report. The Roofline analysis runs a Survey analysis followed by a FLOP analysis automatically.

- **Survey analysis** - Identifies:
  - Where vectorization, or parallelization with threads, will pay off the most
  - If vectorized loops are providing benefit, and if not, why not
  - Un-vectorized loops and why they are not vectorized
  - Performance problems in general

- **Trip Counts analysis** - Dynamically identifies the number of times loops and functions are invoked and executed (also called call count/loop count and iteration count respectively). Use Trip Counts data to:
  - Detect loops with too-small trip counts and trip counts that are not a multiple of vector length.
  - Analyze parallelism granularity more deeply.

- **FLOP analysis** - Dynamically measures floating-point and integer operations, and memory traffic. Use the FLOP analysis to generate application memory usage and performance values that help you make better decisions about your vectorization strategy.
• **Roofline analysis** - Helps you visualize actual performance against hardware-imposed performance ceilings, as well as determine the main limiting factor (memory bandwidth or compute capacity), thereby providing an ideal roadmap of potential optimization steps.

Use the **Roofline** chart to answer the following questions:

- What is the maximum achievable performance with your current hardware resources?
- Does your application work optimally on current hardware resources?
- If not, what are the best candidates for optimization?
- Is memory bandwidth or compute capacity limiting performance for each optimization candidate?

**Survey Report Regions**

- **Filters** pane - Filter analysis data by a variety of criteria, such as module, loop/function, and vectorized/non-vectorized.
- **Roofline Chart** pane - visualize actual performance against hardware-imposed performance ceilings, as well as determine the main limiting factor (memory bandwidth or compute capacity), thereby providing an ideal roadmap of potential optimization steps.
- **Loop Information** pane - View integrated compiler report data and Intel Advisor performance data for target application loops, and mark a loop for deeper analysis.
- **Advanced View** pane - View more information for a loop selected in the **Loop Information** pane.
  - **Source** tab - View source code for a selected loop.
  - **Top Down** tab - View the function/loop hierarchy in a stack, and the source code associated with a specific function or loop. Each function or loop appears on a separate grid line. Loops are identified with an icon, the word [loop], followed by the source location and the function or procedure name that executes it.
  - **Code Analytics** tab - View the most important statistics for a selected loop.
  - **Assembly** tab - View assembly representation for a selected loop.
  - **Recommendations** tab - Explore code-specific recommendations for how to fix vectorization issues (Vectorization Advisor only).
  - **Why No Vectorization?** tab - View the reason automated vectorization failed (Vectorization Advisor only).

The associated **Survey Source** window, which you can use to view details about a code region, has the following panes:

- **View Activation** pane - Enable or disable views shown in the Source view
- **Source View** pane - View user-visible source code representation of the selected site.
- **Assembly View** pane - View assembly representation of the selected site.
- **Call Stack View** pane - View the call stack for the selected code region. Click to display related code regions in the **File: filename** pane, or click to display the context menu.

Do one of the following to access the **Survey Source** window:

- Double-click a row in the **Loop Information** pane or in the **Top Down** tab of the **Advanced View** pane.
- Right-click a loop and select **View Source**.

**Survey Analysis**

**Purpose and Usage | Prerequisites | Run | Controls | Data Reference | What Do I Do Next?**

**Survey Analysis Purpose and Usage**

Run a Survey analysis to generate a **Survey Report** that identifies:

- Where vectorization, or parallelization with threads, will pay off the most
- If vectorized loops are providing benefit, and if not, why not
- Un-vectorized loops and why they are not vectorized
• Performance problems in general

**Survey Analysis Prerequisites**

To prepare to run a Survey analysis:

1. Do one of the following to open the Analysis Target tab in the Project Properties dialog box:
   - In the Intel Advisor GUI, choose **Project > Project Properties**…
   - In the Visual Studio IDE, choose **Project > Intel Advisor [version] > Project Properties**…

2. Choose the **Survey Hotspots Analysis** type, then choose the input data set or runtime parameters for your target application. If you want to speed up your target application generally, you can use typical input data. However, if you want to improve performance on specific, slow operations, choose input data to emphasize these slow operations. After you prepare appropriate input data, configure your target application with the chosen data.

**Tip**
- If you plan to run Refinement Reports, set parameters at the same time you set parameters for the **Survey Hotspots Analysis** and **Survey Trip Count Analysis** types. If possible, use the **Inherit settings from Survey Hotspots Analysis Type** checkbox for other Analysis Types.
- Setting search directories in the Binary/Symbol Search tab and Source Search tabs is optional for the Vectorization Advisor.
- Choose the optimal compiler settings. The most up-to-date settings are in *User Guide: Before You Begin*.

**Run a Survey Analysis**

To run a Survey analysis, do one of the following:

- In the Vectorization Workflow tab, click the **Collect** button under **Survey Target**.
- In the Visual Studio Solution Explorer, right-click the project, then choose **Intel Advisor [version] > Start Survey Analysis**.

While analysis is running, you can do the following in the Workflow tab:

- Stop analysis and data collection, and retain the already collected data: Click the **Stop** button.
- Cancel analysis and data collection, and discard the collected data: Click the **Cancel** button.
- Interrupt post-collection data finalization: Click the **Cancel** button.

**Tip**
- If the Survey analysis does not collect sufficient data to produce a **Survey Report**, it displays a **Target executed too quickly or does not contain debug symbols message**. Consider increasing the target workload or data to run the analysis at least a few seconds; or check whether debug information is specified as a build option; or specify a different target application.
- There are a variety of techniques available to minimize data collection, result size, and execution time. Check **Minimizing Analysis Overhead**.
Survey Report Controls

There are many controls available to help you focus on the data most important to you, including the following:

1. Click the control to save a read-only result snapshot you can view any time.
   Intel Advisor stores only the most recent analysis result. Visually comparing one or more snapshots to each other or to the most recent analysis result can be an effective way to judge performance improvement progress.

To open a snapshot, choose File > Open > Result...

2. Click the various Filter controls to temporarily limit displayed data based on your criteria.

3. Click the control to view loops in non-executed code paths for various instruction set architectures (ISAs). Prerequisites:
   - Compile the target application for multiple code paths using the Intel compiler.
   - Enable the Analyze loops in not executed code path checkbox in Project Properties > Analysis Target > Survey Hotspots Analysis.

4. This toggle control currently combines two features: The View Configurator and the Smart Mode filter.
   - View Configurator - Toggle on the Customize View control to choose the view layout to display: Default, Smart Mode, or a customized view layout. To create a customized view layout you can apply to this and other projects:
     1. Click the Settings control next to the View Layout drop-down list to open the Configure Columns dialog box.
     2. Choose an existing view layout in the Configuration drop-down list.
     3. Enable/disable columns to show/hide.

     Outcome: Copy n is added to the name of the selected view layout in the Configuration drop-down list.

     4. Click the Rename button and supply an appropriate name for the customized view layout.

     5. Click OK to save the customized view layout.
• **Smart Mode Filter** - Toggle on the **Customize View** control to temporarily limit displayed data to the top potential candidates for optimization based on **Total CPU Time** (the time your application spends actively executing a function/loop and its callees). In the **Top** drop-down list, choose one of the following:
  - The **Number** of top loops/functions to display
  - The **Percent** of **Total CPU Time** the displayed loops/functions must equal or exceed

5. Click the button to search for specific data.
6. Click the tab to open various Intel Advisor reports or views.
7. Right-click a column header to:
   - Hide the associated report column.
   - Resume showing all available report columns.
   - Open the **Configure Columns** dialog box (see #4 for more information).
8. Click the toggle to show all available columns in a column set, and resume showing a limited number of preset columns in a column set.
9. Click the control to:
   - Show options for customizing data in a column or column set.
   - Open the **Configure Columns** dialog box (see #4 for more information).

For example, click the control in the **Compute Performance** column set to:
- Show data for floating-point operations only, for integer operations only, or for the sum of floating-point and integer operations.
- Determine what is counted as an integer operation in integer calculations:
  - Choose **Show Pure Compute Integer Operations** to count only ADD, MUL, IDIV, and SUB operations.
  - Choose **Show All Operations Processing Integer Data** to count ADD, ADC, SUB, MUL, IMUL, DIV, IDIV, INC/DEC, shift, and rotate operations.
10. Click the control to show/hide a chart that helps you visualize actual performance against hardware-imposed performance ceilings, as well as determine the main limiting factor (memory bandwidth or compute capacity), thereby providing an ideal roadmap of potential optimization steps.
11. Click a data row in the top of the **Survey Report** to display more data specific to that row in the bottom of the **Survey Report**. Double-click a loop data row to display a **Survey Source** window. To more easily identify data rows of interest:
  - ⚪️ = Vectorized function
  - ⚫️ = Vectorized loop
  - ☄️ = Scalar function
  - ☁️ = Scalar loop
12. Click a checkbox to mark a loop for deeper analysis.
13. If present, click the image to display code-specific how-can-I-fix-this-issue? information in the **Recommendations** pane.
14. If present, click the image to view the reason automatic compiler vectorization failed in the **Why No Vectorization?** pane.
Next Steps: After Running Survey Analysis

After you run a Survey analysis:

1. Sort by the Self-Time and/or Total-Time column to find top time-consuming loops.

2. Check whether your target loop or function is vector or scalar. Intel Advisor helps you to differentiate vector and scalar via the following icons:
   - ![](VectorizedIcon.png) - vectorized function
   - ![](VectorizedLoopIcon.png) - vectorized loop
   - ![](ScalarFunctionIcon.png) - scalar function
   - ![](ScalarLoopIcon.png) - scalar loop

3. Use filters to hide the code sides that you don’t want to tweak now: ![](VectorizedIcon.png) and ![](NotVectorizedIcon.png)

4. Decide what loops or functions to investigate:
   - If loop/function is scalar
   - If loop/function is vectorized

If Loop/Function is Scalar

If the target loop/function is scalar (![](ScalarFunctionIcon.png) or ![](ScalarLoopIcon.png)), you need to understand why the compiler did not vectorize the loop/function.

Several reasons are possible:

<table>
<thead>
<tr>
<th>Possible Reason</th>
<th>To Confirm</th>
<th>To Do</th>
</tr>
</thead>
<tbody>
<tr>
<td>Assumed dependency</td>
<td>Refer to Why No Vectorization? column. Search for Vector dependence prevents vectorization issue.</td>
<td>Run the Dependencies analysis.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• If no dependencies are found, force vectorization with the omp simd directive or provide other vectorization recommendations to compiler.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• If dependencies are confirmed, resolve them, or move to the next loop.</td>
</tr>
<tr>
<td>Function call in the loop</td>
<td>Refer to Why No Vectorization? column. Search for issues: Function call present</td>
<td>For issue: Function call present, do one of the following:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Inline function into the loop.</td>
</tr>
</tbody>
</table>
### Possible Reason | To Confirm | To Do
--- | --- | ---
Indirect function call present | • Indirect function call present  
Serialized user function call present | • Vectorize the function with the omp declare simd directive.  
For issues *Indirect function call present* or *Serialized user function call present*, refer to guidelines in the **Recommendations** tab.  

**Compiler-assumed inefficient vectorization**  
Refer to **Why No Vectorization?** column. Search for the *Loop vectorization possible but seems inefficient* issue. |  
Try forcing vectorization with the omp simd directive.  
If forcing vectorization doesn’t provide tangible results, consider experimenting with other directives.  
To better understand performance implications and potential speed-up, consider running additional analyses:  
• Trip Counts  
• Memory Access Patterns  

**Other**  
Refer to  
• **Why No Vectorization?** column  
• **Vector Issues** column | Study the Compiler Diagnostic Details and Advisor Recommendations to resolve the issues.  

### If Loop/Function is Vectorized

If the target loop is vectorized (✓ or ✗), ensure vector efficiency is above 90%.

If efficiency is below 90%, consider the following:

| Possible Reason | To Confirm | To Do |
--- | --- | ---
ISA | Refer to **Vectorized Loops/Vector ISA** column to check the ISA version used in the application. | Change the target ISA by specifying corresponding compiler flags.  

**Inefficient peel/remainder**  
Refer to **Vector Issues** column. Search for the *Inefficient Peel/Reminder* issue. Or check if the time spent in peel/reminder is significant. | Resolve the issues:  
• Check **Recommendations** tab.  
• Run the Trip Counts analysis.  

**Possible inefficient memory access**  
Refer to **Vector Issues** column. Search for the *Possible Inefficient Memory Access* issue.  
Refer to **Instruction Set Analysis/Traits** column. Search for the following traits:  
Run the Memory Access Patterns analysis. |
### Possible Reason | To Confirm | To Do
--- | --- | ---
Type conversions present | Refer to Instruction Set Analysis/Traits column. Search for the Type Conversions metric. | Remove redundant type conversions from float to double that might lead to smaller vector length and reduced vectorization efficiency. |
Unaligned vector access in loop | Refer to Advanced/Vectorization Details column. Search for the Unaligned access in vector loop metric. | Align data. |
Register pressure | Refer to Vector Issues column. Search for the Vector register spilling possible issue. | Resolve the issue by doing one of the following:
- Decrease loop unroll factor.
- Split the loop into smaller parts. |
Potential underutilization of FMA instructions | Refer to Vector Issues column. Search for the Potential underutilization of FMA instructions issue. | Resolve the issue by doing one of the following:
- Change the target ISA.
- Explicitly enable FMA generation and vectorization. |
Other | Refer to Vector Issues column. | Follow the Intel Advisor recommendations to resolve the issues. |

**Advanced View Pane (Survey Report)**

Pane Purpose and Usage | Pane Location | Pane Regions and Usage | Pane Controls | Data Reference
--- | --- | --- | --- | ---
Advanced View Pane Purpose and Usage
View more information for a loop selected in the Loop Information pane.

Advanced View Pane Location
Bottom of the Survey Report window

Advanced View Pane Regions and Usage
From left to right:
- **Source** tab - View source code for a selected loop.
- **Top Down** tab - View the function/loop hierarchy in a stack, and the source code associated with a specific function or loop. Each function or loop appears on a separate grid line. Loops are identified with an icon, the word [loop, followed by the source location and the function or procedure name that executes it.
- **Code Analytics** tab - View the most important statistics for a selected loop.
- **Assembly** tab - View assembly representation for a selected loop.
- **Recommendations tab** - Explore code-specific recommendations for how to fix vectorization issues (Vectorization Advisor only).
- **Why No Vectorization? tab** - View the reason automated vectorization failed (Vectorization Advisor only).

**Tip**
The following are available in the `advice` directory in the Intel Advisor installation directory:

- A full list of all recommendations reported by the Vectorization Advisor
- A full list of compiler diagnostic data integrated into the Vectorization Advisor interface
- *Selected Topics from the Intel® C++ Compiler Classic User and Reference Guides*
- *Selected Topics from the Intel® Fortran Compiler Classic User and Reference Guides*

### Advanced View Pane Controls

There are many controls available to help you focus on the data most important to you, including the following:

1. Click the control to save a read-only result snapshot you can view any time.
   
   Intel Advisor stores only the most recent analysis result. Visually comparing one or more snapshots to each other or to the most recent analysis result can be an effective way to judge performance improvement progress.
   
   To open a snapshot, choose **File > Open > Result...**

2. Click the various **Filter** controls to temporarily limit displayed data based on your criteria.

3. Click the control to view loops in non-executed code paths for various instruction set architectures (ISAs). Prerequisites:
   - Compile the target application for multiple code paths using the Intel compiler.
   - Enable the **Analyze loops in not executed code path** checkbox in **Project Properties > Analysis Target > Survey Hotspots Analysis**.

4. This toggle control currently combines two features: The **View Configurator** and the **Smart Mode** filter.
- **View Configurator** - Toggle on the **Customize View** control to choose the view layout to display: **Default**, **Smart Mode**, or a customized view layout. To create a customized view layout you can apply to this and other projects:

1. Click the **Settings** control next to the **View Layout** drop-down list to open the **Configure Columns** dialog box.
2. Choose an existing view layout in the **Configuration** drop-down list.
3. Enable/disable columns to show/hide.

Outcome: Copy n is added to the name of the selected view layout in the **Configuration** drop-down list.

4. Click the **Rename** button and supply an appropriate name for the customized view layout.
5. Click **OK** to save the customized view layout.

- **Smart Mode Filter** - Toggle on the **Customize View** control to temporarily limit displayed data to the top potential candidates for optimization based on **Total CPU Time** (the time your application spends actively executing a function/loop and its callees). In the **Top** drop-down list, choose one of the following:
  - The **Number** of top loops/functions to display
  - The **Percent** of **Total CPU Time** the displayed loops/functions must equal or exceed

5. Click the button to search for specific data.

6. Click the tab to open various Intel Advisor reports or views.

7. Right-click a column header to:
   - Hide the associated report column.
   - Resume showing all available report columns.
   - Open the **Configure Columns** dialog box (see #4 for more information).

8. Click the toggle to show all available columns in a column set, and resume showing a limited number of preset columns in a column set.

9. Click the control to:
   - Show options for customizing data in a column or column set.
   - Open the **Configure Columns** dialog box (see #4 for more information).

For example, click the control in the **Compute Performance** column set to:

- Show data for floating-point operations only, for integer operations only, or for the sum of floating-point and integer operations.
- Determine what is counted as an integer operation in integer calculations:
  - Choose **Show Pure Compute Integer Operations** to count only ADD, MUL, IDIV, and SUB operations.
  - Choose **Show All Operations Processing Integer Data** to count ADD, ADC, SUB, MUL, IMUL, DIV, IDIV, INC/DEC, shift, and rotate operations.

10. Click the control to show/hide a chart that helps you visualize actual performance against hardware-imposed performance ceilings, as well as determine the main limiting factor (memory bandwidth or compute capacity), thereby providing an ideal roadmap of potential optimization steps.
11 Click a data row in the top of the Survey Report to display more data specific to that row in the bottom of the Survey Report. Double-click a loop data row to display a Survey Source window. To more easily identify data rows of interest:

- \( f \) = Vectorized function
- \( \bigcirc \) = Vectorized loop
- \( f \) = Scalar function
- \( \bigcirc \) = Scalar loop

12 Click a checkbox to mark a loop for deeper analysis.

13 If present, click the image to display code-specific how-can-I-fix-this-issue? information in the Recommendations pane.

14 If present, click the image to view the reason automatic compiler vectorization failed in the Why No Vectorization? pane.

15 Click the control to show/hide the Workflow pane.

Top Down Tab (Survey Report)
Tab Purpose and Usage | Tab Location | Tab Regions and Usage | Tab Controls

Top Down Tab Purpose and Usage
View the function/loop hierarchy in a stack, the source code associated with a specific function or loop, and more. Each function or loop appears on a separate grid line. Loops are identified with an icon \( \bigcirc \) and the word loop, followed by the function or procedure name that executes it and the source location.

Top Down Tab Location
Bottom of the Survey Report window

Top Down Tab Regions and Usage
From left to right:

- Function Call Sites and Loops: View a hierarchical listing of functions and loops in your code. You can expand and collapse entries, or double-click the name of a function or loop to view its source code.
- Table Columns: View additional information about functions and loops in the grid, such as CPU time, type (function, scalar, etc.), compute performance statistics, the instruction sets and extensions (such as VNNI) used, and trip counts. See Data Reference for descriptions of the data columns in Survey and Refinement Reports.

Top Down Tab Controls
You can customize the columns shown in the Top Down grid. Click the Customize View button in the upper right of the application to display the View Layout drop-down list and the Settings control (gear icon) in the upper-right of the Top Down tab.

Select a column layout from the View Layout drop-down list to change the columns to match an existing column layout.

You can modify a column layout. Select the Settings control next to the View Layout drop down list to open the Configure Columns dialog box, then:

1. Choose an existing view layout in the Configuration drop-down list.
2. Enable/disable columns to show/hide.
Outcome: A new view layout is added to the Configuration drop-down list, with Copy n added to the name of the original layout.

3. Click the Rename button and supply an appropriate name for the customized view layout.
4. Click OK to save the customized view layout.

You can also right-click the name of a column in the grid to Hide Column, Show All Columns, or Configure Column Layouts. You can choose to display one layout in the main Survey Report grid, and choose another layout for the Top Down tab.

NOTE Hiding or showing columns in a column layout will apply your changes to any grid (the Survey Report grid or the Top Down tab) that is currently using the layout. However, you can rearrange columns in one grid without affecting another grid.

See Also
Advanced View Pane (Survey Report)

Code Analytics Tab
Tab Purpose and Usage | Tab Location | Tab Regions and Usage

Code Analytics Tab Purpose and Usage
View the most important statistics for a selected loop.

Code Analytics Tab Location
Bottom of the Survey Report and Roofline Analysis window

Code Analytics Tab Regions and Usage
- **Summary**: View a quick list of basic information about the loop, such as the code source, whether the loop is scalar or vector, instruction set (and whether extensions, such as VNNI, are used), total time, and the static and dynamic instruction mix.
- **Traits**: View additional scalar and vectorization characteristics that may impact performance. For a list of possible traits, see the Data Reference.
- **Trip Counts**: View information about the number of times the loop is invoked (trip count), such as the minimum and maximum trip count, the average loop iteration time, etc.
- **Statistics for x**: Click the drop-down list at the top of this section to choose to display performance statistics for a specific operation type: FLOP, INTOP, INT + FLOAT, or All Operations. Click the toggle control to switch between displaying performance statistics using self or total loop metrics.
- **Data Transfer and Bandwidth**: View the amount of data transferred for the loop per memory level and bandwidth for each memory level. Click the toggle control to switch between self and total data.

NOTE To see this pane, run Roofline analysis for all memory levels with cache simulation enabled.

- **Code Optimizations**: View a list of code optimizations applied to the loop by the compiler, as well as information on which compiler was used and what version. This information is only available for binaries compiled by the Intel® C Compiler, Intel® C++ Compiler Classic, or Intel® Fortran Compiler Classic.
- **Roofline**: View a more detailed roofline chart that summarizes recommendations and information from the Roofline Conclusions section, such as whether the loop is compute bound, memory bound, or both. This chart features:
  - The labeled distance between the loop and the performance roof limiting it.
• The metrics used to plot the loop on the chart, Giga OPS (operations per second) and AI (arithmetic intensity).

If you have collected the Roofline for all memory levels, you can use the Memory Level/CARM selector to switch between Roofline guidance views. When you set the selector to Memory Level, the chart features X marks representing memory levels for the loop and an arrowed line indicating the memory level that bounds the loop.

• **Memory Metrics**: View how efficiently the loop/function uses cache and what memory level bounds the loop/function.
  
  • Review the **Impacts** histogram, which shows time spent processing requests for each memory level. A big value indicates a memory level that bounds the selected loop.
  
  • Review the **Shares** histogram, which shows an amount of data that passes through each memory level.

**See Also**

Advanced View Pane (Survey Report)

**Recommendations Tab**

**Tab Purpose and Usage** | **Tab Location** | **Tab Regions and Usage**

**Recommendations Tab Purpose and Usage**

Explore code-specific recommendations for how to fix vectorization issues (Vectorization Advisor only).

**Recommendations Tab Location**

Bottom of the *Survey Report* and *Roofline Analysis* window

**Recommendations Tab Regions and Usage**

Select a loop in the Survey Grid or Roofline Chart to see a list of performance improvement recommendations for the loop.

• You can view a list of all performance issues detectable in Intel Advisor. Next to **All Advisor-detectable issues**, click to display either **C++** or **Fortran** issues.
• Issues and recommendations are displayed in a list. Under each issue, you'll see an explanation of the issue and recommendations for how to resolve it. You may also see **Read More** links with additional information about the topic.
• Advisor provides confidence levels for the issues and recommendations it identifies. To view the confidence level, hover the cursor over the name of a specific issue or recommendation.
• You can jump to a specific issue by clicking its name in the list of detected issues to the right.

**Roofline Conclusions**

This section contains a roofline chart for the selected loop, as well as:

• Performance issues identified with the loop
• Recommendations to fix those issues
• General cautions about problems that may influence performance

Use the information in this section to help answer the question, "*Why is my loop/function placed in its current location on the Roofline chart?*”

**See Also**

Advanced View Pane (Survey Report)
Filters Pane
Pane Purpose and Usage | Pane Location | Pane Controls | Data Reference

Filters Pane Purpose and Usage
Filter analysis data by a variety of criteria, such as module, loop/function, and vectorized/non-vectorized.

Filters Pane Location
Top of the:
- Survey Report window
- Refinements Reports window

Filters Pane Controls

There are many controls available to help you focus on the data most important to you, including the following:

1. Click the control to save a read-only result snapshot you can view any time.
   Intel Advisor stores only the most recent analysis result. Visually comparing one or more snapshots to each other or to the most recent analysis result can be an effective way to judge performance improvement progress.
   To open a snapshot, choose File > Open > Result...

2. Click the various Filter controls to temporarily limit displayed data based on your criteria.

3. Click the control to view loops in non-executed code paths for various instruction set architectures (ISAs). Prerequisites:
   - Compile the target application for multiple code paths using the Intel compiler.
   - Enable the Analyze loops in not executed code path checkbox in Project Properties > Analysis Target > Survey Hotspots Analysis.

4. This toggle control currently combines two features: The View Configurator and the Smart Mode filter.
• **View Configurator** - Toggle on the **Customize View** control to choose the view layout to display: Default, Smart Mode, or a customized view layout. To create a customized view layout you can apply to this and other projects:

1. Click the Settings control next to the View Layout drop-down list to open the Configure Columns dialog box.
2. Choose an existing view layout in the Configuration drop-down list.
3. Enable/disable columns to show/hide.

Outcome: Copy n is added to the name of the selected view layout in the Configuration drop-down list.

4. Click the Rename button and supply an appropriate name for the customized view layout.
5. Click OK to save the customized view layout.

• **Smart Mode Filter** - Toggle on the **Customize View** control to temporarily limit displayed data to the top potential candidates for optimization based on Total CPU Time (the time your application spends actively executing a function/loop and its callees). In the Top drop-down list, choose one of the following:

- The Number of top loops/functions to display
- The Percent of Total CPU Time the displayed loops/functions must equal or exceed

5. Click the button to search for specific data.

6. Click the tab to open various Intel Advisor reports or views.

7. Right-click a column header to:
   - Hide the associated report column.
   - Resume showing all available report columns.
   - Open the Configure Columns dialog box (see #4 for more information).

8. Click the toggle to show all available columns in a column set, and resume showing a limited number of preset columns in a column set.

9. Click the control to:
   - Show options for customizing data in a column or column set.
   - Open the Configure Columns dialog box (see #4 for more information).

For example, click the control in the Compute Performance column set to:

- Show data for floating-point operations only, for integer operations only, or for the sum of floating-point and integer operations.
- Determine what is counted as an integer operation in integer calculations:
  - Choose **Show Pure Compute Integer Operations** to count only ADD, MUL, IDIV, and SUB operations.
  - Choose **Show All Operations Processing Integer Data** to count ADD, ADC, SUB, MUL, IMUL, DIV, IDIV, INC/DEC, shift, and rotate operations.

10. Click the control to show/hide a chart that helps you visualize actual performance against hardware-imposed performance ceilings, as well as determine the main limiting factor (memory bandwidth or compute capacity), thereby providing an ideal roadmap of potential optimization steps.
11. Click a data row in the top of the Survey Report to display more data specific to that row in the bottom of the Survey Report. Double-click a loop data row to display a Survey Source window. To more easily identify data rows of interest:

- \( f \) = Vectorized function
- \( \circ \) = Vectorized loop
- \( f \) = Scalar function
- \( \circ \) = Scalar loop

12. Click a checkbox to mark a loop for deeper analysis.

13. If present, click the image to display code-specific how-can-I-fix-this-issue? information in the Recommendations pane.

14. If present, click the image to view the reason automatic compiler vectorization failed in the Why No Vectorization? pane.

15. Click the control to show/hide the Workflow pane.

**Loop Information Pane (Survey Report)**

**Pane Purpose and Usage | Pane Location | Pane Controls | Data Reference**

**Loop Information Pane Purpose and Usage**

View integrated compiler report data and Intel Advisor performance data for target application loops, and mark a loop for deeper analysis.

**Loop Information Pane Location**

Top of the Survey Report window

**Loop Information Pane Controls**

There are many controls available to help you focus on the data most important to you, including the following:

1. Click the control to save a read-only result snapshot you can view any time.

   Intel Advisor stores only the most recent analysis result. Visually comparing one or more snapshots to each other or to the most recent analysis result can be an effective way to judge performance improvement progress.
To open a snapshot, choose **File > Open > Result...**

2 Click the various **Filter** controls to temporarily limit displayed data based on your criteria.

3 Click the control to view loops in non-executed code paths for various instruction set architectures (ISAs). Prerequisites:
   - Compile the target application for multiple code paths using the Intel compiler.
   - Enable the **Analyze loops in not executed code path** checkbox in **Project Properties > Analysis Target > Survey Hotspots Analysis**.

4 This toggle control currently combines two features: The **View Configurator** and the **Smart Mode** filter.
   - **View Configurator** - Toggle on the **Customize View** control to choose the view layout to display: **Default, Smart Mode**, or a customized view layout. To create a customized view layout you can apply to this and other projects:
     1 Click the **Settings** control next to the **View Layout** drop-down list to open the **Configure Columns** dialog box.
     2 Choose an existing view layout in the **Configuration** drop-down list.
     3 Enable/disable columns to show/hide.
   
   Outcome: *Copy n* is added to the name of the selected view layout in the **Configuration** drop-down list.
   4 Click the **Rename** button and supply an appropriate name for the customized view layout.
   5 Click **OK** to save the customized view layout.
   - **Smart Mode Filter** - Toggle on the **Customize View** control to temporarily limit displayed data to the top potential candidates for optimization based on **Total CPU Time** (the time your application spends actively executing a function/loop and its callees). In the **Top** drop-down list, choose one of the following:
     * The **Number** of top loops/functions to display
     * The **Percent** of **Total CPU Time** the displayed loops/functions must equal or exceed

5 Click the button to search for specific data.

6 Click the tab to open various Intel Advisor reports or views.

7 Right-click a column header to:
   - Hide the associated report column.
   - Resume showing all available report columns.
   - Open the **Configure Columns** dialog box (see #4 for more information).

8 Click the toggle to show all available columns in a column set, and resume showing a limited number of preset columns in a column set.

9 Click the control to:
   - Show options for customizing data in a column or column set.
   - Open the **Configure Columns** dialog box (see #4 for more information).

For example, click the control in the **Compute Performance** column set to:
   - Show data for floating-point operations only, for integer operations only, or for the sum of floating-point and integer operations.
• Determine what is counted as an integer operation in integer calculations:
  • Choose **Show Pure Compute Integer Operations** to count only ADD, MUL, IDIV, and SUB operations.
  • Choose **Show All Operations Processing Integer Data** to count ADD, ADC, SUB, MUL, IMUL, DIV, IDIV, INC/DEC, shift, and rotate operations.

10 Click the control to show/hide a chart that helps you visualize actual performance against hardware-imposed performance ceilings, as well as determine the main limiting factor (memory bandwidth or compute capacity), thereby providing an ideal roadmap of potential optimization steps.

11 Click a data row in the top of the **Survey Report** to display more data specific to that row in the bottom of the **Survey Report**. Double-click a loop data row to display a **Survey Source** window. To more easily identify data rows of interest:
  • \( \mathfrak{f} \) = Vectorized function
  • \( \bigcirc \) = Vectorized loop
  • \( \mathfrak{s} \) = Scalar function
  • \( \bigcirc \) = Scalar loop

12 Click a checkbox to mark a loop for deeper analysis.

13 If present, click the image to display code-specific how-can-I-fix-this-issue? information in the **Recommendations** pane.

14 If present, click the image to view the reason automatic compiler vectorization failed in the **Why No Vectorization?** pane.

15 Click the control to show/hide the **Workflow** pane.

**Survey Source Window**

**Window Purpose and Usage**
Use this window to examine the source code and the execution time for each source line.

**Survey Source Window Access**
Do one of the following to access this window from the **Survey Report** window:
  • Double-click a row in the **Loop Information** pane or in the **Top Down** tab of the **Advanced View** pane.
  • Right-click a loop and select **View Source**.

**Survey Source Window Regions and Usage**
From top left to bottom right:
  • **View Activation** pane - Enable or disable views shown in the Source view
  • **Source View** pane - View user-visible source code representation of the selected site.
  • **Assembly View** pane - View assembly representation for a selected loop.
  • **Call Stack View** pane - View the call stack for the selected code region. Click to display related code regions in the **File: filename** pane, or click to display the context menu.

**Assembly View Pane (Survey Source Window)**

Pane Purpose and Usage | Pane Location | Pane Controls | Data Reference
Assembly View Pane Purpose and Usage
View assembly representation for a selected loop.

Assembly View Pane Location
Middle of Survey Source window

Assembly View Pane Controls

<table>
<thead>
<tr>
<th>Use This</th>
<th>To Do This</th>
</tr>
</thead>
<tbody>
<tr>
<td>Source lines</td>
<td>You can navigate to related source lines or explore assembly representation of the code by using the Call Stack with Loops pane.</td>
</tr>
<tr>
<td>Select multiple source</td>
<td>To view the accumulated time values for multiple source lines below the Self Time column, or enable you to copy multiple source lines using the context menu. Viewing accumulated time can help you decide how to divide the work.</td>
</tr>
<tr>
<td>lines</td>
<td></td>
</tr>
</tbody>
</table>

Call Stack View Pane (Survey Source Window)
Pane Purpose and Usage | Pane Location | Pane Controls | Data Reference

Call Stack View Pane Purpose and Usage
View the call stack for the selected code region. Click to display related code regions in the File: filename pane, or click to display the context menu.

Call Stack View Pane Location
Bottom right of Survey Source window

Call Stack View Pane Controls

<table>
<thead>
<tr>
<th>Use This</th>
<th>To Do This</th>
</tr>
</thead>
<tbody>
<tr>
<td>, , or icon.</td>
<td>View whether:</td>
</tr>
<tr>
<td></td>
<td>• The row displayed is for a function or a loop. A function or loop icon indicates that source code is available.</td>
</tr>
<tr>
<td></td>
<td>• Source code is available for viewing and editing. A function or loop icon indicates that source code is not available.</td>
</tr>
<tr>
<td>Click a row in the Call</td>
<td>Displays source code for the specified location in the call stack tree.</td>
</tr>
<tr>
<td>Stack pane</td>
<td></td>
</tr>
<tr>
<td>Pane border (drag)</td>
<td>Resize the pane.</td>
</tr>
<tr>
<td>Right click a row in the</td>
<td>Customize call stack presentation by using the Call Stack context menu.</td>
</tr>
<tr>
<td>Call Stack pane</td>
<td></td>
</tr>
</tbody>
</table>

Source View Pane (Survey Source Window)
Pane Purpose and Usage | Pane Location | Pane Controls | Data Reference

Source View Pane Purpose and Usage
View user-visible source code representation of the selected site.
Source View Pane Location
Middle of Survey Source window

Source View Pane Controls

<table>
<thead>
<tr>
<th>Use This</th>
<th>To Do This</th>
</tr>
</thead>
<tbody>
<tr>
<td>Source lines</td>
<td>You can navigate to related source lines or explore your source code by using the Call Stack with Loops pane.</td>
</tr>
<tr>
<td>Double-click a source line</td>
<td>To open your code editor to the corresponding source file. The editor allows you to add annotations to your code (right-click to open the context menu). You can use the annotation assistant pane to help you copy parallel site and task annotations.</td>
</tr>
<tr>
<td></td>
<td>• On Windows* OS:</td>
</tr>
<tr>
<td></td>
<td>• When using Microsoft Visual Studio*, the Visual Studio code editor appears with the file open at the corresponding location.</td>
</tr>
<tr>
<td></td>
<td>• When using the Intel Advisor GUI, the file type association (or Open With dialog box) determines the editor used.</td>
</tr>
<tr>
<td></td>
<td>• On Linux* OS: When using the Intel Advisor GUI, the editor defined by the Options &gt; Editor dialog box appears with the file open at the corresponding location.</td>
</tr>
<tr>
<td></td>
<td>To return to the Survey Source or Survey Report window, click the Result tab.</td>
</tr>
<tr>
<td>Select multiple source lines</td>
<td>To view the accumulated time values for multiple source lines below the Self Time column, or enable you to copy multiple source lines using the context menu. Viewing accumulated time can help you decide how to divide the work.</td>
</tr>
<tr>
<td>Right click a source line</td>
<td>Display a context menu to: open your code editor to the corresponding source line, copy the selected source line(s) to the clipboard, or display context-sensitive help relevant to the selected loop or function.</td>
</tr>
</tbody>
</table>

View Activation Pane (Survey Source Window)
Pane Purpose and Usage | Pane Location | Pane Controls | Data Reference

View Activation Pane Purpose and Usage
Enable or disable views shown in the Source view

View Activation Pane Location
Top of Survey Source window

View Activation Pane Controls

<table>
<thead>
<tr>
<th>Use This</th>
<th>To Do This</th>
</tr>
</thead>
<tbody>
<tr>
<td>Source button</td>
<td>Display source code of the selected loop/function in the source code pane.</td>
</tr>
<tr>
<td>Assembly button</td>
<td>Display assembly code of the selected loop/function in the assembly source pane.</td>
</tr>
</tbody>
</table>
Configure Columns Dialog Box

Configure Columns Purpose and Usage | Configure Columns Access | Configure Columns Controls

Configure Columns Purpose and Usage

Use this dialog box to create or apply custom column layouts when displaying the results of a Survey analysis.

Configure Columns Access

To access this dialog box, first run a Survey analysis, then do one of the following:

- In the top right of the interface, toggle on the Customize View control. A gear icon appears in the top right of both the Survey grid and the Top Down tab. Click the gear icon at the top of the grid you’d like to customize.
- After running the FLOP analysis, click the gear icon at the top right of either the Compute Performance or Memory columns.

Configure Columns Controls

1. Choose an existing view layout in the Configuration drop-down list.
2. Enable/disable columns to show/hide.
   A new view layout is added to the Configuration drop-down list, with Copy n added to the name of the original layout.
3. Click the Rename button and supply an appropriate name for the customized view layout.
4. Click OK to save and apply the customized view layout.

   You can choose to display one layout in the main Survey Report grid, and choose another layout for the Top Down tab.

   NOTE Hiding or showing columns in a column layout will apply your changes to any grid (the Survey Report grid or the Top Down tab) that is currently using the layout. However, you can rearrange columns in one grid without affecting another grid.

See Also

- Survey Analysis
- Top Down Tab

Trip Counts Analysis

Purpose and Usage | Prerequisites | Run | Data Reference

Trip Counts Analysis Purpose and Usage

Run a Trip Counts analysis to dynamically identify the number of times loops are invoked and execute (sometimes called call count/loop count and iteration count respectively) and add this data to the Survey Report. Use Trip Counts data to:

- Detect loops with too-small trip counts and trip counts that are not a multiple of vector length.
• Analyze parallelism granularity more deeply.

**Trip Counts Analysis Prerequisites**

To prepare to run a Trip Counts analysis:

1. Perform the same preparation steps for a basic *Survey Report*.
2. In the Analysis Target tab in the Project Properties dialog box, choose the *Survey Trip Count Analysis* type and review the parameters. (*Survey Trip Counts Analysis* parameters should be similar to those for the *Survey Hotspots Analysis* type.)

---

**Tip**

- If you plan to run *Refinement Reports*, set parameters at the same time you set parameters for the *Survey Hotspots Analysis* and *Survey Trip Count Analysis* types. If possible, use the *Inherit settings from Survey Hotspots Analysis Type* checkbox for other Analysis Types.
- Setting search directories in the Binary/Symbol Search tab and Source Search tabs is optional for the Vectorization Advisor.
- Choose the optimal compiler settings. The most up-to-date settings are in *User Guide: Before You Begin*.

---

**Run a Trip Counts Analysis**

To run a Trip Counts analysis: In the Vectorization Workflow tab, ensure the *Trip Counts* checkbox is selected, then click the **Collect** button under *Find Trip Counts and FLOP*.

While analysis is running, you can do the following in the Workflow tab:

- Stop analysis and data collection, and retain the already collected data: Click the **Stop** button.
- Cancel analysis and data collection, and discard the collected data: Click the **Cancel** button.
- Interrupt post-collection data finalization: Click the **Cancel** button.

After analysis is complete, the Intel Advisor adds data to the *Trip Counts* column set in the *Survey Report*.

---

**Tip**

- Collecting Trip Counts data may substantially increase report generation time. There are a variety of techniques available to minimize data collection, result size, and execution time. Check *Minimizing Analysis Overhead*.

---

**FLOP Analysis**

**Purpose and Usage | Prerequisites | Run | Data Reference**

**FLOP Analysis Purpose and Usage**

Run a FLOP analysis to measure both floating-point and integer operations, and memory traffic, and add this data to the *Survey Report*. Use the FLOP analysis to generate application memory usage and performance values that help you make better decisions about your vectorization strategy.
FLOP Analysis Prerequisites

To prepare to run a FLOP analysis:

1. Perform the same preparation steps for a basic Survey Report.
2. In the Analysis Target tab in the Project Properties dialog box, choose the Survey Trip Count Analysis type and review the parameters. (Survey Trip Counts Analysis parameters should be similar to those for the Survey Hotspots Analysis type.)

Tip

- If you plan to run Refinement Reports, set parameters at the same time you set parameters for the Survey Hotspots Analysis and Survey Trip Count Analysis types. If possible, use the Inherit settings from Survey Hotspots Analysis Type checkbox for other Analysis Types.
- Setting search directories in the Binary/Symbol Search tab and Source Search tabs is optional for the Vectorization Advisor.
- Choose the optimal compiler settings. The most up-to-date settings are in User Guide: Before You Begin.

Run a FLOP Analysis

To run a FLOP analysis: In the Vectorization Workflow tab, ensure the FLOP checkbox is selected, then click the button under Find Trip Counts and FLOP.

While analysis is running, you can do the following in the Workflow tab:

- Stop analysis and data collection, and retain the already collected data: Click the button.
- Cancel analysis and data collection, and discard the collected data: Click the button.
- Interrupt post-collection data finalization: Click the button.

After analysis is complete, the Intel Advisor adds data to the Compute Performance and Memory column sets in the Survey Report. By default, FLOP data is initially displayed.

Compute Performance column

Click the control in the Compute Performance column set header and choose the desired drop-down option to:

- Show data for floating-point operations only, for integer operations only, or for both floating-point and integer operations.
- Determine what is counted as an integer operation in integer calculations:
  - Choose Show Pure Compute Integer Operations to count only ADD, MUL, IDIV, and SUB operations.
  - Choose Show All Operations Processing Integer Data to count ADD, ADC, SUB, MUL, IMUL, DIV, IDIV, INC/DEC, shift, and rotate operations.

Memory column

Click the gear icon in the Memory column set header and choose the desired drop-down option to determine which columns to display in the grid:

NOTE This data is only available if cache simulation is enabled. By default, Advisor collects only L1 traffic, so you will not be able to select memory levels or loads/stores.
Show data for L1, L2, L3, or DRAM memory metrics, or show a Customized Column Layout.

Show data for memory load operations only, store operations only, or the sum of both.

You can choose to hide the current column, Show All Columns, or customize the columns displayed in the grid by choosing Configure Column Layouts.

Tip

- Collecting FLOP and integer data may substantially increase report generation time. There are a variety of techniques available to minimize data collection, result size, and execution time. Check Minimizing Analysis Overhead

Roofline Analysis

Roofline Analysis Purpose

Roofline analysis helps you visualize actual performance against hardware-imposed performance ceilings, as well as determine the main limiting factor (memory bandwidth or compute capacity), thereby providing an ideal roadmap of potential optimization steps.

Use the Roofline chart to answer the following questions:

- What is the maximum achievable performance with your current hardware resources?
- Does your application work optimally on current hardware resources?
- If not, what are the best candidates for optimization?
- Is memory bandwidth or compute capacity limiting performance for each optimization candidate?

Intel® Advisor includes the following Roofline models that you can use to analyze your application:

- Basic Cache-Aware Roofline (default), which represents self data and cumulative traffic-based arithmetic intensity
- Roofline with Callstacks, which represents total data and allow you to investigate the source of loops/functions
- Memory-Level Roofline, which collects metrics for all memory levels and allows you to observe each loop/function at different cache levels

Basic Roofline Analysis

In the Vectorization Workflow tab, click the Collect control under Run Roofline.

The Intel Advisor executes the target application twice to:

- Measure the hardware limitations of your machine and collect loop/function timings using the Survey analysis.
- Collect FLOP and integer operations data, and memory traffic data, using the Trip Counts and FLOP analysis - this collection can take three to four times longer than the Survey analysis.

After both analyses are complete, the Intel Advisor adds a Roofline chart to the Survey Report.

By default, Intel Advisor runs the Cache-Aware Roofline, which represents self data and cumulative traffic-based arithmetic intensity.

Roofline Chart Data

The Roofline chart plots an application’s achieved performance and arithmetic intensity against the machine’s maximum achievable performance:
- Arithmetic intensity (x axis) - measured in number of floating-point operations (FLOPs) and/or integer operations (INTOPs) per byte, based on the loop/function algorithm, transferred between CPU/VPU and memory
- Performance (y axis) - measured in billions of floating-point operations per second (GFLOPS) and/or billions of integer operations per second (GINTOPS)

In general:
- The size and color of each **Roofline** chart dot represent relative execution time for each loop/function. Large red dots take the most time, so are the best candidates for optimization. Small green dots take less time, so may not be worth optimizing.
- **Roofline** chart diagonal lines indicate memory bandwidth limitations preventing loops/functions from achieving better performance without some form of optimization. For example: The **L1 Bandwidth** roofline represents the maximum amount of work that can get done at a given arithmetic intensity if the loop *always* hits L1 cache. A loop does not benefit from L1 cache speed if a dataset causes it to miss L1 cache too often, and instead is subject to the limitations of the lower-speed L2 cache it *is* hitting. So a dot representing a loop that misses L1 cache too often but hits L2 cache is positioned somewhere below the **L2 Bandwidth** roofline.
- **Roofline** chart horizontal lines indicate compute capacity limitations preventing loops/functions from achieving better performance without some form of optimization. For example: The **Scalar Add Peak** represents the peak number of add instructions that can be performed by the scalar loop under these circumstances. The **Vector Add Peak** represents the peak number of add instructions that can be performed by the vectorized loop under these circumstances. So a dot representing a loop that is not vectorized is positioned somewhere below the **Scalar Add Peak** roofline.
- A dot cannot exceed the topmost rooflines, as these represent the maximum capabilities of the machine; however, not all loops can utilize maximum machine capabilities.
- The greater the distance between a dot and the highest achievable roofline, the more opportunity exists for performance improvement.

In the following **Roofline** chart representation, loops A and G (large red dots), and to a lesser extent B (yellow dot far below the roofs), are the best candidates for optimization. Loops C, D, and E (small green dots) and H (yellow dot) are poor candidates because they do not have much room to improve or are too small to have significant impact on performance.
Roofline Chart Controls

There are several controls to help you show/hide the **Roofline** chart:

1. Click to toggle between **Roofline** chart view and **Survey Report** view.
2. Click to toggle to and from side-by-side **Roofline** chart and **Survey Report** view.
3. Drag to adjust the dimensions of the **Roofline** chart and **Survey Report**.

There are several controls to help you focus on the **Roofline** chart data most important to you, including the following.
1. **Select Loops by Mouse Rect**: Select one or more loops/functions by tracing a rectangle with your mouse.

2. **Zoom by Mouse Rect**: Zoom in and out by tracing a rectangle with your mouse. You can also zoom in and out using your mouse wheel.

3. **Move View By Mouse**: Move the chart left, right, up, and down.

4. **Undo or Redo**: Undo or redo the previous zoom action.

5. **Cancel Zoom**: Reset to the default zoom level.

6. **Export as x**: Export the chart as a dynamic and interactive HTML or SVG file that does not require the Intel Advisor viewer for display. Use the arrow to toggle between the options.

---

Use the **Cores** drop-down toolbar to:

1. Adjust rooflines to see practical performance limits for your code on the host machine.

2. Build roofs for single-threaded applications (or for multi-threaded applications configured to run single threaded, such as one thread-per-rank for MPI applications. (You can use Intel Advisor filters to control the loops displayed in the Roofline chart; however, the Roofline chart does not support the Threads filter.)

Choose the appropriate number of CPU cores to scale roof values up or down:

- 1 – if your code is single-threaded
- Number of cores equal or close to the number of threads – if your code has fewer threads than available CPU cores
- Maximum number of cores – if your code has more threads than available CPU cores

By default, the number of cores is set to the number of threads used by the application (even values only).

You’ll see the following options if your code is running on a multisocket PC:

- Choose **Bind cores to 1 socket** (default) if your application binds memory to one socket. For example, choose this option for MPI applications structured as one rank per socket.

  **NOTE** This option may be disabled if you choose a number of CPU cores exceeding the maximum number of cores available on one socket.

- Choose **Spread cores between all n sockets** if your application binds memory to all sockets. For example, choose this option for non-MPI applications.
3. Toggle the display between floating-point (FLOP), integer (INT) operations, and mixed operations (floating-point and integer).

4. If you collected Roofline with Callstacks: Enable the display of Roofline with Callstacks additions to the Roofline chart.

Display Roofline chart data from other Intel Advisor results or non-archived snapshots for comparison purposes.

Use the drop-down toolbar to:

- Load a result/snapshot and display the corresponding filename in the Compared Results region.
- Clear a selected result/snapshot and move the corresponding filename to the Ready for comparison region.

**Note:** Click a filename in the Ready for comparison region to reload the result/snapshot.

- Save the comparison itself to a file.

**NOTE** The arrowed lines showing the relationship among loops/functions do not reappear if you upload the comparison file.

Click a loop/function dot in the current result to show the relationship (arrowed lines) between it and the corresponding loop/function dots in loaded results/snapshots.

5. Add visual indicators to the Roofline chart to make the interpretation of data easier, including performance limits and whether loops/functions are memory bound, compute bound, or both.

Use the drop-down toolbar to:

- Show a vertical line from a loop/function to the nearest and topmost performance ceilings by enabling the Display roof rulers checkbox. To view the ruler, hover the cursor over a loop/function. Where the line intersects with each roof, labels display hardware performance limits for the loop/function.
• If you collected Roofline for All Memory Levels: Visually emphasize the relationships among displayed memory levels and roofs and for a selected loop/function dot by enabling the **Show memory level relationships** checkbox.

• Color the roofline zones to make it easier to see if enclosed loops/functions are fundamentally memory bound, compute bound, or bound by compute and memory roofs by enabling the **Show Roofline boundaries** checkbox.

The preview picture is updated as you select guidance options, allowing you to see how changes will affect the Roofline chart’s appearance. Click **Apply** to apply your changes, or **Default** to return the Roofline chart to its original appearance.

Once you have a loop/function’s dots highlighted, you can zoom and fit the Roofline chart to the dots for the selected loop/function by once again double-clicking the loop/function or pressing **SPACE** or **ENTER** with the loop/function selected. Repeat this action to return to the original Roofline chart view.

To hide the labeled dots, select another loop/function, or double-click an empty space in the Roofline chart.

6 **Roofline View Settings:** Adjust the default scale setting to show:

• The optimal scale for each Roofline chart view
• A scale that accommodates all Roofline chart views

**Roofs Settings:** Change the visibility and appearance of roofline representations (lines):

• Enable calculating roof values based on single-threaded benchmark results instead of multi-threaded.
• Click a **Visible** checkbox to show/hide a roofline.
• Click a **Selected** checkbox to change roofline appearance: display a roofline as a solid or a dashed line.
• Manually fine-tune roof values in the **Value** column to set hardware limits specific to your code.

• **Loop Weight Representation:** Change the appearance of loop/function weight representations (dots):

  • **Point Weight Calculation:** Change the **Base Value** for a loop/function weight calculation.
  • **Point Weight Ranges:** Change the **Size**, **Color**, and weight **Range (R)** of a loop/function dot. Click the + button to split a loop weight range in two. Click the - button to merge a loop weight range with the range below.
  • **Point Colorization:** color loop/function dots by weight ranges or by type (vectorized or scalar). You can also change the color of loop with no self time.

You can save your Roofs Settings or Point Weight Representation configuration to a JSON file or load a custom configuration.

7 **Zoom in and out using numerical values.**

8 Click a loop/function dot to:

• Outline it in black.
• Display metrics for it.
• Display corresponding data in other window tabs.

Right-click a loop/function dot or a blank area in the **Roofline** chart to perform more functions, such as:

• Further simplify the **Roofline** chart by filtering out (temporarily hiding a dot), filtering in (temporarily hiding all other dots), and clearing filters (showing all originally displayed dots).
• Copy data to the clipboard.
Show/hide the metrics pane:
• Review the basic performance metrics in the **Point Info** pane.
• *If you collected the Roofline for All Memory Levels:* Review how efficiently the loop/function uses cache and what memory level bounds the loop/function in the **Memory Metrics** pane.

Display the number and percentage of loops in each loop weight representation category.

---

**Roofline with Callstacks**

Intel Advisor basic Roofline model, the Cache-Aware Roofline Model (CARM), offers *self data* capability. Intel Advisor Roofline with Callstacks feature extends the basic model with *total data* capability:

- **Self data** = Memory access, FLOPs, and duration related only to the loop/function itself and excludes data originating in other loops/functions called by it
- **Total data** = Data from the loop/function itself and its inner loops/functions

The total-data capability in the Roofline with Callstacks feature can help you:

- Investigate the source of loops/functions instead of just the loops/functions themselves.
- Get a more accurate view of loops/functions that behave differently when called under different circumstances.
- Uncover design inefficiencies higher up the call chain that could be the root cause of poor performance by smaller loops/functions.

**To run Roofline with Callstacks:**

1. Enable the **With Callstacks** checkbox in the **Vectorization Workflow** tab under **Run Roofline**.
2. Run the Roofline analysis. Upon completion, the Intel Advisor displays a **Roofline** chart.
3. Enable the **With Callstacks** checkbox in the **Roofline** chart.

**Roofline with Callstacks Chart Data**

The following **Roofline** chart representation shows some of the added benefits of the Roofline with Callstacks feature, including:

- A navigable, color-coded **Callstack** pane that shows the entire call chain for the selected loop/function, but excludes its callees
- Visual indicators (caller and callee arrows) that show the relationship among loops and functions
- The ability to simplify dot-heavy charts by collapsing several small loops into one overall representation

Loops/functions with no self data are grayed out when expanded and in color when collapsed. Loops/functions with self data display at the coordinates, size, and color appropriate to the data when expanded, but have a gray halo of the size associated with their total time. When such loops/functions are collapsed, they change to the size and color appropriate to their total time and, if applicable, move to reflect the total performance and total arithmetic intensity.
Enable the display of Roofline with Callstacks additions to the Roofline chart.

2. Show/hide loop/function descendants:
   - Click a loop/function dot \(\square\) control to collapse descendant dots into the parent dot.
   - Click a loop/function dot \(\oplus\) control to show descendant dots and their relationship via visual indicators to the parent dot.

You can also right-click a loop/function dot to open the context menu and expand/collapse the loop/function subtree.

3. Show/hide the Callstack and other panes.

4. Click an item in the Callstack pane to flash the corresponding loop/function dot in the Roofline chart.
• Right-click an item in the Callstack pane to open the context menu and expand/collapse the item subtree.
You can also click an item in the Callstack pane to flash the corresponding loop/function dot in the Roofline chart.

Memory-Level Roofline
Using the cache simulation, Intel Advisor evaluates the data transactions between the different memory layers available on your system and generate a Memory-level Roofline chart. You can choose which memory levels (L1, L2, L3, DRAM) to plot dots and examine this data for a selected loop/function in greater detail, displaying labeled dots with arithmetic intensity for the loop/function at each memory level.

To run Memory-Level Roofline:

1. Enable the For All Memory Levels checkbox in the Vectorization Workflow tab under Run Roofline.
2. Run the Roofline analysis. Upon completion, the Intel Advisor displays a Roofline chart.
3. In the Roofline chart, verify that Show memory level relationships checkbox is enabled in the Guidance drop-down menu.
4. In the filter drop-down menu, select which memory levels to show dots for from the Memory Level section.

NOTE By default, the Memory-Level Roofline chart is generated for the system cache configuration.
You can also generate the chart for a custom cache configuration:

1. Go to Project Properties > Trip Count and FLOP.
2. In the Cache simulator configuration field, click Modify.
3. Click Add and enter/select the desired cache configurations.
4. Run the Roofline for all memory levels.

Memory-Level Roofline Data
Memory-Level Roofline model allows you to observe each loop/function at different cache level and compare arithmetic intensities to understand where performance decreases. The roofs represent the best possible bandwidths for each memory level.

Review the changes in the traffic from one memory level to another and compare it to respective to identify the memory hierarchy bottleneck for the kernel and determine optimization steps based on this information.

• The vertical distance between memory dots and their respective roofline shows how much you are limited by a given memory subsystem. If a dot is close to its roof line, it means that the kernel is limited by the performance of this memory level.
The horizontal distance between each dot indicates how efficiently the loop/function uses cache. For example, if L3 and DRAM dots are very close on the horizontal axis for a single loop, the loop/function uses L3 and DRAM similarly. This means that it does not use L3 and DRAM efficiently. Improve re-usage of data in the code to improve application performance.

Arithmetic intensity determines the order in which dots are plotted, which can provide some insight into your code’s performance. For example, the L1 dot should be the largest and first plotted dot on the chart from left to right. However, memory access type, latency, or technical issues can change the order of the dots. Continue to run the Memory Access Pattern analysis to investigate this issue.

Memory-Level Roofline Chart Controls

1. Visually emphasize the relationships among displayed memory levels and roofs for a selected loop/function dot by enabling the Show memory level relationships checkbox.

   **NOTE** This checkbox is enabled by default.

2. Use the drop-down toolbar to:
   - Select the Memory Level(s) to show for each loop/function in the chart (L1, L2, L3, DRAM).
   - Select which Memory Operation Types(s) to display data for in the Roofline chart: Loads, Stores, or Loads and Stores.

3. Double-click a dot or select a dot and press SPACE or ENTER to examine how the relationships between displayed memory levels and roofs:
   - Labeled dots are displayed, representing memory levels for the selected loop/function. Lines connect the dots to indicate that they correspond to the selected loop/function.

   **NOTE** If you have chosen to display only some memory levels in the chart using the Memory Level option, unselected memory levels are displayed with X marks.

   - An arrowed line is displayed, pointing to the memory level roofline that bounds the selected loop. If the arrowed line cannot be displayed, a message will pop up with instructions on how to fix it.

4. Show/hide the Memory Metrics and other panes.

   In the Memory Metrics pane:
• Review the time spent processing requests for each memory level reported in the Impacts histogram. A big value indicates a memory level that bounds the selected loop.
• Review an amount of data that passes through each memory level reported in the Shares histogram.

What Do I Do Next?
See the Intel® Advisor Cookbook recipes to learn how to use the Roofline for specific use cases:
• Identify Bottlenecks Iteratively: Cache-Aware Roofline
• Visualize Performance Improvements with Roofline Compare

See the Use Automated Roofline Chart to Make Optimization Decisions tutorial to learn how to:
• Address memory bandwidth bottlenecks.
• Address compute capacity bottlenecks.
• Identify the real bottlenecks.

Dependencies and Memory Access Patterns Analyses

Refinement Reports Purpose and Usage
Intel Advisor offers two refinement analyses:
• Dependencies analysis (optional) - For safety purposes, the compiler is often conservative when assuming data dependencies. Run a Dependencies analysis to check for real data dependencies in loops the compiler did not vectorize because of assumed dependencies. If real dependencies are detected, the analysis can provide additional details to help resolve the dependencies. Your objective: Identify and better characterize real data dependencies that could make forced vectorization unsafe.
• Memory Access Patterns (MAP) analysis (optional) - Run a MAP analysis to check for various memory issues, such as non-contiguous memory accesses and unit stride vs. non-unit stride accesses. Your objective: Eliminate issues that could lead to significant vector code execution slowdown or block automatic vectorization by the compiler.

Refinement Reports Regions
• Filters pane - Filter analysis data by a variety of criteria, such as module, loop/function, and vectorized/non-vectorized.
• Loop Information pane - View integrated Memory Access Patterns and Dependencies analysis data.
• Advanced View pane - includes the following tabs:
  • Memory Access Patterns Report tab - View information about types of memory access inside selected loops/functions. (Vectorization Advisor only.)
  • Dependencies Report tab - View any predicted data sharing problems and informational remark messages.
  • Recommendations tab - provides memory-specific recommendations.

Loop Information Pane (Refinement Reports)

Loop Information Pane Purpose and Usage
View integrated Memory Access Patterns and Dependencies analysis data.
Loop Information Pane Location
Top of the Refinement Reports window

Loop Information Pane Controls

<table>
<thead>
<tr>
<th>Use This</th>
<th>To Do This</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data grid row</td>
<td>Display more data specific to the selected loop in the Advanced View pane (click).</td>
</tr>
<tr>
<td>Double-click a row</td>
<td>Display source view tab with source and assembly code, and Details view.</td>
</tr>
</tbody>
</table>

Running Multiple Analyses Automatically

Intel® Advisor provides the **Batch Mode** feature that enables you to select several Advisor collectors to run automatically in a queue. You can specify several options to reduce the amount of collected data. Using this mode might be useful if you want to fully analyze with Intel® Advisor an application that takes a lot of time to execute.

To enable the **Batch Mode**, you must be in the Intel® Advisor Vectorization Workflow. Use the **ON/OFF** selector to enable/disable the **Batch Mode**.

<table>
<thead>
<tr>
<th>Use This</th>
<th>To Do This</th>
</tr>
</thead>
<tbody>
<tr>
<td>[ ] ON Batch mode</td>
<td>To switch to the batch mode and back.</td>
</tr>
<tr>
<td>Check boxes</td>
<td>To select the collectors that you want to run in the batch mode.</td>
</tr>
<tr>
<td><strong>Manual Selection</strong></td>
<td>Select loops for Memory Access Patterns and Dependencies analysis. You can select loops in the Survey report view after running the Survey analysis.</td>
</tr>
<tr>
<td><strong>Automatic Selection</strong></td>
<td>Let the Intel® Advisor automatically select loops for deeper analysis according to the specified (check-marked) criteria.</td>
</tr>
<tr>
<td><strong>Scalar Serial Loops Only</strong></td>
<td>Intel® Advisor will analyze scalar loops that are not included into any parallel region. The loops that do not meet this criteria will be excluded from the analysis.</td>
</tr>
<tr>
<td>(for Dependencies analysis only)</td>
<td></td>
</tr>
<tr>
<td><strong>Innermost Loops Only</strong></td>
<td>Intel® Advisor will analyze innermost loops. All other loops will be excluded from analysis.</td>
</tr>
<tr>
<td>(for Dependencies analysis only)</td>
<td></td>
</tr>
<tr>
<td><strong>Diagnosed as &quot;Vector Dependence Prevent Vectorization&quot; Only</strong></td>
<td>Intel® Advisor will analyze scalar loops that the compiler cannot vectorize automatically and reports the &quot;Vector Dependence Prevent Vectorization&quot; diagnostic. All other loops that do not meet this criteria will be excluded from analysis.</td>
</tr>
<tr>
<td>(for Dependencies analysis only)</td>
<td></td>
</tr>
<tr>
<td><strong>With &quot;Possible Inefficient Memory Access Pattern&quot; issue only</strong></td>
<td>Intel® Advisor will analyze the loops with performance issue &quot;Possible Inefficient Memory Access Pattern&quot; detected. The loops that do not meet this criteria will be excluded from the analysis.</td>
</tr>
<tr>
<td>(for Memory Access Patterns analysis only)</td>
<td></td>
</tr>
</tbody>
</table>
Loop hierarchy choice (for Memory Access Patterns analysis only):

- **Innermost Loops Only**
- **Second-level Loops Only**
- **Third-level Loops Only**

Select the hierarchy level of nested loops for analysis. The loops that do not meet this criteria will be excluded from the analysis. "Innermost Loops" stands for loops that don't have nested loops. "Second-level Loops" stands for the loops that include innermost loop(s) only. "Third-level Loops" stands for the loops that include second-level loops only.

Exclude loops without source location (for Memory Access Patterns analysis only)

Intel® Advisor will exclude loops with empty source location from the analysis.

Above 2% of Total CPU Time Only (for Memory Access Patterns analysis only)

Intel® Advisor will analyze only the loops with total time more than 2% of the Total CPU Time. The loops that do not meet this criteria will be excluded from the analysis.

### Dependencies Analysis

**Dependencies Analysis Purpose and Usage**

View any predicted data sharing problems and informational remark messages.

**Dependencies Analysis Prerequisites**

To prepare to run a Dependencies analysis:

1. Do one of the following to open the Analysis Target tab in the Project Properties dialog box:
   - In the Intel Advisor GUI, choose Project > Project Properties...
   - In the Visual Studio IDE, choose Project > Intel Advisor [version] Project Properties....

2. Choose the Dependencies Analysis type, then specify the same application as in the Survey Hotspots Analysis type, but a smaller input data set if possible. Select **Track stack variables** to detect all possible dependencies.


4. Mark one or more un-vectorized loops for deeper analysis by enabling the corresponding checkbox in the Survey Report.

**Tip**

- If possible, use the Inherit settings from Survey Hotspots Analysis Type checkbox.
- The Dependencies Analysis type consumes more resources than the Survey Hotspots Analysis type. If analysis takes too long, consider decreasing the workload.
- Setting search directories in the Binary/Symbol Search tab and Source Search tabs is optional for the Vectorization Advisor.
- Marking loops for deeper analysis using the checkbox is a Vectorization Advisor-specific alternative to adding annotations in target application source code.
- Dependencies analysis is time intensive. The fewer loops you mark for deeper analysis, the less time it takes to collect the data.
- Choose the optimal compiler settings. The most up-to-date settings are in User Guide: Before You Begin.
**Run a Dependencies Analysis**

To run a Dependencies analysis, do one of the following:

- Click the `Collect` button under **Check Dependencies** in the Workflow tab.
- In the Visual Studio Solution Explorer, right-click the project, then choose **Intel Advisor [version] > Start Dependencies Analysis**.

While analysis is running, you can do the following under the **Check Dependencies** step in the Workflow tab:

- Stop analysis and data collection, and retain the already collected data: Click the `button.
- Cancel analysis and data collection, and discard the collected data: Click the `button.
- Interrupt post-collection data finalization: Click the `button.

There are also controls and annotations available to minimize data collection, result size, and execution time. Check See Also.

**Dependencies Report Regions and Purpose**

In the **Dependencies Report** tab at the bottom of the Refinement Report:

- **Problems and Messages** pane - Select the problems that you want to analyze by viewing their associated observations.
- **Code Locations** pane - View details about the code locations for the selected problem in the **Dependencies Report** window. Icons identify the focus code location ![Focus Code Location](image) and related code location ![Related Code Location](image).
- **Filters pane** - Filter contents of the report tab.

Associated **Dependencies Source** window, from top left to bottom right:

- **Focus Code Location** pane - Use this pane to explore source code associated with focus code location in the **Dependencies Source** window.
- **Focus Code Location Call Stack** pane - Use this pane to select which source code appears in the **Focus Code Location** pane in the **Dependencies Source** window.
- **Related Code Locations** pane - Use this pane to explore source code associated with related code locations (related to the focus code location) in the **Dependencies Source** window.
- **Related Code Location Call Stack** pane - Use this pane to select which source code appears in the **Related Code Location** pane.
- **Code Locations** pane - Use this pane to view the details about the code location for the selected problem in the **Dependencies Report** window.
- **Relationship Diagram** pane - Use this pane to view the relationships among code locations for the selected problem.

**Using Dependencies Data**

Use the **Dependencies Report** to view each reported problem and its associated code locations. Use the **Dependencies Source** window to view the focus and related source code regions to help you understand the cause of the reported problem.

To learn about a reported problem, right-click its name in the **Dependencies Report, Problems and Messages** pane and select **What Should I Do Next?**. This displays the help topic for that problem type.

**See Also**

- Dependencies Problem and Message Types Reference

**Code Locations Pane**

**Pane Purpose and Usage** | **Pane Location** | **Pane Controls** | **Data Reference**
**Code Locations Pane Purpose and Usage**

View details about the code locations for the selected problem in the **Dependencies Report** window. Icons identify the focus code location and related code location.

**Code Locations Pane Location**

Bottom of **Dependencies Report** tab

**Code Locations Pane Controls**

<table>
<thead>
<tr>
<th>Use This</th>
<th>To Do This</th>
</tr>
</thead>
<tbody>
<tr>
<td>Title bar</td>
<td>View the problem type.</td>
</tr>
</tbody>
</table>
| Code Location data row(s) | Review related code locations:  
  - ID - Code location identifier  
  - Description - What happens at this code location.  
  - Source - The source file for this code location.  
  - Function - Function name.  
  - Modules - The executable associated with this problem.  
  - State - Indicates whether the problem has been fixed or not. To change the state, use the context menu. |
| Click [ ] to the left of a code location name | Display a code snippet associated with the selected code location. |
| [ ] icon, [ ] icon, or no icon in the Source column | Shows:  
  - Whether this is a related code location.  
  - If code location source code is available for viewing and editing. |
| [ ] icon, [ ] icon, or no icon in the Source column | Shows:  
  - Whether this is the focus code location.  
  - If code location source code is available for viewing and editing. |
| [ ] icon, [ ] icon, or no icon in the Source column | Shows if code location source code is available for viewing and editing. |
| Double-click a code location data row or source line, or right-click and select the **View Source** context menu item | Display the **Dependencies Source** window. |
| Right-click and select the **Edit Source** context menu item | Display a code editor with the corresponding source file.  
  - On Windows* OS:  
    - When using Visual Studio, the Visual Studio code editor appears with the file open at the corresponding location. |
<table>
<thead>
<tr>
<th><strong>Use This</strong></th>
<th><strong>To Do This</strong></th>
</tr>
</thead>
</table>
| • When using the Intel Advisor GUI, the file type association (or **Open With** dialog box) determines the editor used.  
• On Linux* OS: When using the Intel Advisor GUI, the editor defined by the Options > Editor dialog box appears with the file open at the corresponding location. | |
| **Column labels** | Click a column heading to sort the data grid rows in either ascending or descending order. |
| **Pane border** | Resize the pane (drag). |
| Right click a row to display a context menu | Display a context menu to: expand or collapse all code snippets, open the **Dependencies Source** window, edit sources in the code editor, copy the selected data row(s) to the clipboard, mark the state as fixed or not fixed, or display context-sensitive help. |

**Filter Pane (Dependencies Report)**  
**Pane Purpose and Usage**  
Filter contents of the report tab.  
**Filter Pane Location**  
Right side of Dependencies Report tab  
**Filter Pane Controls**

<table>
<thead>
<tr>
<th><strong>Use This</strong></th>
<th><strong>To Do This</strong></th>
</tr>
</thead>
</table>
| **Category column** | Review categories that you can filter, such as **Severity**, **Type**, **Site Name**, **Source**, and so on.  
**NOTE**  
You can apply only one filter criterion per category; however, you can filter the listed problems and messages by multiple categories simultaneously. |
| Click a filter criterion, such as:  
• Error under the **Severity** category  
• Memory Reuse under the **Type** category | View only problems and messages of a specific type, and hide other types of problems and messages in the same category. |
| **All button** to the right of the category's name | To deselect all filter criteria and display all problems and messages in that category. |
| ![Delete] button | To deselect all filter criteria in all categories. |
See Also
• Dependencies Problem and Message Types Reference

Problems and Messages Pane
Pane Purpose and Usage | Pane Location | Pane Controls | Data Reference

Problems and Messages Pane Purpose and Usage
Select the problems that you want to analyze by viewing their associated observations.

Problems and Messages Pane Location
Top of Dependencies Report tab.

Problems and Messages Pane Controls

<table>
<thead>
<tr>
<th>Use This</th>
<th>To Do This</th>
</tr>
</thead>
<tbody>
<tr>
<td>Column labels</td>
<td>Click a column label to sort the data grid data rows in either ascending or descending order.</td>
</tr>
<tr>
<td>Selected data row</td>
<td>Review the characteristics of each data row in the grid. The columns are:</td>
</tr>
<tr>
<td></td>
<td>• ID - Identifier for the problem.</td>
</tr>
<tr>
<td></td>
<td>• 🕵️‍♂️ (severity) - The severity of the problem, such as error 🕵️, warning 🕵️, or an informational remark message 🕵️. For example, the location of parallel sites executed are indicated by the message 🕵️ Parallel Site.</td>
</tr>
<tr>
<td></td>
<td>• Type - The problem type or message type. For more information about a problem, right click to display the context menu.</td>
</tr>
<tr>
<td></td>
<td>• Site Name - The name of the site associated with this problem.</td>
</tr>
<tr>
<td></td>
<td>• Sources - The source file associated with this problem.</td>
</tr>
<tr>
<td></td>
<td>• Modules - The modules (executable) associated with this problem.</td>
</tr>
<tr>
<td></td>
<td>• State - Indicates whether the problem has been fixed or not. To change the state, use the context menu in this pane.</td>
</tr>
<tr>
<td>Pane border</td>
<td>Resize the pane (drag).</td>
</tr>
<tr>
<td>Right click a row to display a context menu</td>
<td>Display a context menu to: open the code editor to the corresponding source line, display the Dependencies Source window, copy the selected data row(s) to the clipboard, or display context-sensitive help for that problem or message.</td>
</tr>
</tbody>
</table>

Dependencies Source Window
Window Purpose and Usage | Window Access | Window Regions | Data Reference

Dependencies Source Window Purpose and Usage
Use this window to examine the source code for a selected Problem, Message, or Code Location. To modify your source code, double-click a source line or use the Edit Source context menu item to display that file in a code editor.

• On Windows® OS:
  • When using Visual Studio, the Visual Studio code editor appears with that file open at the corresponding location.
When using the Intel Advisor GUI, the file type association (or Open With dialog box) determines the editor used.

On Linux* OS: When using the Intel Advisor GUI, the editor defined by the Options > Editor dialog box appears with the file open at the corresponding location.

<table>
<thead>
<tr>
<th>Use This</th>
<th>To Do This</th>
</tr>
</thead>
<tbody>
<tr>
<td>Workflow Tab</td>
<td>Run a tool of your choice and see results in the Result tab.</td>
</tr>
<tr>
<td>Result Tab</td>
<td>Select between available reports.</td>
</tr>
<tr>
<td><strong>Focus Code Location</strong></td>
<td>Explore the source code associated with the focus code location.</td>
</tr>
<tr>
<td><strong>Focus Code Location Call Stack</strong></td>
<td>Select the source code to appear in the Focus Code Location pane.</td>
</tr>
<tr>
<td><strong>Related Code Location</strong></td>
<td>Does not appear if the Focus Code Location does not have a Related Code Location.</td>
</tr>
<tr>
<td><strong>Related Code Location Call Stack</strong></td>
<td>Select the source code to appear in the Related Code Location pane. This pane does not appear if the Focus Code Location does not have a Related Code Location.</td>
</tr>
<tr>
<td>Code Locations pane</td>
<td>View details about the code locations for the selected problem in the Dependencies Source window.</td>
</tr>
<tr>
<td><strong>Relationship Diagram</strong></td>
<td>View the relationships among code locations for the selected problem.</td>
</tr>
</tbody>
</table>

**Dependencies Source Window Access**

To access this window in the Refinement Reports, double-click a data row or use the corresponding context menu item to view the source code associated with a Problem, Message, or Code Location.

**Dependencies Source Window Regions**

From top left to bottom right:

- **Focus Code Location** pane - Use this pane to explore source code associated with focus code location in the Dependencies Source window.
- **Focus Code Location Call Stack** pane - Use this pane to select which source code appears in the Focus Code Location pane in the Dependencies Source window.
- **Related Code Locations** pane - Use this pane to view the details about the code location for the selected problem in the Dependencies Report window.
- **Relationship Diagram** pane - Use this pane to view the relationships among code locations for the selected problem.

**Code Locations Pane (Dependencies Source Window)**

Pane Purpose and Usage | Pane Location | Pane Controls | Data Reference

**Code Locations Pane Purpose and Usage**

Use this pane to view the details about the code location for the selected problem in the Dependencies Report window.
**Code Locations Pane Location**
Bottom left of Dependencies Source window

**Code Locations Pane Controls**

<table>
<thead>
<tr>
<th>Use This</th>
<th>To Do This</th>
</tr>
</thead>
<tbody>
<tr>
<td>Title bar</td>
<td>View the problem type.</td>
</tr>
<tr>
<td>Code location data row(s)</td>
<td>Review related code locations:</td>
</tr>
<tr>
<td></td>
<td>• ID - Code location identifier</td>
</tr>
<tr>
<td></td>
<td>• Description - What happens at this code location.</td>
</tr>
<tr>
<td></td>
<td>• Source - The source file associated with this code location.</td>
</tr>
<tr>
<td></td>
<td>• Function - Function name.</td>
</tr>
<tr>
<td></td>
<td>• Modules - The executable associated with this problem.</td>
</tr>
<tr>
<td></td>
<td>• State - Indicates whether the problem has been fixed or not. To change the state, use the context menu.</td>
</tr>
</tbody>
</table>

- Icon, icon, or no icon in the **Source column**
  Shows:
  - Whether this is a related code location.
  - If code location source code is available for viewing and editing.

- Icon, icon, or no icon in the **Source column**
  Shows:
  - Whether this is the focus code location.
  - If code location source code is available for viewing and editing.

- Icon, icon, or no icon in the **Source column**
  Shows if code location source code is available for viewing and editing.

- Column labels
  Click a column heading to sort the data grid rows in either ascending or descending order.

- Pane border
  Resize the pane (drag).

- Right click a row to display a context menu
  Display a context menu to: set this code location as the focus or related code location, copy the selected data row(s) to the clipboard, mark the state as fixed or not fixed, or display context-sensitive help.

**Focus Code Location Pane**
Pane Purpose and Usage | Pane Location | Pane Controls | Data Reference

**Focus Code Location Pane Purpose and Usage**
Use this pane to explore source code associated with focus code location in the **Dependencies Source** window.

**Focus Code Location Pane Location**
Top left of Dependencies Source window
Focus Code Location Pane Controls

<table>
<thead>
<tr>
<th>Use This</th>
<th>To Do This</th>
</tr>
</thead>
</table>
| ☐ icon, ☐ icon, or no icon in the Source column | Shows:  
• Whether this is the focus code location.  
• If code location source code is available for viewing and editing. |
| Pane border | Resize the pane (drag). |
| Source code | • Explore source code associated with the focus code location  
• Display the code editor at the corresponding source file by double-clicking a data row or by using the corresponding context menu item. |
| Right click a row to display a context menu | Display a context menu to: open the code editor to the corresponding source line, copy the selected data row(s) to the clipboard, or display context-sensitive help. |

Focus Code Location Call Stack Pane

Pane Purpose and Usage | Pane Location | Pane Controls | Data Reference

Focus Code Location Call Stack Pane Purpose and Usage
Use this pane to select which source code appears in the Focus Code Location pane in the Dependencies Source window.

Focus Code Location Call Stack Pane Location
Top right of Dependencies Source window

Focus Code Location Call Stack Pane Controls

<table>
<thead>
<tr>
<th>Use This</th>
<th>To Do This</th>
</tr>
</thead>
</table>
| ☐ or ☐ icon | View whether:  
• Source code is available for viewing and editing. An ☐ icon indicates that source code is not available. |
| Click a row in the Call Stack pane | Displays source code for the specified call stack entry. |
| Pane border | Resize the pane (drag). |
| Right click a row in the Call Stack pane | Customize the call stack presentation by using the Call Stack context menu. |

Related Code Locations Pane

Pane Purpose and Usage | Pane Location | Pane Controls | Data Reference

Related Code Locations Pane Purpose and Usage
Use this pane to explore source code associated with related code locations (related to the focus code location) in the Dependencies Source window.

Related Code Locations Pane Location
Middle left of Dependencies Source window
Related Code Locations Pane Controls

<table>
<thead>
<tr>
<th>Use This</th>
<th>To Do This</th>
</tr>
</thead>
</table>
| 📑 icon, 📒 icon, or no icon in the Source column | View:  
  • Whether this is a related code location.  
  • If code location source code is available for viewing and editing. |
| Pane border | Resize the pane (drag). |
| Source code | • Explore source code associated with the focus code location  
  • Display the code editor at the corresponding source file by double-clicking a data row or by using the corresponding context menu item. |
| Right click a line to display a context menu | Display a context menu to: open the code editor to the corresponding source line, copy the selected data row(s) to the clipboard, or display context-sensitive help. |

Related Code Location Call Stack Pane

Pane Purpose and Usage | Pane Location | Pane Controls | Data Reference

Related Code Location Call Stack Pane Purpose and Usage
Use this pane to select which source code appears in the Related Code Location pane.

Related Code Location Call Stack Pane Location
Middle right of Dependencies Source window

Related Code Location Call Stack Pane Controls

<table>
<thead>
<tr>
<th>Use This</th>
<th>To Do This</th>
</tr>
</thead>
</table>
| 📑 or 📒 icon | View whether:  
  • Source code is available for viewing and editing. An 📑 icon indicates that source code is not available. |
| Click a row in the Call Stack pane | Displays source code for the specified call stack entry. |
| Pane border | Resize the pane (drag). |
| Right click a row in the Call Stack pane | Customize the all stack presentation by using the Call Stack context menu. |

Relationship Diagram Pane

Pane Purpose and Usage | Pane Location | Pane Controls | Data Reference

Relationship Diagram Pane Purpose and Usage
Use this pane to view the relationships among code locations for the selected problem.

Relationship Diagram Pane Location
Bottom right of Dependencies Source window
Relationship Diagram Pane Controls

<table>
<thead>
<tr>
<th>Use This</th>
<th>To Do This</th>
</tr>
</thead>
<tbody>
<tr>
<td>Title bar</td>
<td>View the problem type.</td>
</tr>
</tbody>
</table>
| 📈 icon, 📊 icon, or no icon in the Source column | View:  
• Whether this is a related code location.  
• If code location source code is available for viewing and editing. |
| 📈 icon, 📊 icon, or no icon in the Source column | View:  
• Whether this is the focus code location.  
• If code location source code is available for viewing and editing. |
| 📈 icon, 📊 icon, or no icon in the Source column | View if code location source code is available for viewing and editing. |
| Pane border | Resize the pane (drag). |
| Diagram | View the relationship among code locations in a problem:  
• Each box in a diagram represents a code location in a problem.  
• A diagram with a single box is a trivial problem with no related code locations.  
• Boxes arranged left-to-right with connecting arrows indicate a time ordering.  
• Boxes with connecting lines indicate association. |

Memory Access Patterns Analysis

Purpose and Usage | Prerequisites | Run | Controls | Data Reference

Memory Access Patterns Analysis Purpose and Usage
View information about types of memory access inside selected loops/functions.

Memory Access Patterns Analysis Prerequisites

To prepare to run a Memory Access Patterns analysis:

1. Do one of the following to open the Analysis Target tab in the Project Properties dialog box:
   • In the Intel Advisor GUI, choose Project > Project Properties...
   • In the Visual Studio IDE, choose Project > Intel Advisor [version] Project Properties...
2. Choose the Memory Access Patterns Analysis type, then specify the same application as in the Survey Hotspots Analysis type, but a smaller input data set if possible.
4. Mark one or more loops for deeper analysis by enabling the corresponding 🗳 checkboxes in the Survey Report.
Tip
- If possible, use the Inherit settings from Survey Hotspots Analysis Type checkbox.
- The Memory Access Patterns Analysis type consumes more resources than the Survey Hotspots Analysis type. If analysis takes too long, consider decreasing the workload.
- Setting search directories in the Binary/Symbol Search tab and Source Search tabs is optional for the Vectorization Advisor.
- Marking loops for deeper analysis using the checkbox is a Vectorization Advisor-specific alternative to adding annotations in target application source code.
- Memory Access Patterns analysis is time intensive. The fewer loops you mark for deeper analysis, the less time it takes to collect the data.
- Choose the optimal compiler settings. The most up-to-date settings are in User Guide: Before You Begin.

Run a Memory Access Patterns Analysis
To run a Memory Access Patterns Report, do one of the following:
- Click the Collect button under Check Memory Access Patterns in the Workflow tab.
- In the Visual Studio Solution Explorer, right-click the project, then choose Intel Advisor [version] > Start Memory Access Patterns Analysis.

While analysis is running, you can do the following under the appropriate analysis step in the Workflow tab
- Stop analysis and data collection, and retain the already collected data: Click the button.
- Cancel analysis and data collection, and discard the collected data: Click the button.
- Interrupt post-collection data finalization: Click the button.

Memory Access Patterns Report Controls
In the Memory Analysis Patterns Report tab at the bottom of the Refinement Report:

<table>
<thead>
<tr>
<th>Use This</th>
<th>To Do This</th>
</tr>
</thead>
<tbody>
<tr>
<td>Double-click any line</td>
<td>View the selected operation’s source code.</td>
</tr>
</tbody>
</table>

Associated Memory Access Patterns Source window, from top left to bottom right:
- View Activation pane - Enable or disable views shown in the Source view.
- Source View pane - View source code of the selected loop/function.
- Assembly View pane - View assembly source of the selected loop/function.
- Details View pane - View details of the selected site.

Memory Access Patterns Source Window
Window Purpose and Usage | Window Access | Window Regions | Data Reference

Memory Access Patterns Source Window Purpose and Usage
Use this window to examine the source code for a selected site.

Memory Access Patterns Source Window Access
To access this window in the Memory Access Patterns tab under the Refinement Reports, double-click a data row.
Memory Access Patterns Source Window Regions and Usage

When using the Intel Advisor GUI on Linux* OS, the editor defined by the Options > Editor dialog box appears with the file open at the corresponding location. When using the Intel Advisor GUI on Windows* OS, the file type association (or Open With dialog box) determines the editor used. When using Visual Studio*, the Visual Studio code editor appears with that file open at the corresponding location.

From top left to bottom right:
- View Activation pane - Enable or disable views shown in the Source view.
- Source View pane - View source code of the selected loop/function.
- Assembly View pane - View assembly source of the selected loop/function.
- Details View pane - View details of the selected site.

Memory Access Patterns Source Window Controls

<table>
<thead>
<tr>
<th>Use This</th>
<th>To Do This</th>
</tr>
</thead>
<tbody>
<tr>
<td>Source view</td>
<td>View user-visible source code representation of the selected loop.</td>
</tr>
<tr>
<td>Assembly view</td>
<td>View assembly representation of the selected loop's code.</td>
</tr>
<tr>
<td>Details</td>
<td>View details for the selected code region.</td>
</tr>
</tbody>
</table>

Assembly View Pane (Memory Access Patterns Source Window)

Pane Purpose and Usage | Pane Location | Pane Controls | Data Reference
-----------------------|---------------|---------------|-------------------
Assembly View Pane Purpose and Usage

View assembly source of the selected loop/function.

Assembly View Pane Location

Bottom of Memory Access Patterns Source window

Assembly View Pane Controls

<table>
<thead>
<tr>
<th>Use This</th>
<th>To Do This</th>
</tr>
</thead>
<tbody>
<tr>
<td>Source lines</td>
<td>You can navigate to related source lines or explore assembly representation of the code by using the Call Stack with Loops pane.</td>
</tr>
<tr>
<td>Select multiple source lines</td>
<td>To view the accumulated time values for multiple source lines below the Self Time column, or enable you to copy multiple source lines using the context menu. Viewing accumulated time can help you decide how to divide the work.</td>
</tr>
</tbody>
</table>

Details View Pane (Memory Access Patterns Source Window)

Pane Purpose and Usage | Pane Location | Pane Controls | Data Reference
-----------------------|---------------|---------------|-------------------
Details View Pane Purpose and Usage

View details of the selected site.

Details View Pane Location

Bottom right of Memory Access Patterns Source window
Details View Pane Controls

<table>
<thead>
<tr>
<th>Use This</th>
<th>To Do This</th>
</tr>
</thead>
<tbody>
<tr>
<td>&gt;</td>
<td>Expand list of variable references</td>
</tr>
</tbody>
</table>

Source View Pane (Memory Access Patterns Source Window)

Pane Purpose and Usage | Pane Location | Pane Controls | Data Reference

Source View Pane Purpose and Usage

View source code of the selected loop/function.

Source View Pane Location

Middle of Memory Access Patterns Source window

Source View Pane Controls

<table>
<thead>
<tr>
<th>Use This</th>
<th>To Do This</th>
</tr>
</thead>
<tbody>
<tr>
<td>Source lines</td>
<td>To navigate to related source lines.</td>
</tr>
<tr>
<td>Double-click a source line</td>
<td>To open your code editor to the corresponding source file. The editor allows you to add annotations to your code (right-click to open the context menu).</td>
</tr>
<tr>
<td></td>
<td>• On Windows* OS:</td>
</tr>
<tr>
<td></td>
<td>• When using Microsoft Visual Studio*, the Visual Studio code editor</td>
</tr>
<tr>
<td></td>
<td>appears with the file open at the corresponding location.</td>
</tr>
<tr>
<td></td>
<td>• When using the Intel Advisor GUI, the file type association (or Open With dialog box) determines the editor used.</td>
</tr>
<tr>
<td></td>
<td>• On Linux* OS:</td>
</tr>
<tr>
<td></td>
<td>When using the Intel Advisor GUI, the editor defined by the Options &gt; Editor dialog box appears with the file open at the corresponding location.</td>
</tr>
<tr>
<td>Select multiple source lines</td>
<td>To view the accumulated details in the Details View pane.</td>
</tr>
<tr>
<td>Right click a source line or multiple source lines</td>
<td>Display a context menu to: open your code editor to the corresponding source line, copy the selected source line(s) to the clipboard, or display context-sensitive help relevant to the selected loop or function.</td>
</tr>
</tbody>
</table>

View Activation Pane (Memory Access Patterns Source Window)

Pane Purpose and Usage | Pane Location | Pane Controls | Data Reference

View Activation Pane Purpose and Usage

Enable or disable views shown in the Source view.

View Activation Pane Location

Top of Memory Access Patterns Source window
View Activation Pane Controls

<table>
<thead>
<tr>
<th>Use This</th>
<th>To Do This</th>
</tr>
</thead>
<tbody>
<tr>
<td>Source button</td>
<td>Display source code of the selected loop/function in the source code pane.</td>
</tr>
<tr>
<td>Assembly button</td>
<td>Display assembly code of the selected loop/function in the assembly source pane.</td>
</tr>
<tr>
<td>Details button</td>
<td>Display details pane.</td>
</tr>
</tbody>
</table>

Suitability Analysis

The Intel® Advisor Suitability tool runs your annotated serial program's executable (target). It measures the target executable and uses your annotated parallel sites to predict (estimate) its approximate parallel performance characteristics.

**NOTE** The Suitability tool is available in the Threading Workflow only. Use the button to switch to the Threading Workflow.

Use data collected by the Suitability tool to understand:
- The likely performance gains from parallelism based on the inserted annotations and the specified number of available cores.
- The potential effects of parallel overhead, including which parallel frameworks are worth investigating.
- The impact of changing the number of iterations and the iteration duration on performance of the selected site.

Running the Suitability Tool

You can run the Suitability tool:
- Within Microsoft Visual Studio*. Use the Tools menu, Advisor Workflow tab, or the toolbar icons (see Using the Menu Items and Toolbars).
- Within the Intel Advisor standalone GUI. Use the Tools menu, the Advisor Workflow tab, or the toolbar icons after you open a project.
- Using the advixe-cl command line. Specify the --collect option after you open a project.

Viewing the Suitability Report

Similarly, you can view the collected data in the Suitability Report and related windows by using Visual Studio, the standalone GUI, or the command line.

Why do I need to Collect the Suitability Data?

Use the Suitability tool to help you minimize the effort you spend locating areas where parallelism may not be worthwhile, such as creating a parallel site whose parallel overhead would outweigh any performance gains. As you refine and adjust your parallel site and task annotations, run this tool again to see how these changes impact the predicted performance characteristics. Later, after you fix data sharing problems, run the Suitability tool again to view the modified program's predicted performance.
The Suitability tool eliminates the need for you to manually time and/or calculate such items as the total parallel site time, total task time for each task, average number of tasks per site, the number of task executions, and the expected maximum gain. It does the measurements and calculations for you, providing you with predicted summary data, such as how much CPU time each parallel site uses and the site maximum gain.

The maximum possible gain for a perfectly parallelized program on a 2-core system is 2.0, so the program would run in half the time. Possible gains on a system with \(N\) cores might be any of the following:

- Typical values are from 1.0 to \(N\), where 1.0 means the parallel program only runs as fast as the serial program, and \(N\) is the ideal case of perfect parallelism.
- Values less than 1.0 indicate that the parallel program spends more time doing parallel overhead work than it gains from parallelism. That is, the serial program runs faster than the parallel program.

**What Else do I need to Know About the Suitability Tool?**

When you run the Suitability tool, your program will take longer to run than normal because the Suitability tool needs to sample and measure the characteristics of your program's parallel sites and tasks. The analyzed data appears in the **Suitability Report** window. Use this window to view the predicted performance for the annotated parallel sites and their tasks. You can change the mathematical modeling parameters - such as the CPU count - and view the impact of these changes on the site's maximum gain (Site Gain) performance estimate and a site's contribution to the program's Maximum Program Gain for All Sites.

**NOTE**

The Suitability tool *estimates* the behavior of your parallel program using the observed behavior of your annotated serial program. It does not consider specific processor characteristics. Thus, although it can provide approximate predictions to help you choose parallel code regions, it cannot provide exact predictions of your parallel program. After you have modified your program to run in parallel, use Intel® VTune™ Profiler to obtain more accurate performance data as well as tune your parallel program. Because Intel VTune Profiler observes the program running in parallel, it can provide more accurate performance information than the Suitability tool's estimates.

**Annotating Code for Deeper Analysis**

Before you can *mark* the best parallel opportunities by adding Intel® Advisor annotations, you need to *choose* likely places to add parallelism. This section provides a series of topics that explain factors to consider as you examine the candidate code regions and their execution and choose candidate places.

The operations of a serial program execute one after another in a well-defined order, starting at the beginning, continuing to the end, and then stopping. A parallel program, on the other hand, is made up of tasks - portions of the program that may execute independently on separate cores. Tasks can either be implemented in separate functions or in iterations of a loop.

You mark your proposed code regions by adding Intel® Advisor annotations that identify the:

- Parallel site: A code region that contains one or more parallel tasks. Execution of a parallel site constrains the time during which the tasks that it contains can execute. Although execution of a parallel site begins when its execution reaches its beginning, its execution terminates only after all tasks that started within it have completed. In parallel frameworks, this corresponds to the *join* location in the code where all tasks have completed.
- Parallel tasks: Task code regions run independently, at the same time as other tasks within the parallel site and the enclosing parallel site itself. Also, each task can have multiple instances of its code executing. As shown in the table below, there are two forms of task annotations:
  - For a loop with only a single task, add a single iteration task annotation within the two site annotations.
For other code, add a task annotation pair to mark the task region's begin and end within the two site annotations.

<table>
<thead>
<tr>
<th>Characteristics of Parallel Site Code</th>
<th>Parallel Site and Task Annotations</th>
<th>Comments and Limitations</th>
</tr>
</thead>
</table>
| A loop that requires only a single task. For simple loops, begin with the type of task annotation, unless the task does not include the entire loop body. | Add three annotations to mark:  
- The parallel site region by adding site begin and site end annotations.  
- The parallel task loop by adding a single iteration task annotation at the start of the loop body. | Based on the Suitability tool performance predictions, you may want to try using multiple tasks. In this case, remove the single iteration task annotation and replace it with task begin and task end annotations for each task (see the next row).  
If the loop structure is complex, you may need to mark the task begin and task end region by using the task annotations in the next row. |
| Example code: nqueens_Advisor C/C++ sample and nqueens Fortran and C# samples | | |
| Complex loop, code that allows multiple tasks, or non-loop code | Add four annotations to mark:  
- The parallel site region by adding site begin and site end annotations.  
- Each parallel task region by adding task begin and task end annotations. | |
| Example code: stats C++ sample | | |

After you choose several places to add parallelism, view the data displayed in the Survey Report window. Use this data and your code editor to add annotations to mark the candidate parallel sites and their task(s). Make sure that these annotations are executed by the selected target executable.

The site and task annotations enable the Intel® Advisor Suitability and Dependencies tools to predict your serial program's execution as a parallel program. These tools perform extensive analysis of your running serial program to provide data needed to help you decide the best place(s) to add parallelism.

To take advantage of the Intel® Advisor parallel design capabilities, experiment with different possible parallel code regions by modifying the site and task annotations and their locations, rebuilding your application's target, and running the Suitability and Dependencies tools again.

The following figure illustrates the nqueens_Advisor C/C++ sample code to show the task (blue background) and its enclosing parallel site (orange background).
Before you convert your serial program into a parallel program, you need to:

- Understand where your program is spending its time.
- Decide how to divide that work up into tasks that can execute in parallel.

**Before Annotating Code for Deeper Analysis**

Before you can mark the best parallel opportunities by adding annotations, you need to choose likely places to add parallelism. This section introduces several topics that explain factors you should consider as you closely examine the candidate code regions and their execution.

Each code region where you might add parallelism consists of a single parallel site and one or more parallel tasks enclosed within the parallel site. Each parallel site defines the scope of parallel execution. You can have multiple parallel sites in a program.

No matter how much you improve one part of your program, the program cannot complete any faster than the part that you did not speed up. So, focus your efforts on the parts of your program that use the most time.

Use the **Survey Report** provided by the Survey tool to help you understand where your program spends its time.

**Using Amdahl's Law and Measuring the Program**

There are two rules of optimization that apply to parallel programming:

- Focus on the part of the program that uses the most time.
- Do not guess, measure.

**Amdahl's Law**

In the context of parallel programming, Gene Amdahl formalized a rule called Amdahl's Law, which states that the speed-up that is possible from parallelizing one part of a program is limited by the portion of the program that still runs serially.
The consequence may be surprising: parallelizing the part of your program where it spends 80% of its time cannot speed it up by more than a factor of five, no matter how many cores you run it on.

Therefore, to get maximum benefit from parallelizing your program, you could add parallelism to all parts of your program as suggested by Amdahl's Law. However, it is more practical to find where it spends most of its time and focus on areas that can provide the most benefit.

**Do Not Guess - Measure**

This leads to another rule of optimization: *Do Not guess - Measure*. Programmers’ intuitions about where their programs are spending time are notoriously inaccurate. Intel® Advisor includes a Survey tool you can use to profile your running program and measure where it spends it time.

After you add Intel® Advisor annotations to your program to mark the proposed parallel code regions, run the Suitability tool to predict the approximate maximum performance gain for the program and the annotated sites. These estimated performance gain values are based on a model of parallel execution that reflects the impact of Amdahl's law.

**See Also**

- Task Organization and Annotations
- Intel Advisor Workflow Tab
- Intel Advisor Menu Items and Toolbars

**Task Organization and Annotations**

You will choose a region of code to execute as a task. This region is the *static extent* of the task. The task includes not just its static extent, but also any other code that is called from the static extent when it executes - this is the *dynamic extent*.

In addition to choosing tasks, you will also decide which tasks can execute in parallel with one another. To do this, you will choose *parallel sites*. A parallel site, like a task, has a static extent which is a block of code and a dynamic extent which includes all the code that is called from it.

**NOTE**

If you have a loop with a single task and the task includes the entire loop body, you can use the simplified parallel site with one iteration task annotation. The remainder of this topic and this group of topics describe the more complex case where multiple tasks are needed within a parallel site.

The execution of tasks with the serial execution done by Intel® Advisor works like this:

1. A parallel site begins when execution reaches the begin-site annotation.
2. A task is created when execution reaches the begin-task annotation. The task executes independently, in parallel with any other tasks that are already executing, including the parallel site itself.
3. When the execution of a task reaches an end-task annotation, the task terminates. Intel® Advisor end-task annotations do not allow or require an end-task label, so be aware that in some cases the task’s execution could reach a task-end annotation for a different task, which can impact the predicted parallel performance.
4. When execution reaches the end-site annotation for the parallel site, Intel® Advisor predicts that execution suspends (waits) until all tasks that were created within it have terminated, after which execution exits the parallel site.

With C/C++ code, note that goto, break, continue, return, and throw statements must not bypass the end of the static extent of a task or parallel site! With Fortran code, such statements include goto and return. You may need to add extra *end* annotations before these operations so the Intel® Advisor tools will correctly model the end of a site or task.
Because you will later add parallel framework code after you no longer need the Intel® Advisor annotations, you need to be aware of the requirements of the parallel framework. For example, some parallel frameworks might not allow a branch out of a task, such as a loop task. Whenever possible, plan your tasks to suit the needs of the parallel framework code. The annotations are present only while you need Intel® Advisor to help you predict the proposed parallel behavior and make decisions about the best locations for your tasks.

After you decide where the parallel sites and tasks are in your program, add source annotations.

**See Also**
- Annotating Parallel Sites and Tasks
- Site and Task Annotations for Simple Loops with One Task
- Copying Annotations and Build Settings Using the Annotation Assistant Pane

**Annotating Parallel Sites and Tasks**

You add annotations into your program to mark the tasks and parallel sites. The annotations are one-line macro uses or function calls that have no effect on the behavior of your program.

Annotations allow you to mark your tentative decisions about your program’s task structure before you modify the program to use parallel execution. Annotations are used by the Intel® Advisor Suitability and Dependencies tools.

After you decide on the parallel site(s) and task(s), add the annotations into your source code.

To simplify adding Intel® Advisor annotations:
- When using the Microsoft Visual Studio® code editor, you can use the Annotation Wizard.
- With any editor, use the annotation assistant in the Survey Report window, Survey Source window, or the No Data message to copy example annotation code and build settings.

Code examples throughout this group of topics illustrate the use of these annotations.

As you use Intel® Advisor to investigate possible code regions for adding parallel execution, you will find some areas are not feasible. Adding a comment to explain why that site (or task) was not chosen may help later. For example, with C/C++ code:

```c++
...  // Investigated the following function call as a parallel task and dismissed  // June 2014. Need to first re-write the function to improve parallel  // performance and fix the data race.  //  // ANNOTATE_TASK_BEGIN(func1);
```

**See Also**
- Task Patterns
- Intel Advisor Annotation Definitions File
- Annotation Types Summary
- Copying Annotations and Build Settings Using the Annotation Assistant Pane
- Adding Annotations Using the Annotation Wizard
- Add Parallelism

**Task Patterns**

To summarize:
- You choose _parallel sites_ in your program.
- You choose _tasks_ in your parallel sites.
- Tasks in a parallel site can execute in parallel with one another and with tasks in an outer parallel site, but not in parallel with tasks in unrelated parallel sites.
You are free to arrange your sites and tasks any way that you want, but there are several simple, common patterns that you will probably want to use.

The following sections describe the process of identifying task patterns, as well as information about data parallelism and task parallelism.

**Multiple Parallel Sites**

You may be able to introduce parallelism independently in more than one place in a program.

For example, consider a C/C++ program with the general structure:

```c
initialize(data);
while (!done) {
    display_on_screen(data);
    update(data);
}
```

You might be able to parallelize the display and update operations independently:

```c
display_on_screen(data)
{
    ANNOTATE_SITE_BEGIN(site_display);
    for (each block of data) {
        ANNOTATE_ITERATION_TASK(task_display);
        display the block of data;
    }
    ANNOTATE_SITE_END();
}
update(data)
{
    ANNOTATE_SITE_BEGIN(site_update);
    for (each block of data) {
        ANNOTATE_ITERATION_TASK(task_update);
        update the block of data;
    }
    ANNOTATE_SITE_END();
}
```

Each iteration of the main loop would still do the display and then the update, but the display and update operations could be performed much faster.

Depending on your program, you need to decide whether to implement multiple parallel sites at the same or at different times:

- When two parallel sites are truly disjoint or have overlapping functions that are purely functional and do not show problems reported by the Dependencies tool, you can consider parallelizing those sites separately at different times.
- When considering multiple parallel sites that overlap on the same call trees - such as multiple sites that call the same (common) utility functions - consider parallelizing or not parallelizing the entire set of parallel sites at the same time.

You need to determine the cause of each dependency and fix it. If you have multiple parallel sites that overlap on the same call trees - such as multiple sites that call the same utility functions (common code) - read the help topic Fixing Problems in Code Used by Multiple Parallel Sites.

**See Also**

- Data and Task Parallelism
- Using Partially Parallel Programs with Intel Advisor Tools
- Data Sharing Problems
- Fixing Problems in Code Used by Multiple Parallel Sites
Data and Task Parallelism

This topic describes two fundamental types of program execution - data parallelism and task parallelism - and the task patterns of each.

Data Parallelism

In many programs, most of the work is done processing items in a collection of data, often in a loop. The data parallelism pattern is designed for this situation. The idea is to process each data item or a subset of the data items in separate task instances. In general, the parallel site contains the code that invokes the processing of each data item, and the processing is done in a task.

In the most common version of this pattern, the serial program has a loop that iterates over the data items, and the loop body processes each item in turn. The data parallelism pattern makes the whole loop a parallel site, and the loop body is a task. Consider this C/C++ simple loop:

```c
ANNOTATE_SITE_BEGIN(sitename);
for (int I = 0; I != n; ++I) {
    ANNOTATE_ITERATION_TASK(task_process);
    process(a[I]);
}
ANNOTATE_SITE_END();
```

The following C/C++ code shows a situation where the data items to be processed are in the nodes of a tree. The recursive tree walk is part of the serial execution of the parallel site - only the `process_node` calls are executed in separate tasks.

```c
ANNOTATE_SITE_BEGIN(sitename);
process_subtree(root);
ANNOTATE_SITE_END(sitename);
...
void process_subtree(node) // in the dynamic extent of the parallel site
{
    ANNOTATE_TASK_BEGIN(task_process);
    process_node(node);
    ANNOTATE_TASK_END();
    for (child = first_child(node);
        child;
        child = next_child(child) )
    {
        process_subtree(child);
    }
}
```

In the data parallelism pattern, the parallel site usually contains a single task.

The sample `tachyon_Advisor` demonstrates data parallelism.

Task Parallelism

When work is divided into several activities which you cannot parallelize individually, you may be able to take advantage of the task parallelism pattern.

**NOTE**

The word *task* in *task parallelism* is used in the general sense of an activity or job. It is just a coincidence that we use the same word to refer to "a body of code that is executed independently of other bodies of code".
In this pattern, you have multiple distinct task bodies in a parallel site performing different activities at the same time.

Suppose that neither the display nor the update operation from the previous example can be parallelized individually. You still might be able to do the display and the update simultaneously. Consider this C/C++ code:

```
initialize(data);
while (!done) {
    old_data = data;
    ANNOTATE_SITE_BEGIN(sitename);
    ANNOTATE_TASK_BEGIN(task_display);
    display_on_screen(old_data);
    ANNOTATE_TASK_END();
    ANNOTATE_TASK_BEGIN(task_updatedata);
    update(data);
    ANNOTATE_TASK_END();
    ANNOTATE_SITE_END();
}
```

The most obvious shortcoming of the task-parallel pattern is that it cannot take advantage of more cores than the number of distinct tasks. In this example, any more than two cores would be wasted. On the other hand, the task parallel pattern may be applicable to programs that simply do not fit the data parallel pattern - some parallelism may be better than none.

The tasks used in task parallelism are not limited to called functions. For example, consider this C/C++ code that creates two tasks that separately increment variables $X$ and $Y$:

```
main() {
    ANNOTATE_SITE_BEGIN(sitename);
    ANNOTATE_TASK_BEGIN(task_x);
    X++;
    ANNOTATE_TASK_END();

    ANNOTATE_TASK_BEGIN(task_y);
    Y++;
    ANNOTATE_TASK_END();
    ANNOTATE_SITE_END();
}
```

The sample stats demonstrates task parallelism.

See Also
Mixing and Matching Tasks
Annotations

Mixing and Matching Tasks
You can combine the data parallel and task parallel patterns. Continuing with the display/update example, suppose that you can parallelize the update operation, but not the display operation. Then you could execute the display operation in parallel with multiple tasks from the update operation. Consider this C/C++ code:

```
initialize(data);
while (!done) {
    old_data = data;
    ANNOTATE_SITE_BEGIN(sitename);
    ANNOTATE_TASK_BEGIN(task_display);
    display_on_screen(old_data);
    ANNOTATE_TASK_END();
    update(data);
    ANNOTATE_SITE_END();
```
display_on_screen(data) {
    ... 
}
update(data) {
    for (each block of data) {
        ANNOTATE_TASK_BEGIN(task_update);
        update the block of data;
        ANNOTATE_TASK_END();
    }
}

See Also
Choosing the Tasks
Annotations

Choosing the Tasks
When choosing tasks, you should consider task interactions and the factors that influence how large a task should be. The following sections describe the process of choosing the tasks.

Task Interactions and Suitability
If your tasks access the same memory locations, then, left to themselves, they will tend to trip over each other. You can solve this by adding synchronization code to make sure the tasks are well-behaved when they access shared memory locations, but synchronization code can be tedious to add and hard to get right, and it is easy to end up with tasks that spend more time doing synchronization than doing work.

You can use the Suitability tool to provide performance data that helps you choose your tasks wisely.

It is better to minimize data access conflicts in the first place by choosing your tasks wisely. It can be hard to tell, just by looking at your code, where all the sharing problems will be, which is why you will learn how to automate the process by using the Dependencies tool.

However, you can make a good guess whether two proposed tasks are mostly independent of each other or are completely intertwined.

See Also
How Big Should a Task Be?
Suitability Analysis
Dependencies Analysis

How Big Should a Task Be?
The ideal task size is very dependent on the details of your program. Here are a few general considerations to keep in mind.

Task Overhead
In general, if your program can keep most of the cores on your system busy doing useful work, then it will be using the system about as efficiently as possible. There are two parts to this: keeping the cores busy, and doing useful work.

It takes time to start a new task. If your tasks are too small, then your program may spend more time creating tasks than it saves by running them in parallel - the cores are kept busy, but not doing useful work.
Load Balance
On the other hand, very large tasks can reduce parallelism: your parallel program cannot finish any more quickly than the longest-running task. A rule of thumb is to try to have the number of tasks in a parallel site be at least several times larger than the number of cores available, so that there will always be some work to do when a core is free.

Choosing the Right Level
You will often have the opportunity to create tasks at different loop nesting levels or function call depths. This may provide an easy way to choose your task size. For example, consider the C/C++ code:

```c
for (i = 0; i != N; ++i) {
    for (j = 0; j != N; ++j) {
        x[i, j] = y[i, j] * z[j, i];
    }
}
```

The inner loop body is too small to be a useful task. You can view the Suitability Report for a task's Average Instance Time. The entire inner loop might be more suitable:

```c
ANNOTATE_SITE_BEGIN(sitename);
for (i = 0; i < N; ++i) {
    ANNOTATE_ITERATION_TASK(task_process_array);
    for (j = 0; j < N; ++j) {
        x[i, j] = y[i, j] * z[j, i];
    }
}
ANNOTATE_SITE_END();
```

Blocking
If you have a loop which seems like an obvious place to introduce parallelism, but the loop body is too small to make a good task, consider grouping several iterations together. When you specify a loop body as a parallel construct, Intel® Threading Building Blocks and OpenMP® will automatically group multiple loop iterations together to create tasks of an appropriate size. Therefore, given a simple loop, the question is not whether the loop body is the right size for a good task, but whether the total loop execution time can be divided up into chunks of the right size.

For example, there is only one loop level here, and its body looks too small to be a good task:

```c
for (i = 0; i < 100000; ++i) {
    a[i] = b[i] * c[i];
}
```

Go ahead and choose it, and it may run as though you had written it as:

```c
ANNOTATE_SITE_BEGIN(sitename);
for (i = 0; i < 100000; i += 1000) {
    ANNOTATE_ITERATION_TASK(task_calculate_a);
    for (j = i; j < i + 1000; ++j) {
        a[j] = b[j] * c[j];
    }
}
ANNOTATE_SITE_END();
```

Sizing to Avoid Interactions
It is not uncommon for loop iterations or other potential task bodies to be almost independent at one level, but have many interactions at other levels. In this case, it may be worth accepting a less than perfect program gain in exchange for simpler programming and cleaner code.
The outer loop of the Sudoku problem generator repeatedly calls the `generate()` function to generate problems. There are opportunities for introducing parallelism at many different levels in the problem generation function, but the individual calls to `generate()` are almost perfectly independent, and each call to `generate()` takes less than a second. Parallelizing the outermost loop would be a trivial project. No user is likely to care if it takes 0.8 seconds instead of 0.2 seconds to generate a single problem, and the speedup for generating more than a handful of problems should be nearly perfect.

**Using the Survey Report**

Ultimately, choosing your tasks is more of an art than a science. Locations close to the root of the call tree tend to form larger tasks, but may have more conflicts on shared variables; locations toward the leaves of the call tree tend to be smaller, causing problems with task overhead, but typically have fewer conflicts. We can offer some rules of thumb. Start by looking at a function $F$ that uses a significant portion of the time of the program part you are trying to improve - remember Amdahl's law!

- If almost all of the time spent in $F$ is spent in a block of code that is executed many times in a loop, then that block of code may be a prime candidate for a data-parallel task.
- If $F$ is basically just a wrapper around a call to a function $G$, then look at $G$ instead.
- If almost all of the time in $F$ is spent in multiple calls to a function $G$ that is too large to be a good task, then you may want to enclose the calls to $G$ in a parallel site, but introduce the actual tasks inside $G$ or another function that is called from $G$.
- If the time spent in $F$ is distributed across a number of distinct activities, you should consider whether it is better to apply the task parallelism pattern to $F$, or to use the multiple parallel sites pattern to look for parallelism in each of the activities.

**Recursion**

Recursive algorithms can present a special challenge. The problem occurs when you have a large amount of time spent in a function that only does a small amount of work in any one invocation, but that is called recursively a great many times. The actual work may be data-parallel, but the function body is too small to be a useful task by itself, and the blocking strategy (see Blocking above) is harder to apply to a recursive algorithm.

The general solution is to use a threshold to control recursive parallelism. For example, a recursive sort might solve sub-problems in parallel only if they are above a certain threshold size.

**See Also**

Using Partially Parallel Programs with Intel Advisor Tools
Data and Task Parallelism

Using Partially Parallel Programs with Intel® Advisor Tools

Intel® Advisor tools are designed to collect data and analyze *serial* programs. If you have a partially parallel program, *before* you use the Intel® Advisor Suitability and Dependencies tools to examine it to add more parallelism, read the guidelines in this topic and modify your program so it runs as a serial program with a single thread within each parallel site.

**Run Your Program as a Serial Program**

To run the current version of your program as a serial program, you need to limit the number of threads to 1. To run your program with a single thread:
With Intel® Threading Building Blocks (Intel® TBB), in the main thread create a

```cpp
tbb::task_scheduler_init init(1);
// ...rest of program...
return 0;
```

The effect of `tbb::task_scheduler_init` applies separately to each user-created thread. So if the program creates threads elsewhere, you need to create a `tbb::task_scheduler_init init(1);` for that thread's lifetime as well. Use of certain Intel TBB features can prevent the program from running serially. For more information, see the Intel TBB documentation.

With OpenMP*, do one of the following:

- Set the OpenMP* environment variable `OMP_NUM_THREADS` to 1 before you run the program.
- Omit the compiler option that enables recognition of OpenMP pragmas and directives. On Windows* OS, omit `/Qopenmp`, and on Linux* OS omit `-openmp`.

For more information, see your compiler documentation.

### Add or Remove Intel® Advisor Annotations

Intel® Advisor site, task, and lock annotations are used by the Suitability and Dependencies tools. You can add Intel® Advisor parallel site and task annotations to mark the already parallel code regions. For example, the `nqueens_Advisor` sample `nqueens_cilk.cpp`:

```cpp
... 
ANNOTATE_SITE_BEGIN(solve);
   cilk_for(int i=0; i<size; i++) {
      // try all positions in first row using separate array for each recursion
      ANNOTATE_ITERATION_TASK(setQueen);
      int * queens = new int[size];
      setQueen(queens, 0, i);
   }
ANNOTATE_SITE_END();
... 
```

If needed, you can comment out annotations, or add preprocessor directives by using conditional compilation. For example, use the `#ifdef`, `#ifndef`, and `#endif` preprocessor directives:

```cpp
... 
// Comment out the next line to hide the annotations.
#define ANNOTATE_ON
... 
#ifdef ANNOTATE_ON
    ANNOTATE_SITE_BEGIN(solve); 
#endif
#ifdef ANNOTATE_ON
    // add parallel code here 
#endif
#define ANNOTATE_ON
... 
```
After you add the parallel framework code and test it, you can remove the annotations.

**Effect of Parallel Code on Intel® Advisor Tools' Reports**

Because Intel® Advisor tools are designed to collect data and analyze serial program targets.

Parallel code that creates one or more threads within any annotated parallel site usually cause the Suitability or Dependencies tool reports to contain unreliable data. To use these two tools, there must be only a single thread within each parallel site. Also, when using parallel frameworks that use dynamic scheduling or work stealing at run-time, execution times can be assigned to the wrong source code.

If you use the Survey tool to profile your program, the Self Time in the Survey Report shows the sum of the CPU time for all threads. However, because Intel® Advisor's purpose is to analyze serial code, some of the time used by parallel code may be added to the wrong places. For example, Self Time may be added to the parallel framework run-time system entry points instead of the caller(s) in the thread that entered the parallel region. Also in the Survey Report, when examining parallel code, some entry points may be parallel framework run-time system entry points instead of the expected functions or loops. Similarly, in the Survey Source window, for a parallel code region the Total Time (and Loop Time) shows the sum of the CPU time for all threads.

Because Intel® Advisor's purpose is to analyze serial code, in the Suitability Report:

- Intel® Advisor assumes there is only a single thread (no parallelism) within any annotated parallel site, including its task(s) and lock(s). When only a single thread executes within a parallel site (as expected), the results for that site may be correct. If the application has multiple parallel sites, and one or more sites were executed by multiple threads, the next two items apply.
- If multiple threads execute within any parallel site, the reported Maximum Program Gain and that site's Impact on Program Gain values are not reliable. To obtain correct values, ensure that only a single thread executes for all parallel sites (see Run Your Program as a Serial Program above).
- If multiple threads execute within a parallel site, the results for that site will be unpredictable and its values will not be reliable. Also, if one thread executes the parallel site annotations and a second thread executes the task annotation(s), the site may appear to not have any tasks and the tasks may appear to not execute within a site. To obtain correct values, ensure that only a single thread executes within each parallel site (see Run Your Program as a Serial Program above).
- Any work-stealing constructs within the site will cause extra time to be added to the suspended site and/or task. All Suitability Report times are approximate.

Similarly in the Dependencies Report, if any parallel site uses multiple threads, this may prevent certain problems from being detected and reported by the Dependencies tool. To obtain correct values, ensure that only a single thread executes within each parallel site (see Run Your Program as a Serial Program above).

**See Also**

Suitability Analysis
Using Intel® Inspector and Intel® VTune™Profiler

**Annotations**

You add Intel® Advisor annotations to mark the places in serial parts of your program where Intel® Advisor tools should assume your program's parallel execution and synchronization will occur. Later, after you modify your program to prepare it for parallel execution, you replace these annotations with parallel framework code that enables parts of your program to execute in parallel.

Annotations are either subroutine calls or macro uses, depending on which language you are using, so they can be processed by your current compiler. The annotations do not change the computations of your program, so your application runs normally.
The three main types of annotations mark the location of:

- A **parallel site**. A parallel site encloses one or more tasks and defines the scope of parallel execution. When converted to a parallel code, a parallel site executes initially using a single thread.

- One or more **parallel tasks** within the parallel site. Each task encountered during execution of a parallel site is modeled as being possibly executed in parallel with the other tasks and the remaining code in the parallel site. When converted to parallel code, the tasks will run in parallel. That is, each instance of a task's code may run in parallel on separate cores, and the multiple instances of that task's code also runs in parallel with multiple instances of any other tasks within the same parallel site.

- Locking synchronization, where mutual exclusion of data access must occur in the parallel program.

In addition, there are:

- Annotations that stop and resume data collection. Data collection occurs while the target executes. These annotations allow you to skip uninteresting parts of the target program's execution.

- Special-purpose annotations used in less common cases.

The three Intel Advisor tools recognize the three main types of annotations and the Stop and Resume Collection annotations. Only the Dependencies tool processes the special-purpose annotations.

Use the parallel site and task annotations to mark the code regions that are candidates for adding parallelism. These annotations enable the Intel® Advisor Suitability and Dependencies tools to predict your serial program's parallel behavior. For example:

- The Suitability tool runs your program and uses parallel site and task boundaries to predict your parallel program's approximate performance characteristics.

- The Dependencies tool runs your program and uses parallel site and task boundaries to check for data races and other data synchronization problems.

One common use of sites and tasks is to enclose an entire loop within a parallel site, and to enclose the body of the loop in a task. For example, the following C/C++ code shows a simple loop that uses two parallel site annotations and one task annotation from the nqueens_Advisor sample. The three added annotations and the line that includes the annotation definitions appear in a bold font below.

```c
#include "advisor-annotate.h"
...
void solve() {
    int * queens = new int[size]; //array representing queens placed on a chess board...
    ANNOTATE_SITE_BEGIN(solve);
    for(int i=0; i<size; i++) {
        // try all positions in first row
        ANNOTATE_ITERATION_TASK(setQueen);
        setQueen(queens, 0, i);
    }
    ANNOTATE_SITE_END();
...
}
```

The following code from the Fortran nqueens sample shows the use of parallel site and task Fortran annotations, such as `call annotate_site_begin("label")`. The three added annotations and the line that references the annotation definitions module (the `use` statement) appear in a bold font below.

```fortran
use advisor_annotate ...
!
Main solver routine
subroutine solve (queens)
    implicit none
    integer, intent(inout) :: queens(:)
    integer :: i
    call annotate_site_begin("solve")
    do i=1,size
        ! try all positions in first row
        call annotate_iteration_task("setQueen")
        call SetQueen (queens, 1, i)
```
The following code from the C# nqueens sample on Windows* OS systems shows the use of parallel site and task C# annotations, such as Annotate.SiteBegin("label"). The three added annotations and the line that allows use of the annotation definitions (using directive) appear in a bold font below.

```csharp
using AdvisorAnnotate;
...
public void Solve()
{
    int[] queens = new int[size]; //array representing queens on a chess board. Index is row position, value is column.
    Annotate.SiteBegin("solve");
    for (int i = 0; i < size; i++)
    {
        Annotate.IterationTask("setQueen");
        // try all positions in first row
        SetQueen(ref queens, 0, i);
    }
    Annotate.SiteEnd();
...
}
```

To simplify adding annotations:

- When using the Microsoft Visual Studio* code editor, you can use the Annotation Wizard.
- With any editor, use the annotation assistant in the Survey windows or the No Data message. The annotation assistant displays example annotated code and build settings that you can copy to your application's code.

If you manually type annotations, you should place each annotation on a separate line and use the correct data type for annotation arguments. With C/C++ code, do not place annotations in macros so that references go to the correct source location.

You can experiment by modifying annotations and running the tools again to locate the best places to add parallelism.

For each source compilation module that contains annotations, in addition to adding the annotations, you need to:

- In files where you add annotations, add a source line to reference the Intel Advisor file that defines the annotations:
  - For C/C++ modules, include the `advisor-annotate.h` header file by adding either `#include "advisor-annotate.h"` or `#include <advisor-annotate.h>`.
  - For Fortran compilation units, add the `use advisor_annotate` statement.
  - For C# modules (on Windows* OS), add the `using AdvisorAnnotate;` directive.
  - Specify the Intel Advisor include directory when you build your C/C++ or Fortran application, so the compiler can find this include file. Similarly, you need to add the C# annotations definition file to your C# project.
  - For native applications, add the build (compiler and linker) settings.

**Annotation Types**

**Annotation Types Summary**

You can use different kinds of Intel® Advisor annotations to mark where you propose to have parallel sites, tasks, locks, or perform special actions. These annotations are:

- Parallel site annotations
- Parallel task annotations
Parallel lock annotations
Annotations that let you pause and resume data collection
Special-purpose annotations

To be useful, a parallel site must contain at least one task. Code within a parallel task can be executed by multiple threads independently of other instances of itself and also other parallel tasks. Many tasks are code within a loop, or they could be a single statement that does an iterative operation. After you use the Survey or similar profiling tool to locate where your program spends its time, you will see two general types of parallel code regions (parallel sites):

- **A simple loop that requires only a single task.** For the common case where the Survey tool identifies a simple loop structure whose iterations consume much of an application’s CPU time and the entire loop body should be a task, you may only need a single task within a parallel site. Unless your time-consuming code is not in a loop or has task(s) in a complex loop, start with this simple form. Add annotations to mark the beginning and end of the parallel site around the loop, and add one task-iteration annotation at the start of the loop body. This annotation form is the easiest to convert to parallel code.

- **Code whose characteristics require multiple tasks.** Depending on the application code characteristics, you may need multiple tasks. For example, you may have statements that can each become separate tasks, or complex or nested loop structures where you need multiple tasks to meet scalability requirements. In this case, add site annotations to mark the beginning and end of the parallel site region and also task annotations that mark the beginning and end of each task.

The two task annotation types use the same parallel site annotations. The following table lists the annotations by category type, including the syntax for the C/C++, Fortran, and C# languages. Each has a link to its detailed description.

Optional arguments are identified using square brackets, such as `annotation([int expr])`.

---

**NOTE**
To help you add annotations, use the Intel Advisor annotation assistant in the Survey windows or the No Data message to copy and add code snippets or the Annotation Wizard if you use the Microsoft Visual Studio* code editor (see Inserting Annotations Using the Annotation Wizard). You also need to add the reference to the annotations definitions file.

<table>
<thead>
<tr>
<th>Brief Description</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>Site and task annotations for a parallel site that contains a loop with a single task:</td>
<td></td>
</tr>
<tr>
<td>Start a parallel site that contains a single task in a loop.</td>
<td>C/C++: <code>ANNOTATE_SITE_BEGIN(sitename)</code>;</td>
</tr>
<tr>
<td></td>
<td>Fortran: <code>call annotate_site_begin(sitename)</code></td>
</tr>
<tr>
<td></td>
<td>C#: <code>Annotate.SiteBegin(sitename);</code></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>Mark an iterative parallel task in a loop. Place this annotation near the start of the loop body within the parallel site's execution.</td>
<td>C/C++: <code>ANNOTATE_ITERATION_TASK(taskname)</code>;</td>
</tr>
<tr>
<td></td>
<td>Fortran: <code>call annotate_iteration_task(taskname)</code></td>
</tr>
<tr>
<td></td>
<td>C#: <code>Annotate.IterationTask(taskname);</code></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>End a parallel site. The parallel site terminates only after all tasks that started within it have completed.</td>
<td>C/C++: <code>ANNOTATE_SITE_END([sitename]);</code> // sitename is optional</td>
</tr>
<tr>
<td></td>
<td>Fortran: <code>call annotate_site_end</code></td>
</tr>
<tr>
<td></td>
<td>C#: <code>Annotate.SiteEnd();</code></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>Site and task annotations for parallel site code that contains multiple tasks (all other situations):</td>
<td></td>
</tr>
<tr>
<td>Brief Description</td>
<td>Name</td>
</tr>
<tr>
<td>----------------------------------------------------------------------------------</td>
<td>----------------------------------------------------------------------</td>
</tr>
</tbody>
</table>
| Start a parallel site that contains multiple tasks, or task(s) within non-loop code or complex loop code. | C/C++: ANNOTATE_SITE_BEGIN(sitename);  
Fortran: call annotate_site_begin(sitename)  
C#: Annotate.SiteBegin(sitename); |
| Start a parallel task. Must execute within a parallel site that contains multiple tasks, or task(s) within non-loop code or complex loop code. | C/C++: ANNOTATE_TASK_BEGIN(taskname);  
Fortran: call annotate_task_begin(taskname)  
C#: Annotate.TaskBegin(taskname); |
| End a parallel task. Must execute within a parallel site that contains multiple tasks, or task(s) within non-loop code or complex loop code. | C/C++: ANNOTATE_TASK_END([taskname]); //taskname is optional  
Fortran: call annotate_task_end  
C#: Annotate.TaskEnd(); |
| End a parallel site. The parallel site terminates only after all tasks that started within it have completed. | C/C++: ANNOTATE_SITE_END([sitename]); // sitename is optional  
Fortran: call annotate_site_end  
C#: Annotate.SiteEnd(); |
| **Lock Annotations:** describe synchronization locations.                         |                                                                      |
| Acquire a lock (0 is a valid address). Must occur within a parallel site.         | C/C++: ANNOTATE_LOCK_ACQUIRE(pointer-expression);  
Fortran: call annotate_lock_acquire(address)  
C#: Annotate.LockAcquire([int expr]);  
// this C# argument is optional |
| Release a lock. Must occur within a parallel site.                                | C/C++: ANNOTATE_LOCK_RELEASE(pointer-expression);  
Fortran: call annotate_lock_release(address)  
C#: Annotate.LockRelease([int expr]);  
// this C# argument is optional |
| **Pause Collection and Resume Collection Annotations:** lets you pause data collection to skip uninteresting code. |                                                                      |
| Pause Collection. The target program continues to execute.                       | C/C++: ANNOTATE_DISABLE_COLLECTION_PUSH;  
Fortran: call annotate_disable_collection_push()  
C#: Annotate.DisableCollectionPush(); |
Resume Collection after it was stopped by a Pause Collection annotation.

C/C++: ANNOTATE_DISABLE_COLLECTION_POP;
Fortran: call annotate_disable_collection_pop()
C#: Annotate.DisableCollectionPop();

Special-purpose Annotations: describe certain memory allocations to avoid false conflicts, disable reporting of problems or analysis, or enable reporting more detail for memory accesses. These apply only to the Dependencies tool. For their syntax, see the Special-purpose Annotations help topic.

See Also
Intel Advisor Annotation Definitions File
Site and Task Annotations for Simple Loops With One Task
Site and Task Annotations for Loops with Multiple Tasks
Adding Annotations in Your Source Code
Lock Annotations
Pause Collection and Resume Collection Annotations
Special-purpose Annotations
Annotating Code for Deeper Analysis
Copying Annotations and Build Settings Using the Annotation Assistant Pane
Inserting Annotations Using the Annotation Wizard

Annotation General Characteristics

Usage
Annotations typically expand to calls to one or more functions, with minimal additional code. When you run the Suitability or Dependencies tools, the calls are instrumented during data collection.

Most annotations must be used in pairs that will execute in a begin-end sequence, such as the parallel site annotations for a site with a single task:

- For C/C++: ANNOTATE_SITE_BEGIN(sitename); and ANNOTATE_SITE_END();
- For Fortran: call annotate_site_begin(sitename) and call annotate_site_end
- For C#: Annotate.SiteBegin(sitename); and Annotate.SiteEnd();

Any mismatched annotations show up as error during data collection.
For example, if your C/C++ code has an ANNOTATE_SITE_BEGIN(); that is executed, but no corresponding ANNOTATE_SITE_END();, you will see a message, such as: Error: Missing end site when you run the Suitability or Dependencies tool.
You can also use annotations when they are dynamically paired. This lets you annotate code regions that might have more than one exit point. For example, consider this parallel site with multiple tasks:

```c
//Show that an end task annotation should be repeated for a jump out of a loop
ANNOTATE_SITE_BEGIN(for_site1);
    ANNOTATE_TASK_BEGIN(for_taskA);
    for ()
    {
        if()
            ANNOTATE_TASK_END();
        break;
        ANNOTATE_TASK_END(); // unreachable!
    }
    ANNOTATE_TASK_BEGIN(for_taskB);
    ...
    ANNOTATE_TASK_END();
ANNOTATE_SITE_END();
```

With C/C++, when you add annotations after a loop that executes only one statement without opening and closing braces ( { and } ), add opening and closing braces to allow multi-statement execution of both the original statement and the added annotation statement.

From a program source perspective, the annotation macros expand as a single executable statement (or to nothing if null expansion is used). This allows annotations to be used in locations requiring a single statement safely, as in this example:

```c
... if (!initialized)
    ANNOTATE_RECORD_ALLOCATION(my_buffer, my_buffer_size);
...
```

Guidelines for Placing Annotations in Source Code

Intel Advisor guidelines for placing annotations in source code are similar to debugger breakpoint limitations. The rules include:

- Place each annotation on a separate statement line. That is, do not place multiple annotations in a single statement line.
- With C/C++ code, do not place annotations inside preprocessor macros.

The following shows correct coding using one annotation per statement line:

```c
ANNOTATE_TASK_BEGIN(foo);
    call xyz();
ANNOTATE_TASK_END();
```

If you do not follow these guidelines, you may see unexpected Unmatched annotations in the **Dependencies Report** window (see the Troubleshooting topic below) or annotation-related errors in the Suitability Report window.

Semantics

When you run the Suitability or Dependencies tool to collect interactions between your tasks, the execution of annotations and their implications for other operations are tracked by the tool during serial execution, and the results of analysis are displayed in the corresponding Report.

When you run the Dependencies tool, the primary problems of interest are the data interactions that need attention. However, some semantic errors in the use of the annotations in your program may also be reported.
Site and Task Annotations for Simple Loops With One Task

Parallel site annotations mark the beginning and end of the parallel site. In contrast, to mark an entire simple loop body as a task, you only need a single iteration task annotation in the common case where the Survey tool identifies a single simple loop that consumes much of an application's time. In many cases, a single time-consuming simple loop structure may be the only task needed within a parallel site. This annotation form is also the easiest to convert to parallel code.

**NOTE**
If the task's code does not include the entire loop body, or if you need multiple tasks in one parallel site or for complex loops, use the task begin-end annotation pair to mark each task.

Use the general site/task annotation form for time-consuming code not in a loop, for complex loops containing task(s), or cases that require multiple tasks within a parallel site.

**Syntax: Simple Loops With One Task**

Parallel site annotations mark the parallel site that wraps the loop:

- **C/C++**: `ANNOTATE_SITE_BEGIN(sitename); and ANNOTATE_SITE_END();`
- **Fortran**: `call annotate_site_begin(sitename) and call annotate_site_end`
- **C#**: `Annotate.SiteBegin(sitename); and Annotate.SiteEnd();`

The iteration task annotation occurs within the parallel site. Place this annotation near the start of the loop body to mark an entire simple loop body as a task:

- **C/C++**: `ANNOTATE_ITERATION_TASK(taskname);`
- **Fortran**: `call annotate_iteration_task(taskname)`
- **C#**: `Annotate.IterationTask(taskname);`

For the C/C++ `ANNOTATE_SITE_END();` annotation, the `sitename` argument is optional.

The `sitename` and `taskname` must follow the rules for annotation name arguments:

- For C/C++ code, the `sitename` must be an ASCII C++ identifier. This should be a name you will recognize when it appears in Intel Advisor tool reports.
- For Fortran code, the `sitename` must be a character constant. This should be a name you will recognize when it appears in Intel Advisor tool reports.
- For C# code, the `sitename` must be a string. This name should be a string that you will easily remember when it appears in Intel Advisor tool reports.
Examples: Simple Loops With One Task

The following C/C++ code fragment shows a parallel site for a loop with a single task, where the task includes the entire simple loop body:

```c
... ANNOTATE_SITE_BEGIN(sitename);
    for (i=0; i<N; i++) {
        ANNOTATE_ITERATION_TASK(taskname);
        func(i);
    }
    ANNOTATE_SITE_END();
...```

The following Fortran code fragment shows a parallel site for a loop with a single task, where the task includes the entire simple loop body:

```fortran
... call annotate_site_begin("sitename")
    do i=1,size
        call annotate_iteration_task("taskname")
        call func(i)
    end do
    call annotate_site_end
...```

The following C# code fragment shows a parallel site for a loop with a single task, where the task includes the entire simple loop body:

```csharp
... Annotate.SiteBegin("sitename");
    for (int i = 0; i < N; i++) {
        Annotate.IterationTask("taskname");
        func(i);
    }
    Annotate.SiteEnd();
...```

With Visual Studio projects, parallel sites may span project boundaries, but the parallel sites and their related annotations should be placed within the set of projects that the startup project depends on. You may need to use the Visual Studio* Project Dependencies context menu item to add appropriate dependencies - see the help topic Troubleshooting Unexpected Unmatched Annotations.

The nqueens_Advisor C++ sample and the nqueens_Fortran Fortran sample demonstrate this form of site/task annotations. For example, the C++ annotated code in nqueens_annotated.cpp:

```cpp
ANNOTATE_SITE_BEGIN(solve);
    for(int i=0; i<size; i++) {
        // try all positions in first row
        // create separate array for each recursion
        ANNOTATE_ITERATION_TASK(setQueen);
        // int * queens = new int[size]; //array representing queens placed on a chess ...
        // ADVISOR COMMENT: This is incidental sharing because all the tasks are using ...
        setQueen(queens, 0, i);
    }
    ANNOTATE_SITE_END();
```

The help topic Annotating Parallel Sites and Tasks describes adding parallel sites and tasks.

See Also
Site and Task Annotations with Multiple Tasks
Annotating Parallel Sites and Tasks
Site and Task Annotations for Parallel Sites with Multiple Tasks

Parallel site annotations mark the beginning and end of the parallel site. Similarly, begin-end parallel task annotations mark the start and end of each task region. Use this begin-end task annotation pair if there are multiple tasks in a parallel site, if the task code does not include all of the loop body, or for complex loops or code that requires specific task begin-end boundaries, including multiple task end annotations.

Syntax: Parallel Sites with Multiple Tasks

Parallel site annotations that mark the parallel site:

<table>
<thead>
<tr>
<th>Language</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>C/C++</td>
<td><code>ANNOTATE_SITE_BEGIN(sitename); and ANNOTATE_SITE_END();</code></td>
</tr>
<tr>
<td>Fortran</td>
<td><code>call annotate_site_begin(sitename) and call annotate_site_end</code></td>
</tr>
<tr>
<td>C#</td>
<td><code>Annotate.SiteBegin(sitename); and Annotate.SiteEnd();</code></td>
</tr>
</tbody>
</table>

Parallel task annotations that mark each task within the parallel site:

<table>
<thead>
<tr>
<th>Language</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>C/C++</td>
<td><code>ANNOTATE_TASK_BEGIN(taskname); and ANNOTATE_TASK_END();</code></td>
</tr>
<tr>
<td>Fortran</td>
<td><code>call annotate_task_begin(taskname) and call annotate_task_end</code></td>
</tr>
<tr>
<td>C#</td>
<td><code>Annotate.TaskBegin(taskname); and Annotate.TaskEnd();</code></td>
</tr>
</tbody>
</table>

For the C/C++ `ANNOTATE_TASK_END();` annotation, the `taskname` argument is optional.

The `taskname` must follow the rules for annotation name arguments:

- For C/C++ code, the `taskname` must be an ASCII C++ identifier. This should be a name you will recognize when it appears in Intel Advisor tool reports.
- For Fortran code, the `taskname` must be a character constant. This should be a name you will recognize when it appears in Intel Advisor tool reports.
- For C# code, the `taskname` must be a string. This name should be a string that you will easily remember when it appears in Intel Advisor tool reports.

If you previously used site and task annotations for simple loops with one task and need to convert the task to this general, multiple task form, replace the single iteration loop annotation with a pair of task begin and task end annotations that mark the task region. Both forms use the same parallel site annotations.

Examples: Parallel Site, Multiple Tasks Not in a Loop

The `stats C++` sample application shows task parallelism with multiple tasks that are in a parallel site but not in a loop. In this case, several related statements do a lot of computation work and each can be a separate task:

```cpp
ANNOTATE_SITE_BEGIN(MySite1);
cout << "Start calculating running average..."<<endl;
ANNOTATE_TASK_BEGIN(MyTask1);
runningAvg(vals, SIZE, rnAvg);
ANNOTATE_TASK_END(MyTask1);
```
cout << "Start calculating running standard deviation..." << endl;
ANNOTATE_TASK_BEGIN(MyTask2);
runningStdDev(vals, SIZE, rnStdDev);
ANNOTATE_TASK_END(MyTask2);
ANNOTATE_SITE_END(MySite1);

In addition to calling functions that perform the computations, there are other cases where the Survey tool may indicate that a single statement consumes a lot of CPU time. For example, a Fortran array assignment for a very large array.

**Examples: Parallel Site, Multiple Tasks Within a Loop**

The annotations in the following C/C++ code fragment specify that each iteration of the loop can be two separate tasks, potentially running in parallel with any other iteration and the other task.

```cpp
...
ANNOTATE_SITE_BEGIN(sitename);
for (I=0; i<N; I++) {
    ANNOTATE_TASK_BEGIN(task1);
    func1(I);
    ANNOTATE_TASK_END();
    ANNOTATE_TASK_BEGIN(task2);
    func2(I);
    ANNOTATE_TASK_END();
}
ANNOTATE_SITE_END();
...
```

The following Fortran code fragment also shows the Fortran site and task annotations, where each iteration of the loop can be two separate tasks, potentially running in parallel with any other iteration and the other task.

```fortran
...  
call annotate_site_begin("sitename ")
do i=1,size
   call annotate_task_begin("task1")
call func1(i)
call annotate_task_end
   call annotate_task_begin("task2")
call func2(i)
call annotate_task_end
end do
call annotate_site_end
...  
```

The following C# code fragment also shows the C# site and task annotations, where each iteration of the loop can be two separate tasks, potentially running in parallel with any other iteration and the other task.

```csharp
...
Annotate.SiteBegin("sitename");
for (int i = 0; i < N; i++) {
    Annotate.TaskBegin("task1");
    func1(i);
    Annotate.TaskEnd();
    Annotate.TaskBegin("task2");
    func2(i);
    Annotate.TaskEnd();
}
Annotate.SiteEnd();
...
```
The code for each task will be marked between task begin and task end annotation pairs inside a parallel site. Code that is not executed in any task is executed by the thread entering the site, which may run in parallel with the identified tasks. In this example, the loop control code that increments \( i \) and the compares \( i \) with \( N \) is assumed to be executed separately from the explicitly specified tasks. This means that you may see conflicts between tasks, and the code outside of any task.

When you use the Dependencies tool on the above code, the tool would report data conflicts on global data accessed by either \( \text{func1} \) or \( \text{func2} \) on a later loop iteration.

The help topic **Annotating Parallel Sites and Tasks** describes adding parallel sites and tasks.

**Parallel Site and Task Placement**

Consider the following C/C++ code:

```
...  
ANNOTATE_SITE_BEGIN(sitename);
for (i=0; i<N; i++) {
    ANNOTATE_ITERATION_TASK(taskname);
    func(i);
}
ANNOTATE_SITE_END();
...
```

In the simple case on the left, the single annotated site encapsulates the entire loop. This causes all of the iterations of the loop to potentially run all at the same time. Use this simple form of loop annotations (two site annotations and one iteration task annotation) for loops whenever possible.

In the case on the right, you are not specifying that all of the loop iterations will run in parallel, but rather that the opportunities for parallelism are only within a single iteration of the loop. In this case, only the invocations of \( \text{func1} \) and \( \text{func2} \) from one loop iteration at a time are considered as sources of potential parallelism. So, in the case on the right, you will never see conflicts between successive invocations of \( \text{func1} \), because you are specifying that you do not intend to run them in parallel.

Graphically comparing what the model considers to be in parallel for these two cases, with time progressing from left to right for each case:
The boxes shown overlapping vertically above are modeled as being executed in parallel.

The execution of `ANNOTATE_TASK_BEGIN(taskname)` and `ANNOTATE_TASK_END()` pair delimits the dynamic extent of a task. Each time the annotations are executed during Intel Advisor Dependencies or Suitability analysis to collect interactions between tasks, a dynamic extent is identified that is associated with the most closely containing dynamic site. Each task is assumed to be independent and able to be run in parallel with all other tasks inside the containing sites.

Task annotations in a multiple-task parallel site must use the following rules:

- According to execution paths, each begin task annotation must be terminated by an end task annotation.
- Task boundaries must be within parallel site boundaries.
- The argument to the task annotations follow the rules for annotation name arguments.

The only times tasks are not modeled to be executing in parallel are:

1. When tasks are using synchronization, the specific code inside the synchronized region will not be modeled to be in parallel with other code synchronized using the same lock addresses.
2. When one task creates another task, the code of the parent task executed before the second task is created is assumed to execute before the task creation. However, any code executed after the task creation is assumed to be in parallel with the nested task. For example:

```c
...  
ANNOTATE_SITE_BEGIN(sitename);  
for (I=0; i<N; I++) {  
    ANNOTATE_TASK_BEGIN(taskfunc1a);  
    func1a(I);  
    ANNOTATE_TASK_BEGIN(taskfunc1a);  
    func2(I);  
    ANNOTATE_TASK_END();  
    func1b(I);  
    ANNOTATE_TASK_END();  
}  
ANNOTATE_SITE_END();  
...  
```

In this example, `func1a(I)` is not in parallel with either `func2(I)` or `func1b(I)`. However, `func2(I)` and `func1b(I)` are modeled as being executed in parallel. This semantic interpretation allows modeling of recursion where nested calls create tasks that execute in parallel. In this example, note that while this
parallel relationship holds for tasks inside one iteration, tasks from different loop iterations will all be in parallel because they have no special relationship. For example, $\text{func1}(I)$ from one loop iteration may be executed concurrently with $\text{func2}(I)$ in a different iteration.

While you are checking Dependencies, the Dependencies tool assumes that all tasks in a given site may execute in parallel unless there is explicit synchronization. For example, in this case all $N$ iterations of $\text{func1}$ and $\text{func2}$ will execute in parallel.

```c
... ANNOTATE_SITE_BEGIN(sitename);
for (I=0; i<N; I++) {
    ANNOTATE_TASK_BEGIN(taskfunc1);
    func1(I);
    ANNOTATE_TASK_END();
    ANNOTATE_TASK_BEGIN(taskfunc2);
    func2(I);
    ANNOTATE_TASK_END();
} ANNOTATE_SITE_END();
...```

If you want to model other kinds of relationships, for example $\text{func2}$ invocations will have some form of serialization, that constraint needs to be expressed using lock annotations that mark a lock that is acquired and released for the duration of that task’s execution.

To select where to add task annotations may take some experimentation, considering factors such as average instance time and number of iterations (provided in the Suitability Report). If your parallel site has nested loops and the computation time used by the innermost loop is small, consider adding task annotations around the next outermost loop. See help topics such as How Big Should a Task Be?

**See Also**

Lock Annotations

Inserting Annotations Using the Annotation Wizard

Copying Annotations and Build Settings Using the Annotation Assistant Pane

Annotating Parallel Sites and Tasks

How Big Should a Task Be?

Dependencies Analysis

Fixing Sharing Problems

**Lock Annotations**

Lock annotations mark where you expect you will be adding explicit synchronization.

**Syntax**

```c
C/C++: ANNOTATE_LOCK_ACQUIRE(pointer-expression); and
      ANNOTATE_LOCK_RELEASE(pointer-expression);

Fortran: call annotate_lock_acquire(address) and call
        annotate_lock_release(address)

C#: Annotate.LockAcquire([int expr]); and Annotate.LockRelease([int expr]);
    (for each annotation, its argument is optional)
```

With C/C++ and Fortran programs, all of the lock annotations use an address value to represent distinct locks in your final program. You can use the address value 0 to represent a global “lock” that is the same across the entire program. With C# programs, the argument is an `int` with a default value of 0 (zero).
Intel recommends that you start by using a default lock, unless you need additional locks for performance scaling.

The modeling step is aware of the standard locking routines in the Windows* OS API, as well as Intel® Threading Building Blocks (Intel® TBB) and OpenMP*, so there is no need to annotate existing locking. Lock annotations are only required for cases where you are not already using synchronization.

The lock-acquire and lock-release annotations denote points in your program where you intend to acquire and release locks. These annotations take a single parameter, which is an address that you choose.

For example, if you decided you would have a lock used only for `glob_variable`, you specify the same memory address for all cases where you are protecting access to `glob_variable`, to represent that specific lock. The sample below uses the variable's address to represent the lock that will be associated with `glob_variable`.

You typically can use one of the following four values, using a finer granularity of synchronization when necessary:

- The value of 0 (zero) to represent a single unspecified lock that is the same across the entire program.
- The address of a data structure or other aggregation of data. This represents using a single lock for the collection of data.
- The address of a member of the data collection. This represents finer-grained locking than the previous value and provides better performance.
- A variable representing a lock as you move toward final parallel code.

This C/C++ example shows the intent for the parallel program to acquire and release a lock around the access to the global variable `glob_variable` in each task:

```c
...
extern int glob_variable = 0;
...
ANNOTATE_SITE_BEGIN(sitename);
for (I=0; I<N; I++) {
    ANNOTATE_TASK_BEGIN(taskfunc1);
    func1(I);
    ANNOTATE_LOCK_ACQUIRE(&glob_variable);
    glob_variable++;
    ANNOTATE_LOCK_RELEASE(&glob_variable);
    func2(I);
    ANNOTATE_TASK_END();
} 
ANNOTATE_SITE_END();
...
```

This Fortran example also shows the intent to acquire and release a lock around the access to the global variable `glob_variable` in each task:

```fortran
... integer :: glob_variable = 0
...
call annotate_site_begin("sitename")
do i=1,size
    call annotate_task_begin("taskfunc1")
    call func1(i)
    call annotate_lock_acquire(0)
    glob_variable = glob_variable + 1
    call annotate_lock_release(0)
    call func2(i)
    call annotate_task_end
end do
call annotate_site_end
...
This C# example also shows the intent to acquire and release a lock around the access to the global variable `glob_variable` in each task:

```csharp
... public int glob_variable {
    get {return nrOfSolutions;}
    set {nrOfSolutions = value;}
}

Annotate.SiteBegin("sitename");
for (int i = 0; i < N; i++) {
    Annotate.TaskBegin("taskfunc1");
    func1(i);
    Annotate.LockAcquire();
    glob_variable++;
    Annotate.LockRelease();
    func2(i);
    Annotate.TaskEnd();
}
Annotate.SiteEnd();
...
```

The following C/C++ example is a typical use of a data item’s address. It shows the use of an Entity address, where there is a vector of integers that are each going to have an associated lock, because the program is counting random elements of the array that will be accessed by different tasks, some of which may occasionally have the same random value. The text from adding annotations appears in **bold** below.

```c++
struct Entity {
    int val;
};
...
std::vector<Entity> v;
...

for (int I=0; i<v.size()*10000; I++) {
    int random_int = random_n();
    ANNOTATE_LOCK_ACQUIRE(&v[random_int]);
    v[random_int].val++;
    ANNOTATE_LOCK_RELEASE(&v[random_int]);
}
...
```

**Using Lock Annotations**

Lock addresses are the basis of lock annotations, and each lock address corresponds to the intent to create a unique lock, or other synchronization mechanism, in the final program. Tasks sharing a parallel site are modeled as executing in parallel unless you describe synchronization using lock addresses, or known locking mechanisms.

**See Also**

Special-purpose Annotations
Synchronizing Independent Updates
Data Sharing Problems
Inserting Annotations Using the Annotation Wizard
Copying Annotations and Build Settings Using the Annotation Assistant Pane
Pause Collection and Resume Collection Annotations

The Pause Collection and Resume Collection annotations let you stop and resume data collection to skip uninteresting parts of the target program's execution. If you pause data collection, the target executable continues to execute until you resume data collection. Pausing data collection minimizes the amount of data collected and speeds up the analysis of large applications.

In addition to these annotations, you can click certain buttons on the side command toolbar to pause or resume data collection:

- You can start the Survey and Suitability tools with data collection either paused or enabled. For example, the **StartPaused** button starts executing the target being analyzed with data collection (analysis) disabled. Also, once the tool is started, you can pause and resume data collection by using the **Pause** or **Resume** buttons or by executing the equivalent Pause Collection and Resume Collection annotations.
- You start the Dependencies tool with data collection enabled, but you can pause data collection either by using the **Pause** button or by executing the equivalent Pause Collection annotation. You can add Pause Collection and Resume Collection annotations as described below.

**Pause Collection**

This annotation completely stops the analysis of your program until the matching Resume Collection (disable-collection-pop) annotation is executed. Use this annotation to reduce the analysis overhead for certain uninteresting parts of your program. This annotation is recognized by the Dependencies, Survey, and Suitability tools. Because this annotation completely disables monitoring of most annotations, add it carefully in your source code, such as outside a parallel site. If there are multiple pushes, all have to be popped before re-enabling collection.

Syntax:

<table>
<thead>
<tr>
<th>Language</th>
<th>Annotation Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>C/C++</td>
<td>ANNOTATE_DISABLE_COLLECTION_PUSH;</td>
</tr>
<tr>
<td>Fortran</td>
<td>call annotate_disable_collection_push()</td>
</tr>
<tr>
<td>C#</td>
<td>Annotate.DisableCollectionPush();</td>
</tr>
</tbody>
</table>

This annotation takes no arguments.

**Resume Collection**

This annotation resumes the analysis previously stopped by a Pause Collection (disable-collection-push) annotation. This annotation is recognized by the Dependencies, Survey, and Suitability tools. Because the Pause Collection annotation completely disables monitoring of most annotations, add this Resume Collection annotation carefully in your source code, such as outside a parallel site.

Syntax:

<table>
<thead>
<tr>
<th>Language</th>
<th>Annotation Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>C/C++</td>
<td>ANNOTATE_DISABLE_COLLECTION_POP;</td>
</tr>
<tr>
<td>Fortran</td>
<td>call annotate_disable_collection_pop()</td>
</tr>
<tr>
<td>C#</td>
<td>Annotate.DisableCollectionPop();</td>
</tr>
</tbody>
</table>

This annotation takes no arguments.
Special-purpose Annotations

All Intel Advisor special-purpose annotations are recognized by the Dependencies tool, which observes memory accesses in great detail. Some of these annotations prevent the Dependencies tool from reporting all or specific data sharing problems, while one (Observe Uses of Storage) provides more detail about memory accesses.

NOTE
In the C/C++ syntax descriptions below, addresses and sizes are C++ expressions. Similarly, the Fortran var is a Fortran integer address.

This topic describes the following special-purpose annotations:

- Inductive Expressions Uses
- Reduction Uses
- Observe Uses of Storage
- Clear Uses of Storage
- Disable Observation Annotations
- Enable Observation Annotations
- Memory Allocation Annotations

Inductive Expressions Uses

Induction variables (such as ++i) can often be eliminated when you add parallel framework code. Use this annotation to disable reporting data sharing problems for the specified memory region. This annotation is only recognized by the Dependencies tool.

Terminate this annotation with a Clear Uses of Storage annotation.

Syntax:

| C/C++: | ANNOTATE_INDUCTION_USES(address, size); |
| Fortran: | call annotate_induction_uses(var) |
| C#: | Not supported |

- address is a C++ identifier or expression that provides information about the memory region for this annotation.
- size is a C++ identifier or expression that provides information about the memory region for this annotation.
- var is a Fortran integer address that provides information about the memory region for this annotation.

Reduction Uses

Reduction variables (such as sum += data[i]) can often be replaced with reduction operations when you add parallel framework code. Use this annotation to disable reporting data sharing problems for the specified memory region. This annotation is only recognized by the Dependencies tool.

Terminate this annotation with a Clear Uses of Storage annotation. For example, with C/C++ code:

```
ANNOTATE_REDUCTION_USES($sum, 4);
sum += a[i];
ANNOTATE_CLEAR_USES($sum);
```

Syntax:

| C/C++: | ANNOTATE_REDUCTION_USES(address, size); |
• address is a C++ identifier or expression that provides information about the memory region location for this annotation.
• size is a C++ identifier or expression that provides information about the memory region location for this annotation.
• var is a Fortran integer address that provides information about the memory region for this annotation.

Observe Uses of Storage
Use this annotation to report all accesses to the specified memory region. For example, this can help you find all of the uses of a variable to determine how you should refactor your code. This annotation gets reported as a Memory watch remark message in the Dependencies Report. This annotation is only recognized by the Dependencies tool.

NOTE
For performance reasons, this annotation may not report memory access for variables stored on the stack.

To terminate this annotation, add a Clear Uses of Storage annotation.

Syntax:

C/C++: ANNOTATE_OBSERVE_USES(address, size);
Fortran: call annotate_observe_uses(var)
C#: Not supported

• address is a C++ expression that provides information about the memory region location for this annotation.
• size is a C++ expression that provides information about the memory region location for this annotation.
• var is a Fortran integer address that provides information about the memory region for this annotation.

Clear Uses of Storage
Use this annotation to terminate these annotations: Inductive Expressions Uses, Reduction Uses, and Observe Uses of Storage. For example, when the C/C++ ANNOTATE_CLEAR_USES(); annotation terminates ANNOTATE_OBSERVE_USES();, the Dependencies tool stops reporting all uses of the specified variable. This annotation is only recognized by the Dependencies tool.

Syntax:

C/C++: ANNOTATE_CLEAR_USES(address);
Fortran: call annotate_clear_uses(var)
C#: Not supported

• address is a C++ identifier or expression that provides information about the memory region location for this annotation.
• var is a Fortran integer address that provides information about the memory region for this annotation.
Disable Observation Annotations

This annotation disables the reporting of problems until the matching Enable Observation Annotation is executed. After executing this annotation, the Dependencies tool does not report problems but continues to monitor other annotations so it can resume reporting problems if a matching Enable Observation Annotation is executed. This can be useful to suppress Dependencies problems that are false-positives or not useful in your program. Unlike `ANNOTATE_CLEARUSES` - which applies to a specific memory area - this annotation remains active until a `disable-observation-pop` annotation is executed to enable annotations. This annotation is only recognized by the Dependencies tool.

Syntax:

| C/C++: | `ANNOTATE_DISABLE_OBSERVATION_PUSH;` |
| Fortran: | `call annotate_disable_observation_push()` |
| C#: | `DisableObservationPush();` |

This annotation takes no arguments.

Enable Observation Annotations

This annotation enables the reporting of Dependencies stopped by a previous Disable Observation Annotation was executed to disable observation annotations. This annotation is only recognized by the Dependencies tool.

Syntax:

| C/C++: | `ANNOTATE_DISABLE_OBSERVATION_POP;` |
| Fortran: | `call annotate_disable_observation_pop()` |
| C#: | `Annotate.DisableObservationPop();` |

This annotation takes no arguments.

Memory Allocation Annotations

Memory allocation annotations apply only to C/C++ programs. They describe non-standard or user-defined memory allocations to avoid false conflicts reported by the Dependencies tool. Only use these Memory allocation annotations if you see false conflicts related to memory allocation in the Dependencies tool. This annotation is only recognized by the Dependencies tool.

Heap-allocated memory can be freed and then reused. If the same memory region is allocated during one task, then freed, and then re-allocated for use by a second task, this can confuse Dependencies tool analysis, because it appears as if two threads were accessing the same parallel memory region without synchronization. When the program runs in parallel runs in parallel, each thread could allocate different memory, so there is not really a data race.

The Dependencies tool understands the standard library memory allocation routines, such as `malloc` and `free`, `operator new`, and so on. However, if you have a user-defined memory allocator, the Dependencies tool may not accurately understand the memory relationships between different tasks. If your application utilizes a user-defined memory allocator, you may need to use these annotations to help the Dependencies tool understand the relationships. You place:

- `ANNOTATE_RECORD_ALLOCATION` *after* a call to your non-standard or user-defined allocator.
- `ANNOTATE_RECORD_DEALLOCATION` *before* the call to your non-standard or user-defined deallocator.

If you do not have such an allocator you can skip these annotations.
If you do have a user-defined memory allocator and you omit these annotations, you may see the effects as **Memory reuse** problems for the storage that is actually allocated by your allocator, and **Data communication** problems for the control information used by the allocator.

**Syntax:**

C/C++: 

```c
ANNOTATE_RECORD_ALLOCATION(address, size); and
ANNOTATE_RECORD_DEALLOCATION(address);
```

Fortran:  

Not supported

C#:  

Not supported

**ANNOTATE_RECORD_ALLOCATION(address, size);** specifies the storage allocated by a user-memory allocator with a specific `address` and `size`:

1. The `address` is a C++ expression that provides information about the memory region location for this annotation.
2. The `size` is a C++ expression that provides information about the memory region size for this annotation.

Use **ANNOTATE_RECORD_DEALLOCATION(address);** each time your deallocator is freeing memory.

**Static Loop Scheduling Annotations**

Loop scheduling annotation inform the Suitability tool that the following loop will be divided into equal-sized (or as equal as possible) chunks. By default, chunk size is `loop_count/number_of_threads`.

**Syntax:**

C/C++: 

```c
ANNOTATE_AGGREGATE_TASK;
```

Fortran:  

Not supported

C#:  

Not supported

**See Also**

Tips for Annotation Use with C/C++ Programs  
Pause Collection and Resume Collection Annotations  
Annotation General Characteristics  
Annotation Types Summary  
Inserting Annotations Using the Annotation Wizard  
Copying Annotations and Build Settings Using the Annotation Assistant Pane

**Intel® Advisor Annotation Definitions Files**

Intel® Advisor provides macro or routine definitions that enable use of its annotations for each language:

- For C/C++, the `advisor-annotate.h` header file defines macros that begin with `ANNOTATE_`, so you can use annotations such as `ANNOTATE_SITE_BEGIN();`.
- For Fortran, the `advisor.annotate` module declares subroutines starting with `annotate_`, so you can call annotations such as `annotate_site_begin();`.
- For C# on Windows* OS systems, the `AdvisorAnnotate` header declares an `Annotate` class containing member routines, so you can use annotations such as `Annotate.SiteBegin();`.

**Referencing the Annotation Definitions from Your Source Files**

Before you add Intel® Advisor annotations into your source files, you need to reference the definitions for the Intel® Advisor annotations:
• For C/C++, add: `#include "advisor-annotate.h"` or `#include <advisor-annotate.h>` (see Including the Annotations Header File in C/C++ Sources).

• For Fortran, add: `use advisor_annotate`

• For C#, add: `using AdvisorAnnotate;` (Windows OS systems only)

**Where to Add USE Statements in Fortran Programs**

Fortran does not have file scope declarations, so the `USE` statement needs to be inside the subroutine, function or main program where the annotation(s) appear. For example:

```fortran
program F_example
  ! The main program does not contain annotations, do not add use advisor_annotate here!
  ! some code . . .
  !
  subroutine F_sub
  ! This subroutine contains annotations, so add the use advisor_annotate statement
  use advisor_annotate
  ! some code . . .
  ! add Intel Advisor site and task annotations around compute intensive code
  ! For example, begin a parallel site: call annotate_site_begin(sitel)
  !
  end subroutine F_sub
  ! some code . . .
end program F_example
```

If the call is in a module procedure, the `USE` statement can be at the module level. For more details about placing `USE` statements, see your Fortran compiler documentation.

**Specifying Build Settings**

Specific build settings are needed for each language. Certain build settings are needed for each module that contains Intel® Advisor annotations, such as specifying the directory where the annotations definitions are located. For C/C++ and Fortran applications, other build (compiler and linker) settings are needed for all modules in an application, such as full debug information. Read the Build Settings... topics by clicking the links below under See Also for your language.

**Redistributing the Annotations Definition File(s)**

You only need annotations in your code when you are using the Intel® Advisor Suitability and Dependencies tools to predict your serial program's parallel behavior. Before you distribute your application, you will typically replace these annotations when you add the parallel framework code. However, because the annotations do not change how your applications runs unless you use Intel® Advisor tools, you can distribute your application with the annotations still present.

For information about redistributing the annotation definition files, see the installed End User License Agreement (EULA.rtf or EULA.txt) and the `redist.txt` file installed in the Intel® Advisor.../documentation/<locale> directory.

**Special Considerations for C/C++ Applications**

With C/C++ programs:

- If your program encounters errors when you include the `advisor-annotate.h` file, see Handling Compilation Issues that Appear After Adding `advisor-annotate.h` (primarily for Windows systems).
On Windows OS systems: If you do need to modify the advisor-annotate.h file, you can add a copy of it for a specific project or solution. If the version of advisor-annotate.h changes, you will need to update your copies of the file. See Adding a Copy of the Annotations Include File to Your Visual Studio Project.

If you do not need to modify this file, you can reference the same installed advisor-annotate.h from multiple projects or solutions as a read-only file. If you use the Intel® Advisor environment variable and the version of Intel® Advisor advisor-annotate.h changes, you only need to change this reference if the environment variable name changes, such as for a major version. Thus, using a read-only version can minimize future maintenance.

On Linux* OS systems: Except in very rare circumstances, you can reference the same installed advisor-annotate.h from multiple projects or solutions as a read-only file.

Since the annotations do not change the values computed by your program, you can change the expansions of the macro, or suppress expansion altogether, as described in Controlling the Expansions in advisor-annotate.h.

Referencing the Annotations Definitions Directory

You need to specify the directory containing the Intel® Advisor definition file as an additional include directory when you compile your program. Intel® Advisor installs its annotation definition files into a default directory on your system. For example:

With a Visual Studio project or solution for a C/C++ or Fortran application, you need to specify the property Additional Include path. You can use the environment variable ADVISOR_<version>_DIR followed by the include directory.

With the C/C++ or Fortran command line, use the compiler option -Idir (Linux* OS) or /Idir (Windows* OS), where dir is the directory containing the annotation definition files. You can use the environment variable ADVISOR_<version>_DIR followed by the include directory.

With Fortran modules, you also need to specify the library name and directory of the annotations definitions to the linker.

With a Visual Studio project or solution for a C# program, you need to specify Properties > Add > Existing Item and browse to and select the annotations definitions file.

Tip
For the most current information on optimal C/C++ and Fortran build settings, see Build Your Target Application.

Adding a Copy of the C/C++ Annotation Definitions File to Your Visual Studio* Project

If you do not want to refer to the installed C/C++ annotation include header file, you can reference a solution- or project-specific copy of it.

To add a project-specific annotations include file to your Visual Studio project:

1. In Solution Explorer, right-click the project where you want to create the Intel® Advisor annotation header file.
2. Click Add > New Item... The Add New Item dialog box opens.
3. Under Installed Templates, click Intel Advisor [version].
4. In the middle column, click advisor-annotate.
5. Type a file name for this include file, such as advisor-annotate.h for the C/C++ header file.
6. Verify the directory containing the solution- or project-specific header file and click Add.

In Solution Explorer, a copy of the header file appears as a file under the project folder.

See Also
Including the Annotations Header File in C/C++ Sources
Inserting Annotations Using the Annotation Wizard
Including the Annotations Header File in C/C++ Sources

When you add annotations to your C/C++ source files, you also need to include the Intel® Advisor annotation header file `advisor-annotate.h` in those files. Use the code editor to type the line or use the context menu item to add a `#include` directive.

To include the annotations C/C++ header file, specify one of the following forms listed below:

<table>
<thead>
<tr>
<th>Form</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Use the quoted form to have the preprocessor first search for the header file in the same directory as the source file that contains the <code>#include</code> directive, and then other directories (see your compiler documentation for details).</td>
<td><code>#include &quot;advisor-annotate.h&quot;</code></td>
</tr>
<tr>
<td>Use the angle bracket form to have the preprocessor first search for the header file in the directory specified by the <code>/I</code> option (Additional Include Directories), and then other directories (see your compiler documentation for details).</td>
<td><code>#include &lt;advisor-annotate.h&gt;</code></td>
</tr>
</tbody>
</table>

To use the include file with Fortran or C# sources, see Intel® Advisor Annotation Definitions File.

See Also

Inserting Annotations Using the Annotation Wizard
Setting and Using Intel Advisor Environment Variables

Adding Annotations into Your Source Code

You can add Intel® Advisor annotations in your source code by:

- Copying annotations with the annotation assistant in the Survey Report window, Survey Source window, or the No Data message. Use the annotation assistant to copy the main annotations for parallel sites, tasks, and locks. For example, the annotation assistant appears in the lower part of the Survey Report window in the Assistance tab.

- **On Windows® OS only:** When using the Visual Studio® code editor, you can use the Annotation Wizard to select an annotation type and add the annotations and their arguments into your code. You can use the Annotation Wizard to add parallel site, task, lock, pause/resume data collection, and special-purpose annotations.

**NOTE**

If your sources include huge source files that contain annotations, be aware that only the first 8 MB of each file will be parsed for annotations. If not all of your annotations are being parsed in such huge source files, consider breaking each huge source file into several source files.

Inserting Annotations Using the Annotation Wizard

Adding annotations requires you to reference the annotation definitions include file as well as include it from each source file that contains Intel® Advisor annotations.

**NOTE**

The Annotation Wizard is supported only in the Microsoft Visual Studio® code editor. Alternatively, you can copy annotation code snippets using any editor.

Use the annotation wizard to add Intel® Advisor annotations to your program.

1. In the Visual Studio® editor, select the code section that you wish to annotate.
2. Right-click to open the context menu and select Intel Advisor [version] > Annotation Wizard...
The Annotation Wizard opens with the default **Annotate Site – select task annotations below** annotation selected:

3. From the **Choose the Annotation Type** drop-down, choose the appropriate annotation type. For example, if you want to add the parallel site (parallel code region) annotations and a single task annotation within that site, start by selecting the site code block and choose **Site and Iteration Task Snippet, single iteration task in loop**. In other cases, you may need to add two separate annotations - one for the site and one for the task(s). In this case, after adding them, move individual annotation lines around your existing site and task(s) code.

   Your code appears in the **Example** section of the dialog, with the annotation line(s) highlighted in red font.

4. Click **Next** to configure the parameters of the opening annotation line.

   For Annotation types that include parameters, page 2 of the wizard appears. Site, task, and other annotations take name arguments. You should replace the added name with a name that helps you quickly identify its source location. For example, if `MySite1` is the argument to a site annotation, replace it with a meaningful function or loop name. The added name must be unique amongst the annotations in this project. For Annotation types that do not include parameters, go to step 8.

5. Specify the parameter values for the first parameter in the Annotation type, or use the default text that appears in the wizard.

   The highlighted annotation line now has your specified parameter value entered in the annotation line.

6. Click **Next** to configure, or keep the default text for the next parameter. Repeat for all the parameters.

   The highlighted annotation line now has all parameters filled in with values entered in the annotation line.

7. Click **Next** to go to page 3 of the wizard and review the annotation line(s) before adding it to your code.

8. Click **Finish** to add the annotation line with your specified parameters to your code.

   The Wizard closes and the editor shows the annotation lines added to the code.

   The annotation line(s) are added in the code editor.

   If a loop only executes a single statement and does not contain an opening brace (`{`) to allow multi-statement execution, add braces (`{` and `}`) around the existing statement and the annotation.

---

**Code After Adding a Pair of Parallel Site Annotations**

The following screen capture shows the C/C++ annotation lines for an annotation of type **Annotate Site**, where the site name `MySite1` parameter was replaced by typing a meaningful name `queens`: 

---
To access this dialog box: in the Visual Studio code editor, right-click to open the context menu, and select > Annotation Wizard.

The Annotation Wizard helps you add annotations into your code. After you are done adding annotations, rebuild your program.

Use page 1 of the wizard to select the type of annotation from the Choose the annotation type drop down list.

The wizard shows the annotation line(s) that will be added into your code, in red font.

- For annotations that have start and end lines, it adds them around your selected code lines with placeholders for the parameters.
- For annotations that have only one line, it adds that annotation line before your selected code line, with placeholders for the parameters.

After choosing the annotation type, click Next to go to the next wizard page and fill in the parameters for the annotation.

See Also
- Annotation Wizard - Page 2
- Annotation Wizard - Page 3

Annotation Wizard - Page 2

The Annotation Wizard helps you add annotations into your code. After you are done adding annotations, rebuild your target executable.

Use page 2 of the wizard to define the annotation parameters. Replace the placeholder text in the Specify annotation parameter field with the parameters you want to define for the annotation, or keep the default text. For example, for an Annotation Site annotation, replace the <site_name> placeholder text with a meaningful site name.

The wizard shows your parameters within the annotation line(s).

See Also
- Annotation Wizard - Page 1
- Annotation Wizard - Page 3
Annotation Wizard - Page 3

The **Annotation Wizard** helps you add annotations into your code. After you are done adding annotations, rebuild your program.

Use page 3 of the wizard to check the annotation line(s) that you defined using the wizard and verify that the line(s) and insertion location(s) are correct:

- If you are satisfied with the annotation line(s), click **Finish** to add the line(s) to your code.
- To revise the annotation, click **Back** and revise the annotation type, or the parameters defining the annotation.
- To change the location of the added lines, click **Cancel**, select a different range of code lines and right-click and choose **Intel Advisor [version] > Annotation Wizard** to add annotation lines around the new selection.

**Copying Annotations and Build Settings Using the Annotation Assistant Pane**

The Intel® Advisor provides an annotation assistant near the bottom of the **Survey Report** and **Survey Source** windows, as well as with the **No Data** message. Use this assistant to view and copy selected annotated code snippets and build setting information into a code editor.

The assistant provides a drop-down list under **Example**: from which you select one of the following:

<table>
<thead>
<tr>
<th>Select This</th>
<th>To Do This</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Iteration Loop, Single Task</strong></td>
<td>View and copy an annotation code snippet for a simple loop structure, where the task's code includes the entire loop body. Use this common task structure when only a single task is needed within a parallel site. For example code, see the help topic Site and Task Annotations for Simple Loops With One Task.</td>
</tr>
<tr>
<td><strong>Loop, One or More Tasks (bounded)</strong></td>
<td>View and copy an annotation code snippet for a loops where the task code does not include all of the loop body, or for complex loops or code that requires specific task begin-end boundaries, including multiple task end annotations. Also use this structure when multiple tasks are needed within a parallel site. For example code, see the help topic Site and Task Annotations for Parallel Sites with Multiple Tasks.</td>
</tr>
<tr>
<td><strong>Function, One or More Tasks (bounded)</strong></td>
<td>View and copy an annotation code snippet for code that calls multiple functions (task parallelism). Use this structure when multiple tasks are needed within a parallel site. For example code, see the help topic Site and Task Annotations for Parallel Sites with Multiple Tasks.</td>
</tr>
<tr>
<td><strong>Pause/Resume Collection</strong></td>
<td>View and copy an annotation code snippet whose annotations temporarily pause data collection and later resume it. This lets you skip uninteresting parts of the target program's execution to minimize the data collected and speed up the analysis of large applications. Add these annotations outside a parallel site.</td>
</tr>
<tr>
<td><strong>Build Settings</strong></td>
<td>View and copy build settings. The Build Settings are specific to the language in use.</td>
</tr>
<tr>
<td>Select This</td>
<td>To Do This</td>
</tr>
<tr>
<td>-------------</td>
<td>------------</td>
</tr>
<tr>
<td>Click the <img src="clipboard" alt="Copy to Clipboard" /> button to copy the selected snippet to the clipboard.</td>
<td></td>
</tr>
</tbody>
</table>

Site, task, and other annotations take name arguments. You should replace the placeholder name with a name that helps you quickly identify its source location. For example, in place of `MySite5` in the argument to a site annotation, replace it with a meaningful function or loop name. The name you add must be unique amongst the annotations in this project.

### Inserting Annotations in the Visual Studio* Code Editor

To add Intel® Advisor annotations into your source files, you can use the Visual Studio* code editor. Intel® Advisor simplifies the process of adding annotations so you do not need to type the annotation names. Alternatively, you can use the annotation assistant in the Survey Report or Survey Source windows, or when using the Visual Studio code editor, the Annotation Wizard (on Windows* OS systems). Alternatively, you can type the exact macro name and its arguments manually.

To add Intel® Advisor annotations:

1. Open the source file into which you want to add Intel® Advisor annotations in your code editor. You should start with the outermost code regions, such as a parallel site, and then add the tasks within the boundaries of the enclosing site.

2. Select the code region around which you will add your first annotation, such as a parallel site. Carefully include the correct group of lines, including any opening and closing braces (`{ `and `}`) needed. For example:

```c
void solve(int size) {
    int * queens = new int[size]; // array representation

    // ADVISOR COMMENT: When surveying this is typically
    // ADVISOR COMMENT: Uncomment the four annotations
    // ADVISOR COMMENT: Don't forget to uncomment

    for(int i=0; i<size; i++) {
        // try all positions in first row
        setQueen(queens, 0, i);
    }
}
```

3. Within the highlighted code region, right-click the mouse to display the context menu. Select Intel Advisor [version] and the type of annotation to be added, such as Intel Advisor [version] > Annotate Site. For example:

4. This adds the selected type of annotations. For the begin site annotation, Intel® Advisor adds a unique annotation identifier as an argument.

```c
ANNOTATE_SITE_BEGIN(MySite1);
for(int i=0; i<size; i++) {
    // try all positions in first row
    setQueen(queens, 0, i);
}
ANNOTATE_SITE_END(MySite1);
```
You should replace the added name with a name that helps you quickly identify its source location. For example, in place of MySite1 in the argument to ANNOTATE_SITE_BEGIN() and ANNOTATE_SITE_END() shown above, you might instead type the word solve (the function name). The added name must be unique amongst the annotations in this project. Annotation name arguments for:

- C/C++ code use an ASCII C++ identifier.
- Fortran code use a character constant.
- C# code use a string (Windows OS only)

Choose a string that you will easily remember when it appears in Intel Advisor tool reports. Other annotations use address or size arguments.

5. To add more annotations in the same file, repeat this process from step 2. To add annotations in a different file, repeat this process from step 1.

This enables you to quickly add annotations into the appropriate source files.

This wizard provides only the more frequently used annotations, so some annotations are not available in this wizard. Either use the Survey windows' annotation assistant to copy other annotations or type the annotations into your code editor.

If a C/C++ loop only executes a single statement and does not contain an opening brace ({}) to allow multi-statement execution, add braces ({}) around the existing statement and the annotation.

See Also
Annotation General Characteristics
Inserting Annotations Using the Annotation Wizard
Copying Annotations and Build Settings Using the Annotation Assistant Pane
Annotation Types Summary

Inserting Annotations in a Text Editor

To add Intel® Advisor annotations into your source files on a Linux* system, you can use any text editor. Intel® Advisor simplifies the process of locating where to add annotations.

To add Intel® Advisor annotations:

1. Open the Intel® Advisor GUI and the relevant project.
2. In the File menu, select Options.
3. Select Editor to display the Options - Editor dialog box. Follow the instructions to associate an editor with your source language(s).
4. For your project, open the Survey Source window.
5. Double-click a source line to display the specified editor opened to the corresponding source location.
6. Use the annotation assistant pane in the lower part of the Survey Source window to select the type of annotation you want to add.
7. Copy the selected annotations from the annotation assistant pane by clicking the button.
8. Paste the copied annotations into your editor.
9. You may need to move some annotation lines around.
10. Repeat as needed for other annotations from step 4.

This enables you to quickly add annotations into the appropriate source files.

See Also
Annotation General Characteristics
Annotation Types Summary
Copying Annotations and Build Settings Using the Annotation Assistant Pane

Tips for Annotation Use with C/C++ Programs

The following topics provide tips related to using annotations with C/C++ programs:
• Depending on your particular environment, you may want to control the expansion of macros in *advisor-annotate.h* by using the `ANNOTATE_EXPAND_NULL` environment variable. See the help topic *Controlling the Expansion of advisor-annotate.h*.

• **Tips for Windows** OS only:
  • Because the *advisor-annotate.h* header file includes *windows.h*, including *advisor-annotate.h* may cause type and symbols conflicts, which result in unexpected compiler messages. See the help topic *Handling Compilation Issues that Appear After Adding advisor-annotate.h*.
  • If you run into certain unexpected problems, you need to learn how *advisor-annotate.h* and *libittnotify.dll* interact. See the help topic *advisor-annotate.h* and *libittnotify.dll*.

### Controlling the Expansion of advisor-annotate.h

Depending on your particular environment, you may want to control the expansion of C/C++ macros in *advisor-annotate.h* at the inclusion site.

Defining `ANNOTATE_EXPAND_NULL` before you include *advisor-annotate.h* causes the annotation macros to have a null expansion, which will remove the actions from your code. If you have a project where some configurations, or some users, want to have annotations, while others should not have them present, this control may be helpful.

```
#define ANNOTATE_EXPAND_NULL
#include "advisor-annotate.h"
```

You can also do this by defining the value as part of your compilation command using the `/D` option. For example:

```
/DANNOTATE_EXPAND_NULL
```

### See Also

- *Handling Compilation Issues that Appear After Adding advisor-annotate.h*
- *Target Applications*
- *Including the Annotations Header File in C/C++ Sources*

### Handling Compilation Issues that Appear After Adding advisor-annotate.h

**NOTE**

This topic primarily applies to Windows systems. It is possible for similar errors to occur on Linux systems.

### Symptoms

On Windows* systems, the *advisor-annotate.h* header file includes *windows.h* to define some types and functions. As a result, in some cases including *advisor-annotate.h* may cause compilation errors. For example, the following conflict for the type `UINT`:

```
error C2371: 'UINT' : redefinition; different basic types
```

On Linux systems, something similar could occur under certain very specific conditions when using a different header file for operating system threading software.

### Possible Correction Strategies

To fix this problem, you can use a declaration/definition approach, where all uses of *advisor-annotate.h* other than one generate a set of declarations, and *windows.h* is only needed in a single implementation module. In all cases, you `#define` either `ANNOTATE_DECLARE` or `ANNOTATE_DEFINE` just before the `#include "advisor-annotate.h"` as follows:
1. In nearly all modules that contain annotations, insert \texttt{#define ANNOTATE_DECLARE} just before
   \texttt{#include "advisor-annotate.h"}. This causes \texttt{advisor-annotate.h} to declare an external
   function, and not include \texttt{windows.h} (or Linux equivalent), which avoids the type/symbol conflicts with
   the operating system threading header file, such as \texttt{windows.h}.

2. In a single module that either does not include annotations or does not have type/symbol conflicts with
   \texttt{windows.h}, you insert \texttt{#define ANNOTATE_DEFINE} just before \texttt{#include "advisor-annotate.h"}.
   This causes \texttt{advisor-annotate.h} to define the global function to resolve the external reference and
   the \texttt{#include "advisor-annotate.h"} is the only one that uses the operating system threading
   header file \texttt{windows.h} (or Linux equivalent). These two lines can be placed in an otherwise empty .cpp
   file.

   One way to do this is to add an empty .cpp to your project with two lines in it, shown as \texttt{empty.cpp}
   below.

   For example, on Windows systems:

   //File foo.cpp/.h:
   ...
   // Insert \texttt{#define ANNOTATE_DECLARE} in all modules that contain annotations just before the
   // \texttt{#include "advisor-annotate.h"}. This prevents inclusion of \texttt{windows.h} to avoid the
   // type/symbol conflicts.

   \texttt{#define ANNOTATE_DECLARE}
   \texttt{#include "advisor-annotate.h"}
   ...
   // annotation uses
   \texttt{ANNOTATE_SITE_BEGIN(MySite1)}
   ...
   \texttt{ANNOTATE_SITE_END()}
   ...

   //File empty.cpp:
   // Insert \texttt{#define ANNOTATE_DEFINE} just before the \texttt{#include "advisor-annotate.h"} in only one
   // module.
   // This single implementation file (.cpp/.cxx) causes \texttt{windows.h} to be included, and the
   // support
   // routine is defined as a global routine called from the various annotation uses.
   \texttt{#define ANNOTATE_DEFINE}
   \texttt{#include "advisor-annotate.h"}
   ...

   If the problem persists, please request support, such as by using the support forum.

   \texttt{advisor-annotate.h} and \texttt{libittnotify.dll}

   \begin{quote}
   \textbf{NOTE}
   This topic is provided for reference, but it should not be needed. If you read this because you believe
   you need it to understand a problem, please provide feedback (see the release notes for support
   information).
   \end{quote}

   Code using \texttt{advisor-annotate.h} should work when running your application regardless of whether or not
   you are running your application under Visual Studio using Intel\textsuperscript{®} Advisor on Windows\textsuperscript{*} OS. However, should
   you run into problems, this topic provides a few implementation details that might be helpful to understand
   issues.

   Each compilation unit that includes \texttt{advisor-annotate.h}, and that has one or more annotations in it, will
   have a global inline routine named \_AnnotateRoutine. This routine is called from the various locations
   where you have used the \texttt{ANNOTATE*} macros, and will be used to invoke one or more routines in
   \texttt{libittnotify.dll} (on Windows OS) or \texttt{libittnotify.so} (on Linux* OS).
__AnnotateRoutine is an inline function that will have only one copy per executable in your program. So, if you are only working on modeling semantic behavior in a single Dynamic Link Library (DLL), you will only have one copy of it in that one library. If you have multiple executables where you have annotations, there will be a single copy of __AnnotateRoutine for each executable.

When the ANNOTATE macros are used, the first one executed in a given executable attempts to load libittnotify.dll from the current path.

Once the library is loaded, calls from the annotations will go to the __itt_model* routines in the library. If an expected routine is not found, the code asserts.

The following figure shows what happens when you have a main executable and a DLL that both have annotations:

Finally, the annotation routines -__itt_model* in the .dll - by themselves are only markers for a tool that interprets the calls. Unless the program is run under the tool, these routines will not do anything. The intent is that the application will run normally when it is not run under the tool.

**See Also**
Target Applications

**Annotation Report**
To access this window, in the Result tab, click the Annotation Report button. Alternatively, if you are using the Advisor Workflow tab, click the button below 2. Annotate Sources or 5. Add Parallel Framework.
The **Annotation Report** window lists all annotations found during source scanning or running the Suitability and Dependencies tools. It lists the annotation type, source location, and annotations label in a table-like grid format, where each annotation appears on a separate row. Intel® Advisor updates the listed annotations when changes occur to the specified source directories. For example, when you save a source file with a code editor.

### Annotation Report Layout

<table>
<thead>
<tr>
<th>1. Workflow Tab</th>
<th>2. Result Tab</th>
<th>3. Annotation Report window grid</th>
</tr>
</thead>
</table>

#### Use This To Do This

<table>
<thead>
<tr>
<th>Use This</th>
<th>To Do This</th>
</tr>
</thead>
<tbody>
<tr>
<td>Workflow tab</td>
<td>Run a tool of your choice and see results in the Result tab.</td>
</tr>
<tr>
<td>Result Tab</td>
<td>Select between available reports.</td>
</tr>
</tbody>
</table>

**Annotation Report** window grid

View a summary of the annotations found as well as data collected by the Suitability and Dependencies tools. Each annotation's data appears on a separate row in the grid. The columns are explained below.

Right-click a row in the **Annotation Report** window grid

Displays a context menu that lets you expand or collapse code snippets, edit corresponding source code using a code editor, copy data to the clipboard, or display context-sensitive help.

To sort the grid using a column’s values, click on the column’s heading. The columns of the grid are:

<table>
<thead>
<tr>
<th>Use This Column</th>
<th>To Do This</th>
</tr>
</thead>
<tbody>
<tr>
<td>Annotation</td>
<td>View the type of annotation, such as <strong>Site</strong>, <strong>Task</strong>, or <strong>Lock</strong>.</td>
</tr>
</tbody>
</table>

To show or hide a code snippet showing the annotation, click the icon next to its name.

For information about each annotation type, see the help topic Summary of Annotation Types.

To view the source associated with an annotation in your code editor, double-click its name or a line in the code snippet (or right-click and select **Edit Source** from the context menu) in this column.

- **On Windows® OS:**
  - When using Visual Studio, the Visual Studio code editor appears with the file open at the corresponding location.
  - When using the Intel® Advisor GUI, the file type association (or Open With dialog box) determines the editor used.

- **On Linux® OS:** When using the Intel® Advisor GUI, the editor defined by the Options > Editor dialog box appears with the file open at the corresponding location.
**Use This Column** | **To Do This**
--- | ---
Source Location | View the name of the source file that contains the annotation and the line number. Icons indicate where source is available or not available. To view the source, double-click its name (or right-click and select Edit Source) in this column. The code editor appears.
Annotation Label | View the annotation’s label (name). To view the source associated with an annotation, double-click its name (or right-click and select Edit Source) in this column. The code editor appears.

**When to View the Annotation Report**

Use the Annotation Report window to view the types of annotations present in your project sources and access their locations. You can view the Annotation Report at any time to check the annotations. In addition:

- Intel recommends that you view annotations during Adding Parallelism to Your Program workflow step to help you locate site, task, lock, and other annotations that should be replaced by parallel framework code.
- When using the Intel Advisor product GUI, you can use this report to verify that the correct sources have been defined in the Project Properties > Source Search tab.

**Annotation Report, Clear Description of Storage Row**

Use this special-purpose annotation to stop tracking references to a memory location by the Dependencies tool. This information can help you understand what code accesses a memory location. When you have learned enough, simply remove this annotation.

To view the source code for this annotation, click the icon.

**When to View the Annotation Report**

Use the Annotation Report window to view the types of annotations present in your project sources and access their locations. You can view the Annotation Report at any time to check the annotations. In addition:

- Intel recommends that you view annotations during Adding Parallelism to Your Program workflow step to help you locate site, task, lock, and other annotations that should be replaced by parallel framework code.
- When using the Intel Advisor product GUI, you can use this report to verify that the correct sources have been defined in the Project Properties > Source Search tab.

**See Also**

- Analysis Summary

**Annotation Report, Disable Observations in Region Row**

This special-purpose annotation disables the reporting of problems until the matching enable annotation ANNOTATE_DISABLE_OBSERVATION_POP; is executed. Use this annotation to suppress reported problems that are false-positives, or not useful in you.

To view the source code for this annotation, click the icon.

**When to View the Annotation Report**

Use the Annotation Report window to view the types of annotations present in your project sources and access their locations. You can view the Annotation Report at any time to check the annotations. In addition:
• Intel recommends that you view annotations during Adding Parallelism to Your Program workflow step to help you locate site, task, lock, and other annotations that should be replaced by parallel framework code.
• When using the Intel Advisor product GUI, you can use this report to verify that the correct sources have been defined in the Project Properties > Source Search tab.

Annotation Report, Pause Collection Row
This special-purpose annotation temporarily stops (pauses) the analysis of your program's execution until the matching Resume Collection annotation (disable-collection-pop) is executed. Use this annotation to reduce the tool analysis overhead and reported data for certain parts of your program while running the Dependencies, Survey, and Suitability tools.

To view the source code for this annotation, click the icon.

When to View the Annotation Report
Use the Annotation Report window to view the types of annotations present in your project sources and access their locations. You can view the Annotation Report at any time to check the annotations. In addition:
• Intel recommends that you view annotations during Adding Parallelism to Your Program workflow step to help you locate site, task, lock, and other annotations that should be replaced by parallel framework code.
• When using the Intel Advisor product GUI, you can use this report to verify that the correct sources have been defined in the Project Properties > Source Search tab.

See Also
• Analysis Summary

Annotation Report, Inductive Expression Row
This special-purpose annotation marks a line that updates an expression that is inductive in a loop.

To view the source code for this annotation, click the icon.

Inductive expressions cause dependence cycles which normally prevent parallelizing a loop, but it is possible to compute the value of the expression if you know the iteration number. You may have to re-write the inductive expression to compute the value based on the iteration number when the loop is translated to parallel code.

For example, if \( i++ \) is the iteration variable of your loop, the parallel framework that you use may automatically fix this for you. For example, by using cilk_for. Otherwise, you may need to fix it manually. A common example is with \( j+=3 \), and \( i++ \). If \( i \) is your loop index (assuming 0 based), you can replace \( j+=3 \) with \( j = i*3 \). That is, the value of \( j \) actually is a function of the value of \( i \).

When to View the Annotation Report
Use the Annotation Report window to view the types of annotations present in your project sources and access their locations. You can view the Annotation Report at any time to check the annotations. In addition:
• Intel recommends that you view annotations during Adding Parallelism to Your Program workflow step to help you locate site, task, lock, and other annotations that should be replaced by parallel framework code.
• When using the Intel Advisor product GUI, you can use this report to verify that the correct sources have been defined in the Project Properties > Source Search tab.

Annotation Report, Lock Row
A lock row shows the source location of the lock annotation and its argument value.

To view the source code for this lock annotation, click the icon.
When to View the Annotation Report

Use the **Annotation Report** window to view the types of annotations present in your project sources and access their locations. You can view the **Annotation Report** at any time to check the annotations. In addition:

- Intel recommends that you view annotations during **Adding Parallelism to Your Program** workflow step to help you locate site, task, lock, and other annotations that should be replaced by parallel framework code.
- When using the Intel Advisor product GUI, you can use this report to verify that the correct sources have been defined in the **Project Properties > Source Search** tab.

**Annotation Report, Observe Uses Row**

Use this special-purpose annotation to report the access operations to a memory location in the **Dependencies Report**. This information can help you understand what code accesses a memory location. When you have learned enough, remove the annotation from your source code.

To view the source code for this annotation, click the `🔍` icon.

When to View the Annotation Report

Use the **Annotation Report** window to view the types of annotations present in your project sources and access their locations. You can view the **Annotation Report** at any time to check the annotations. In addition:

- Intel recommends that you view annotations during **Adding Parallelism to Your Program** workflow step to help you locate site, task, lock, and other annotations that should be replaced by parallel framework code.
- When using the Intel Advisor product GUI, you can use this report to verify that the correct sources have been defined in the **Project Properties > Source Search** tab.

**Annotation Report, Reduction Row**

This special-purpose annotation marks a line that computes a reduction in a loop. Marking the line as a reduction causes the Dependencies tool to ignore the data race.

To view the source code for this annotation, click the `🔍` icon.

Reductions require special treatment when translating to parallel code (see the help topics Special-purpose Annotations and About Replacing Annotations ... for your parallel framework below).

When to View the Annotation Report

Use the **Annotation Report** window to view the types of annotations present in your project sources and access their locations. You can view the **Annotation Report** at any time to check the annotations. In addition:

- Intel recommends that you view annotations during **Adding Parallelism to Your Program** workflow step to help you locate site, task, lock, and other annotations that should be replaced by parallel framework code.
- When using the Intel Advisor product GUI, you can use this report to verify that the correct sources have been defined in the **Project Properties > Source Search** tab.

**Annotation Report, Re-enable Observations at End of Region Row**

This special-purpose annotation enables reporting problems stopped by a previous `ANNOTATE_DISABLE_OBSERVATION_PUSH;` annotation.

To view the source code for this annotation, click the `🔍` icon.
When to View the Annotation Report

Use the **Annotation Report** window to view the types of annotations present in your project sources and access their locations. You can view the **Annotation Report** at any time to check the annotations. In addition:

- Intel recommends that you view annotations during **Adding Parallelism to Your Program** workflow step to help you locate site, task, lock, and other annotations that should be replaced by parallel framework code.
- When using the Intel Advisor product GUI, you can use this report to verify that the correct sources have been defined in the **Project Properties > Source Search** tab.

**Annotation Report, Resume Collection Row**

This special-purpose annotation resumes the analysis previously stopped by a previous Pause Collection (disable-collection-push) annotation. This annotation is recognized by the Dependencies, Survey, and Suitability tools.

To view the source code for this annotation, click the icon.

When to View the Annotation Report

Use the **Annotation Report** window to view the types of annotations present in your project sources and access their locations. You can view the **Annotation Report** at any time to check the annotations. In addition:

- Intel recommends that you view annotations during **Adding Parallelism to Your Program** workflow step to help you locate site, task, lock, and other annotations that should be replaced by parallel framework code.
- When using the Intel Advisor product GUI, you can use this report to verify that the correct sources have been defined in the **Project Properties > Source Search** tab.

**See Also**

- Analysis Summary

**Annotation Report, Site Row**

A site row shows the source location of the site annotation and the label of the site.

To view the source code for this site annotation, click the icon.

When converting annotations to parallel code:

- For Intel® Threading Building Blocks (Intel® TBB), you need to add a scheduler initialization call in each thread before you create any tasks.
- For OpenMP®, it depends on the pragmas/directives used.

When to View the Annotation Report

Use the **Annotation Report** window to view the types of annotations present in your project sources and access their locations. You can view the **Annotation Report** at any time to check the annotations. In addition:

- Intel recommends that you view annotations during **Adding Parallelism to Your Program** workflow step to help you locate site, task, lock, and other annotations that should be replaced by parallel framework code.
- When using the Intel Advisor product GUI, you can use this report to verify that the correct sources have been defined in the **Project Properties > Source Search** tab.

**Annotation Report, Task Row**

A task row shows the source location of the task annotation and the label of the task. A task identifies time-consuming code whose work can be efficiently done by multiple cores.

To view the source code for this task annotation, click the icon.
When the task is translated to parallel code and you remove or comment out the task annotation(s), this entry is removed from the table.

There are two types of task annotations. If the loop code changes, you can modify the type of task annotation.

**When to View the Annotation Report**

Use the **Annotation Report** window to view the types of annotations present in your project sources and access their locations. You can view the **Annotation Report** at any time to check the annotations. In addition:

- Intel recommends that you view annotations during Adding Parallelism to Your Program workflow step to help you locate site, task, lock, and other annotations that should be replaced by parallel framework code.
- When using the Intel Advisor product GUI, you can use this report to verify that the correct sources have been defined in the **Project Properties > Source Search** tab.

**See Also**

- Analysis Summary

**Annotation Report, User Memory Allocator Use Row**

This row shows a source location where memory is being allocated using a non-standard or user-defined memory deallocation. The Dependencies tool uses this as a hint about the lifetime of memory accesses, so memory that is allocated will not cause conflicts to be reported if the non-standard or user-defined memory allocation occurs with the span this annotation's execution.

To view the source code for this annotation, click the icon.

When translating annotations to parallel code, this special-purpose record_allocation annotation can be removed or commented out.

**When to View the Annotation Report**

Use the **Annotation Report** window to view the types of annotations present in your project sources and access their locations. You can view the **Annotation Report** at any time to check the annotations. In addition:

- Intel recommends that you view annotations during Adding Parallelism to Your Program workflow step to help you locate site, task, lock, and other annotations that should be replaced by parallel framework code.
- When using the Intel Advisor product GUI, you can use this report to verify that the correct sources have been defined in the **Project Properties > Source Search** tab.

**Annotation Report, User Memory Deallocator Use Row**

This row shows a source location where memory is being freed using a non-standard or user-defined memory deallocation. The Dependencies tool uses this as a hint about the lifetime of memory accesses, so memory that is freed and then allocated again will not cause conflicts to be reported if the non-standard or user-defined memory free occurs with the span of this annotation's execution.

To view the source code for this annotation, click the icon.

When translating annotations to parallel code, this special-purpose record_deallocation annotation can be removed or commented out.

**When to View the Annotation Report**

Use the **Annotation Report** window to view the types of annotations present in your project sources and access their locations. You can view the **Annotation Report** at any time to check the annotations. In addition:
• Intel recommends that you view annotations during **Adding Parallelism to Your Program** workflow step to help you locate site, task, lock, and other annotations that should be replaced by parallel framework code.

• When using the Intel Advisor product GUI, you can use this report to verify that the correct sources have been defined in the **Project Properties > Source Search** tab.

**See Also**

• **Analysis Summary**

### Annotation Assistant Pane (No Data Message and Survey Windows)

Use this pane to view and copy annotation example code snippets and build settings. It appears near the bottom of the **Survey Report** and **Survey Source** windows, and also for the **No Data** message.

**The Annotation Assistant is available in the Threading Workflow only.**

<table>
<thead>
<tr>
<th>Use This</th>
<th>To Do This</th>
</tr>
</thead>
<tbody>
<tr>
<td>The drop-down list near the right of this pane.</td>
<td>Select one of four topics to copy. The first three provide example annotated source code for various task code structures. The last provides build settings for the language in use. The selections are:</td>
</tr>
<tr>
<td></td>
<td>• <strong>Iteration Loop, Single Task</strong> - View and copy an annotation code snippet for a simple loop structure, where the task’s code includes the entire loop body. Use this common task structure when only a single task is needed within a parallel site.</td>
</tr>
<tr>
<td></td>
<td>• <strong>Loop, One or More Tasks</strong> - View and copy an annotation code snippet for a loop where the task code does not include all of the loop body, or for complex loops or code that requires specific task begin-end boundaries, including multiple task end annotations. Also use this structure when multiple tasks are needed within a parallel site.</td>
</tr>
<tr>
<td></td>
<td>• <strong>Function, One or More Tasks</strong> - View and copy an annotation code snippet for code that calls multiple functions (task parallelism). Use this structure when multiple tasks are needed within a parallel site.</td>
</tr>
<tr>
<td></td>
<td>• <strong>Pause/Resume Collection</strong> - View and copy an annotation code snippet whose annotations temporarily pause data collection and later resume it. This lets you skip uninteresting parts of the target program’s execution to minimize the data collected and speed up the analysis of large applications. Add these annotations outside a parallel site.</td>
</tr>
<tr>
<td></td>
<td>• <strong>Build Settings</strong> - View and copy build (compiler and linker) settings. The Build Settings are specific to the language in use.</td>
</tr>
</tbody>
</table>

| Text display area in the left part of this pane. | View the selected annotation code or build settings. To copy the text lines to the clipboard, either right-click and select **Copy to Clipboard** from the context menu, or use the [Copy to Clipboard] button (below). |
Use This | To Do This
---|---
Copy to Clipboard button in the right part of this pane. | Copy the selected annotation example code or build settings to the clipboard. Paste the annotated example code into your source code editor, perhaps using an intermediate buffer. Paste build options into your build script.
Hide or show this pane. | Hide or show this pane.

**No annotations found in your project Message**

Intel Advisor provides tools that analyze your running serial program to collect and present data. These tools help you make decisions about adding parallelism to your program.

**Where should I add parallelism?** To locate where your program spends its time, run the Survey tool to profile your serial program (Release build suggested). View the Survey Report to identify at least one possible parallel code region (site).

**How do I tell Intel Advisor tools where I might add parallelism?** Modify your sources to add Intel Advisor annotations to identify the best opportunities for parallel task(s) within the enclosing parallel site.

**Who modifies the source code?** Because you understand your program, you modify its sources. Intel Advisor provides tools to simplify adding annotations and including the annotation definitions file. Later, you will modify sources as you transition your serial program into a parallel program.

**Now that I added site and task annotations, what’s next?** Intel Advisor provides two tools that focus on the annotated parts of your program. These tools predict (model) your program’s parallel behavior so you can experiment to find parallel sites with a good performance gain and sharing problems that you can fix:

- The Suitability tool samples your running program (Release build suggested) to determine the approximate performance of the parallel sites and tasks.
- The Dependencies tool watches your running program (Debug build recommended) in minute detail to predict and locate potential data sharing problems. Intel recommends using a reduced, representative data set. Use the Dependencies Report to explore and understand the problems. Modify your sources so your parallel code will execute correctly.

**My parallel site has a good performance gain and I fixed the data sharing problems. What’s next?**

After you modify sources, run the Suitability and Dependencies tools again. If you agree with the data displayed in the Summary window, you can replace annotations with parallel framework code. After you select a parallel framework, such as Intel® Threading Building Blocks or OpenMP®, use the Annotation Report window to quickly locate each annotation in your sources and replace it with parallel framework code. Rebuild and test your parallel program.

**See Also**
Annotations
Troubleshooting No Annotations Found
Target Applications
Survey Analysis
Annotating Code for Deeper Analysis

**Suitability Tool Overview**

The Suitability tool examines your running serial program to provide approximate estimated performance characteristics of your annotated parallel sites. This shows you both the performance gain from running your parallel program on multiple CPUs and the likely impact of parallel overhead.
To choose the best places to add parallelism, locate the parallel sites that contribute the most to the overall program’s gain. Because of the overhead of parallel execution - such as starting threads - certain parallel sites and tasks may not contribute to the overall program’s gain, or may slow down its performance. After you identify such parallel sites or tasks that do not improve performance, either modify or eliminate their annotations.

Using the Suitability Report Window

After you run the Suitability tool, view its data in the **Suitability Report** window. This window contains multiple areas:

<table>
<thead>
<tr>
<th>Location in Window</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Upper</td>
<td>Any annotation-related error the Suitability tool detects appears at the top of the <strong>Suitability Report</strong> window. If you see such errors, the displayed Suitability data may not be reliable. To view the source location associated with an error, click the <strong>View Source</strong> button. To fix the error, read the displayed error message, modify your source code to fix the problem, rebuild your target executable, and run Suitability tool analysis again.</td>
</tr>
<tr>
<td>Upper-left</td>
<td>The upper-left area shows the <strong>Maximum Program Gain for All Sites</strong> in the program. Your overall goal of adding parallelism is to increase the <strong>Maximum Program Gain for All Sites</strong> so the parallel program will execute as fast as possible. The measured serial execution runtime, predicted parallel runtime, and any measured paused time are displayed below <strong>Maximum Program Gain for All Sites</strong>. Use the predicted Suitability gain values to help you make informed decisions about where to add parallelism.</td>
</tr>
<tr>
<td>Upper-right</td>
<td>Use the upper-right row of modeling parameters to model performance. Choose a hardware configuration and threading model (parallel framework) values from the drop-down lists. If you select a <strong>Target System</strong> for Intel® Xeon Phi™ processors, an additional value for total <strong>Coprocessor Threads</strong> appears. Below this row is a grid of data that shows the estimated performance of each parallel site detected during program execution. The <strong>Site Label</strong> shows the argument to the site annotation. Examine the predicted <strong>Site Gain</strong> and <strong>Impact to Program Gain</strong> (higher values are better) to estimate how much each site contributes to the <strong>Maximum Program Gain for All Sites</strong> for all sites (described above). To expand the data under <strong>Combined Site Metrics</strong> or <strong>Site Instance Metrics</strong>, click the **icon to the right of that heading; to collapse data, click ☞ to the right of that heading.</td>
</tr>
<tr>
<td>Middle-left</td>
<td>If you choose a <strong>Target System</strong> of CPU, to view detailed characteristics of the selected site as well as its tasks and locks, click the <strong>Site Details</strong> tab. The <strong>Scalability of Max Site Gain</strong> graph summarizes performance for the selected site. The number of CPU processors or total number of coprocessor threads appears on the horizontal X axis and the target’s predicted performance gain appears on the Y axis. To change the default <strong>CPU Count</strong> and the <strong>Maximum CPU Count</strong>, set the Options value.</td>
</tr>
<tr>
<td>Lower-left</td>
<td>Below the graph is a list of issues that might be preventing better <strong>predicted</strong> performance gains as well as a summary of serial and predicted parallel time. To expand a line, click the down arrow to the right of the item’s name. Most issues are related to the <strong>Runtime Modeling</strong> modeling parameters. Later, you can use other Analyzer tools like Intel® VTune™Profiler to measure <strong>actual</strong> performance of your parallel program.</td>
</tr>
<tr>
<td>Location in Window</td>
<td>Description</td>
</tr>
<tr>
<td>--------------------</td>
<td>-------------</td>
</tr>
<tr>
<td>Lower-middle</td>
<td>Use the <strong>Loop Iterations (Tasks) Modeling</strong> (or <strong>Tasks Modeling</strong>) modeling parameters to experiment with different loop structures, iteration counts, and instance durations that might improve the predicted parallel performance. Click <strong>Apply</strong> to view the impact on the predicted performance.</td>
</tr>
<tr>
<td>Lower-right</td>
<td>Use the <strong>Runtime Modeling</strong> modeling parameters to learn which parallel overhead categories might have an impact on parallel overhead. If you agree to address a category later by using the chosen parallel framework’s capabilities or by tuning the parallel code after you have implemented parallelism, check that category.</td>
</tr>
<tr>
<td>Bottom-right</td>
<td>If the chosen <strong>Target System</strong> is <strong>Intel Xeon Phi</strong> or <strong>Offload to Intel Xeon Phi</strong>, additional Intel® Xeon Phi™ Advanced Modeling options appear below the <strong>Runtime Modeling</strong> area. To expand this area, click the down arrow to the right of <strong>Intel Xeon Phi Advanced Modeling</strong>. If you chose a <strong>Target System</strong> of <strong>CPU</strong>, the <strong>Site Details</strong> tab shows details about the selected parallel site, as well as details for each task and lock executed in that site.</td>
</tr>
</tbody>
</table>

When using an active result (not a read-only result), you can change the modeling parameters. Changing modeling parameters updates the displayed data, except for **Loop Iterations (Tasks) Modeling** or **Tasks Modeling** (click **Apply**). These modeling parameters help you understand the sensitivity of your annotation choices so you can choose the best places to add parallelism, but the displayed data summary is not an accurate estimate of final execution time on any specific parallel hardware (general processor characteristics are used).

Later, before you add parallel code, you must choose one parallel framework (threading model) for your application.

To view the source code associated with a site, locate the list of sites (upper-right area) and either:

- Double-click a row (or right-click and select **View Source** from the context menu) to display the **Suitability Source** window. Later, to return to the **Suitability Report** window, click **Suitability Report**.
- Right-click a row and select **Edit Source** from the context menu to display the corresponding source file in a code editor. When using the Intel® Advisor GUI on Linux* OS, the editor defined by the Options > Editor dialog box appears with the file open at the corresponding location. When using the Intel® Advisor GUI on Windows* OS, the file type association (or Open With dialog box) determines the editor used. When using Microsoft Visual Studio®, the Visual Studio code editor appears with the file open at the corresponding location. Later, to return to the **Suitability Report** or **Suitability Source** window:
  1. Click the **Result** tab.
  2. Click either **Suitability Report** or **Suitability Source**.

**Using the Suitability Source Window**

Within the **Suitability Source** window, you can:

- Use the **Call Stack** pane to view different source locations in the call stack.
- Double-click a line (or right-click and select **Edit Source**) to open the corresponding source file in a code editor. When using the Intel® Advisor GUI on Linux* OS, the editor defined by the Options > Editor dialog box appears with the file open at the corresponding location. When using the Intel Advisor GUI on
Windows* OS, the file type association (or Open With dialog box) determines the editor used. When using Microsoft Visual Studio*, the Visual Studio code editor appears with the file open at the corresponding location. Later, to return to the Result tab, click Result.

• Return to the Suitability Report window by clicking Suitability Report.

The Suitability Report, Suitability Source, and other Intel Advisor windows appear within the Result tab. There is one Result tab for each project.

Understanding the Scalability Graph in the Suitability Report

One of two different graphs appear depending on the chosen Target System. For an explanation of the Scalability Graph, see Suitability Report Overview.

Tips on Understanding the Performance Data

In the Suitability Report window, you start at the top, select a site, look at its details in the Suitability Report window, and examine its source code. You repeat this process to investigate each annotated site. View this information, and if needed, modify the annotations by using your code editor.

Use the following guidelines to evaluate the feasibility of each site:

• If the Site Gain values for the selected site shows an estimated performance gain of 1.0 or less, the overhead of parallel thread execution exceeds the potential performance gains. Modify or remove the annotations for the task(s) and its enclosing site. Repeat this for each parallel site.

• If the Site Gain values for the selected site shows a performance gain greater than 1.0, look at the site's contribution to the Maximum Program Gain for All Sites, which applies to all parallel sites. For sites that do not contribute significantly to the Maximum Program Gain for All Sites, modify or remove the annotations for the task(s) and its enclosing site. For sites that only contribute slightly to the Maximum Program Gain for All Sites, examine more closely the annotations and the assumptions about fixing the various overhead costs of parallel thread execution. In some cases, you may be able to adjust the annotations to improve the performance gain or reduce the overhead. Repeat this for each parallel site.

• When the Maximum Program Gain for All Sites for all sites and the Site Gain values for all the sites show a moderate or significant performance gain, proceed to the next workflow step that uses the Dependencies tool to check your remaining annotated sites for data sharing problems.

Minimizing Data Collection, Result Size, and Execution Time (Suitability Report)

For medium-large targets, several methods are available to minimize the amount of data collected and target execution time. Minimizing the data collected reduces the amount of data you need to examine in the Suitability Report; it also reduces the size of the generated result.

The Suitability analysis recognizes many annotations, including:

• Pause Collection: Stops data collection to skip uninteresting parts of the target program's execution. This minimizes the data collected, speeds up the analysis of medium-large applications, and minimizes execution time.

• Resume Collection: Resumes data collection previously paused to collect data about the interesting parts of your program.

The Survey and Suitability tools provide several ways to start target execution with data collection paused or paused for a specified time: from the GUI, using an Intel Advisor Option, or a command-line option. Starting collection paused for a specified time lets you automatically skip data/program initialization and related setup steps without modifying your source code. Starting collection paused (without a time) lets you automatically skip data/program initialization and related setup steps but your target needs to execute a Resume Collection annotation.

The Survey and Suitability tools also provide I and (collection) buttons under Check Suitability in the Threading Advisor Workflow tab that are equivalent to executing the Pause Collection and Resume Collection annotations. You might use these GUI buttons instead of the annotations in certain cases, such as your initial profiling runs.
The recommended method is to:

- Start the tool and target execution using one of the following:
  - Click the button under Check Suitability in the Threading Advisor Workflow.
  - Before you run the tool, set the equivalent Project Property in the Intel Advisor GUI under Advanced Resume collection after, ms to start collection paused for a specified time.
  - On the command line, use --collect with either the --start-paused or --resume-after options.
  - If you used the button or --start-paused option, execute the Resume Collection annotation to begin data collection of the first interesting part of the program.
  - Execute pairs of Pause Collection and Resume Collection annotations to collect data about the interesting part(s) of the target program. Place Pause Collection and Resume Collection annotations outside the parallel site begin and parallel site end annotations.

When using the Intel Advisor GUI, you can specify Project Properties for the Sampling interval and the Duration time estimate, which are Advanced options. You should provide a Duration time estimate if the target's execution takes multiple hours, because this can increase the Sampling interval used during data collection to minimize result size. You can also directly adjust the Sampling interval project property:

- To minimize excessive result size, increase the Sampling interval.
- To increase the accuracy of collected data (and result size), reduce the Sampling interval.

See Also

Before Running the Suitability Tool
Pause Collection and Resume Collection Annotations
Analysis Target Tab
advixe-cl Option Reference
resume-after option Resume collection after the specified number of milliseconds.
start-paused option Start executing the target application for analysis purposes, but delay data collection.

Before Running the Suitability Tool

The Suitability tool runs your annotated serial program's executable and measures it to predict its approximate parallel performance. Use the Suitability data to help you choose the best places to add parallelism.

**NOTE** The Suitability tool is available in the Threading Workflow only. Use the button to switch to the Threading Workflow.

Before you run the Suitability tool, do the following:

- Use the Survey tool (previous workflow step) to locate several areas where your program spends it time.
  - If you recently used a different profiling tool to measure your program's hotspots, this step is optional.
- Add Intel Advisor annotations into your sources to mark at least one parallel site and its associated task(s) (previous workflow step). This tool recognizes site, task, and lock annotations as it executes your serial program to predict its parallel behavior.
- Build a target executable using a Release build. For best results, the target must include debug information.

To run the Suitability tool, do one of the following:
• In the **Threading Workflow** tab below **Check Suitability** click the **Collect** button or `Ctrl+C`. The `Ctrl+C` button starts executing your target with data collection disabled. Data collection can be enabled either by a Resume Collection annotation or by clicking the `Ctrl+R` button (see the help topic *While Running the Suitability Tool*).
• Click the `Ctrl+I` icon in the Intel Advisor toolbar or the **Tools > Intel Advisor [version] Project Properties...** menu.

The Suitability tool needs to collect and analyze data about your running program's annotated parallel sites and tasks, so running the Suitability tool will take longer than running your program.

After you change annotations or fix data sharing problems reported by the Dependencies tool, run the Suitability tool again to revise the predicted parallel performance.

You can set the default value for the **CPU Count** in the **Options > General** dialog box. Use this dialog box to also choose the **Maximum CPU Count**, which specifies the X axis values in the scalability graph and limits the maximum value allowed for the CPU Count in the Suitability Report window.

Tip
For the most current information on optimal C/C++ and Fortran build settings, see *User Guide: Before You Begin*.

See Also
While Running the Suitability Tool
Target Applications
Minimizing Data Collection, Result Size, and Execution Time (Suitability Report)
Intel Advisor Menu Items and Toolbars
Survey Analysis
General Tab

**While Running the Suitability Tool**

The Suitability tool runs and examines your running serial program's executable target to provide approximate performance characteristics of your annotated parallel sites. This data helps you predict both the performance gain from running your parallel program on multiple cores and the likely impact of parallel overhead. After you change annotations or fix data sharing problems reported by the Dependencies tool, run the Suitability tool again to revise the program's predicted performance.

While the Suitability tool is running your program:

• Analysis data is collected as the program runs. Non-GUI application output appears in the **Application Output** pane. A log of the collection messages appears in the **Collection Log** pane.
• When data collection completes, data finalization prepares the data for display.
• After finalization completes, the data appears in the **Suitability Report** window. Any annotation-related error the Suitability tool detects appears at the top of the **Suitability Report** window. If you see such errors, the displayed Suitability data may not be reliable. To view the source location associated with an error, click the **View Source** button. To fix the error, read the displayed error message, modify your source code to fix the problem, rebuild your target executable, and run Suitability tool analysis again.
• Updated information appears in the **Summary** window, including the collection times and links that allow you to view the application output and collection output messages.

During data collection, you can temporarily pause and later resume data collection. To manually skip uninteresting parts of your target while the target continues to run, you can:

• Temporarily stop (pause) data collection. To do this, click the `Pause` button under **3. Check Suitability** in the **Threading Workflow** tab.
• Resume data collection that you previously paused. To resume collection, click the button under 3. Check Suitability in the Threading Workflow tab.

Instead of manually clicking the and buttons, you can add the Pause Collection and Resume Collection annotations around uninteresting source code region(s). These annotations are recognized by all three Intel Advisor tools.

If you need to stop or cancel the collection, do one of the following:

• Stop data collection and retain the collected data. To do this, click the button under 3. Check Suitability in the Threading Workflow tab. The tool will finalize any data that has been collected and display it in the Suitability Report.

• Cancel data collection and discard the collected data. To do this, click the button under 3. Check Suitability in the Threading Workflow tab. The tool will discard all the data that has just been collected and instead display the previously collected data in the Suitability Report.

• To interrupt data finalization, click the button.

Once data collection and analysis are complete, data appears in the Suitability Report window. In addition to the Suitability Report window, you can view details in the Suitability Source window.

**Suitability Report Overview**

After the Suitability tool runs your program's target executable to collect data, the Suitability Report window appears. It displays the approximate predicted performance based on its analysis of the annotated parallel sites and tasks.

![Suitability Report](image)

This screen shows data based on a Target System of CPU. The screen shown on your system will differ.

1. The upper-left area shows the Maximum Program Gain for All Sites in the program. Your overall goal of adding parallelism is to increase the Maximum Program Gain for All Sites so the parallel program will execute as fast as possible. The measured serial execution runtime, predicted parallel runtime, and any measured paused time are displayed below Maximum Program Gain for All Sites. Use the predicted Suitability gain values to help you make informed decisions about where to add parallelism.
If the Suitability tool detects any annotation-related errors, they appear at the top of the Suitability Report window. If you see this type of error, the displayed Suitability data may not be reliable. Annotation-related errors may be caused when the correct sequence of annotations do not occur because of missing annotations, when unexpected execution paths occur, or if Suitability data collection was paused while the target was executing.

2. Use the upper-right row of modeling parameters to model performance. Choose a hardware configuration and threading model (parallel framework) values from the drop-down lists. If you select a Target System for Intel® Xeon Phi™ processors, an additional value for total Coprocessor Threads appears.

Below this row is a grid of data that shows the estimated performance of each parallel site detected during program execution. The Site Label shows the argument to the site annotation. Examine the predicted Site Gain and Impact to Program Gain (higher values are better) to estimate how much each site contributes to the Maximum Program Gain for All Sites for all sites (described above). To expand the data under Combined Site Metrics or Site Instance Metrics, click the ▷ icon to the right of that heading; to collapse data, click ◄ to the right of that heading.

To view source code for a selected parallel site, click its row to display the Suitability Source window.

To show or hide the side command toolbar, click the or icon.

3. The Scalability of Maximum Site Gain graph summarizes performance for the selected site. The number of CPU processors or total number of coprocessor threads appears on the horizontal X axis and the target's predicted performance gain appears on the Y axis. To change the default CPU Count and the Maximum CPU Count, set the Options value.

If you choose a Target System of CPU, to view detailed characteristics of the selected site as well as its tasks and locks, click the Site Details tab.

4. Use the Loop Iterations (Tasks) Modeling (or Tasks Modeling) modeling parameters to experiment with different loop structures, iteration counts, and instance durations that might improve the predicted parallel performance.

For example, you might want to see the impact of modifying your nested change loop structure, modify the loop body code, or change number of iterations.

If the task annotations indicate likely task parallelism, the title will appear as Task Modeling (instead of Loop Iterations (Task) Modeling for data parallelism).

5. Use the Runtime Modeling modeling parameters to learn which parallel overhead categories might have an impact on parallel overhead. If you agree to address a category later by using the chosen parallel framework's capabilities or by tuning the parallel code after you have implemented parallelism, check that category.

If the chosen Target System is Intel Xeon Phi or Offload to Intel Xeon Phi, additional Intel® Xeon Phi™ Advanced Modeling options appear below the Runtime Modeling area. To expand this area, click the down arrow to the right of Intel Xeon Phi Advanced Modeling.

6. Below the graph is a list of issues that might be preventing better predicted performance gains as well as a summary of serial and predicted parallel time. To expand a line, click the down arrow to the right of the item's name. Most issues are related to the Runtime Modeling modeling parameters. Later, you can use other Analyzer tools like Intel® VTune™ Profiler to measure actual performance of your parallel program.
Target System Hardware Configurations

The Target System lets you select the type of hardware configuration to be analyzed. From this drop-down list, you can check each type to learn the likely predicted performance characteristics for each:

- **CPU** shows the predicted performance of only the CPU. Choose this item for Intel® Xeon® or similar processors that do not have significant parallel coprocessors. For an Intel® Xeon Phi™ processor, choose this setting to only model the host processor, such as an Intel Xeon processor. If you choose this configuration, you can specify the **CPU Count** modeling parameter.

- **Intel Xeon Phi** shows the predicted performance when using only the Intel Xeon Phi coprocessor cores, and not the host processor. This parameter does not account for data exchange amongst Intel Xeon Phi coprocessor cores and the host CPU. If you choose this configuration, you can specify the **Coprocessor Threads** modeling parameter.

- **Offload to Intel Xeon Phi** shows the predicted performance when using Intel Xeon Phi coprocessor manycores to execute parallel code after the host CPU starts the program and before execution resumes on the host CPU for program completion. If you choose this configuration, you can specify the **Coprocessor Threads** and **CPU Count** modeling parameters.

Data Displayed When the Target System is Intel® Xeon Phi™

A sample screen below shows changes in orange boxes when the Target System is Intel Xeon Phi (instead of CPU).

- The displayed data changes, such as the **Maximum Program Gain for All Sites** and the serial and predicted parallel time.
- The graph’s appearance changes to a gray-green color and the X axis displays **Coprocessor Threads** (instead of **CPU Count**) to represent the predicted performance of the manycore parallel coprocessor. This graph shows the predicted parallel performance of the manycore parallel coprocessor without accounting for data exchange amongst Intel Xeon Phi coprocessor cores and the host CPU. For many applications, the number of task instances does not scale enough to fully utilize the many cores of the parallel coprocessor, as indicated by a hover tip. Applications that are not appropriate for an Intel Xeon Phi processing system have values that appears in the gray part of the graph; in this case, try modeling other types of the Target System.
• The lines between the graph's gray and green areas is a reference baseline, where the reference CPU chosen to calculate the Intel Xeon processor peak baseline is a dual-socket 8-core Intel Xeon processor E5-26xx product family (2.7 GHz, 16 cores total). When the Maximum Site Gain exceeds this baseline, you might consider using an Intel Xeon Phi coprocessor rather than an Intel Xeon or similar processor.

When the Target System is either Intel Xeon Phi or Offload to Intel Xeon Phi, the Intel Xeon Phi Advanced Modeling options appear. See Intel® Xeon Phi™ Advanced Modeling.

Data and Modeling Parameters When the Target System is Offload to Intel Xeon Phi

A sample screen below shows changes in orange boxes when the Target System is Offload to Intel Xeon Phi (instead of CPU) and the Offload to Intel Xeon Phi column is selected.

When you select a Target System of Offload to Intel Xeon Phi coprocessor:
• The displayed data changes, such as the Maximum Program Gain for All Sites and the serial and predicted parallel time.
• An additional modeling parameter appears as a new column for each site named Offload to Intel Xeon Phi. If selected, the Scalability of Maximum Site Gain graph displays Coprocessor Threads on the X axis. If unselected, the graph displays CPU Count on the X axis.
• In the upper-right corner, an additional modeling parameter appears. That is, both the total number of Coprocessor Threads and the CPU Count appear because both the number of CPUs and the coprocessor's total number of hardware threads should be considered to predict parallel execution.
• Additional modeling parameters appear below Runtime Modeling area under Intel Xeon Phi Advanced Modeling - see Intel® Xeon Phi™ Advanced Modeling.
• When the column named Offload to Intel Xeon Phi is selected, the graph's appearance changes to a gray-green color and the X axis displays Coprocessor Threads instead of CPU Count. This graph shows the predicted performance of the manycore parallel coprocessor and its host CPUs. For many applications, the number of task instances does not scale enough to fully utilize the many cores of the parallel...
coprocessor, as indicated by a hover tip. Applications that are not appropriate for an Intel Xeon Phi processing system have values that appear in the gray part of the graph; in this case, try modeling other types of the Target System. Applications that are appropriate for offload to an Intel Xeon Phi processing system have values that appear in the green part of the graph.

The lines between the graph’s gray and green areas is a reference baseline, where the reference CPU chosen to calculate the Intel Xeon processor peak baseline is a dual-socket 8-core Intel Xeon processor E5-26xx product family (2.7 GHz, 16 cores total). When the Maximum Site Gain exceeds this baseline, you might consider using an Intel Xeon Phi coprocessor rather than an Intel Xeon or similar processor.

Site Details Tab

If you chose a Target System of CPU, after you click the Site Details tab (next to Site Performance Scalability), the lower part of the Suitability Report shows details about the selected site, as well as details about each task and lock within that site.

Choosing Modeling Parameters in the Suitability Report

The Suitability Report lets you adjust modeling parameters based on possible application needs. When using an active result, you can adjust modeling parameters and quickly view the likely impact on the predicted performance interactively.
The top row of modeling parameters provides drop-down lists that let you define the likely hardware configuration of target systems as well as the high-level parallel framework. These values let you predict the likely performance characteristics for the selected parallel site.

- **Use the Target System to select the type of hardware configuration to be analyzed:** CPU, Intel Xeon Phi, or Offload to Intel Xeon Phi. The latter two apply to the Intel Xeon Phi processor system.
- **Use the Threading Model** to choose the high-level parallel framework to be used, such as OpenMP* or Intel® Threading Building Blocks (Intel® TBB).
- **Use CPU Count** to specify the number of CPUs to model. To specify the default CPU count by setting the Options value.
- **If you choose a Target System of Intel Xeon Phi, or Offload to Intel Xeon Phi,** use the **Coprocessor Threads** to choose the number of Intel Xeon Phi coprocessor threads.

As you modify these modeling parameters, the predicted performance estimates are updated automatically. Repeat as needed.

If your target app contains multiple parallel sites, select each parallel site you wish to examine. When you select a different parallel site, the predicted performance estimates for that site are updated automatically. Repeat as needed for each site.

Use the **Loop Iterations (Tasks) Modeling** or **Tasks Modeling** area to view the impact of changing the number of iterations and the iteration duration on the predicted performance for the selected parallel site (the label displayed depends on whether iteration loop annotations or general task annotations were detected). For example, you might want to see the impact of modifying your nested change loop structure, modify the loop body code, or change number of iterations. After you slide the **Avg. Number of Iterations (Tasks):** or **Avg. Number of Tasks:** and the **Avg. Duration** values, click the **Apply** button to view the predicted performance estimates. Repeat as needed.

Use the **Runtime Modeling** area to view the predicted impact of adjusting run-time parallel characteristics **after** you add parallelism for the selected parallel site, including using parallel framework capabilities to minimize parallel overhead or tuning your parallel code.

If you agree to later check and modify runtime performance aspects for a category, check the box to the left of that category name. For example, you can also examine and tune actual parallel code performance characteristics using tools like Intel® VTune™ Profiler and implement the runtime capabilities of high-level parallel frameworks to limit parallel overhead, such as task chunking. As you check or uncheck different categories, the predicted performance estimates are updated automatically. Repeat as needed.

If you choose **Target System** as **Intel Xeon Phi** or **Offload to Intel Xeon Phi**, additional **Intel Xeon Phi Advanced Modeling** options (not shown) appear below **Runtime Modeling** (see Intel® Xeon Phi™ Advanced Modeling).

**NOTE**
The Intel® Advisor Suitability tool predicts the general performance characteristics of CPUs and Intel Xeon Phi coprocessors. For example, it does not consider CPU clock frequency, cache characteristics, versions of processors, and so on.

**See Also**
Suitability Tool Overview
Suitability Report Overview
Intel® Xeon Phi™ Advanced Modeling
Fixing Annotation-related Errors Detected by the Suitability Tool

Fixing Annotation-related Errors Detected by the Suitability Tool

As the Suitability tool executes your target executable, it scans for a proper sequence of Intel® Advisor annotations. If it detects an annotation-related or an error related to very small task sizes, it displays a message at the top of the Suitability Report window.

If you see such messages, investigate the cause and fix the error. The messages displayed are generally self-explanatory.

After you modify your source code and rebuild your application, run the Suitability tool again. When no errors appear near the top of the Suitability Report window, you can carefully examine the Suitability data to help you make decisions about the proposed parallel sites and tasks.

Tools to Help You Fix Annotation Errors

Use the Suitability Source window to view source code related to a specific site or task. You can also use the Annotation Report window to view a list of your annotations and display their code snippets.

When resolving annotation-related errors, consider the execution paths your program follows. If necessary, investigate the execution paths using a debugger.

In addition to annotation sequence messages, messages about task size may also appear. For example, if the CPU time used by a task per loop cycle is so small that it does not exceed the task overhead time, consider modifying the task annotation(s) after you examine the loop structure. In some cases, the message may suggest that you use a different type of task annotation (see the help topics under See Also below).

Proper Sequence of Annotations

The rules about a proper sequence of Intel® Advisor annotations include the following:

- **Sites**: A site-begin annotation is followed by annotations that mark one or more tasks. It is eventually terminated by a site-end annotation. For example, if a site-begin annotation is not followed by a task annotation or is not terminated by a site-end annotation, an error occurs.
- **Tasks**: A task may be marked either with one iterative-task annotation or a pair of task-begin and task-end annotations. When used, an iterative-task annotation must be the only task within a site. Only a task-begin and task-end pair allows task nesting.
- **Locks**: A lock-acquire annotation must be immediately followed by a lock-release annotation, and must occur within a task.

See Also

Reducing Parallel Overhead, Lock Contention, and Enabling Chunking
Annotation Types Summary
Task Organization and Annotations
Troubleshooting Sources Not Available
Troubleshooting Debug Information Not Available
Site and Task Annotations for Simple Loops With One Task
Site and Task Annotations for Loops with Multiple Tasks

Intel® Xeon Phi™ Advanced Modeling Options

When you select a Target System of Intel® Xeon Phi™ or Offload to Intel Xeon Phi coprocessor, additional modeling parameters appear below Runtime Modeling area under Intel Xeon Phi Advanced Modeling:
• Select **Consider Code Vectorization** if you agree to modify your parallel code later to improve vector parallel execution. If checked, you can specify:

  • **Reference CPU Vectorization Speedup** you expect can be achieved. This value indicates the speedup multiplier gain for the current site by using vectorization techniques with the reference CPU (dual-socket 8-core Intel® Xeon® processor E5-26xx product family at 2.7 GHz, 16 cores total). When providing this estimate, base your estimates on target device characteristics and your expertise of how much and how well this part of code can be vectorized.

  • **Intel Xeon Phi Vectorization Speedup** you expect can be achieved. This value indicates the speedup multiplier gain for current site by using vectorization techniques with an Intel® Xeon Phi™ processor. When providing this estimate, base your estimates on target device characteristics and your expertise of how much and how well this part of code can be vectorized.

  • When you choose **Target System** as Offload to Intel Xeon Phi, you can select the **Offload Transfer Data Size** to specify data transfer size value you expect can be achieved (unit is KB).

  • **Click Apply** after modifying any of these values.

In some cases, you can restructure your code to enable more efficient vector operations. Loop vectorization allows hardware to process data independently in smaller units (usually 64-byte), such as operations on data arrays.

One way to enable more efficient vector operations is to modify a **single** loop to create a new outer loop where the two loops cover the same iteration space. A technique called strip-mining allows the innermost loop to use vector operations in small chunks.

Other ways to enable more efficient vector operations include examining outermost loops where threading parallelism might already be used, and consider vectorizing its innermost loops and/or callee functions.

Certain innermost loops may benefit from OpenMP 4 constructs. That is, under certain conditions you can use both an `omp parallel for` threading pragma and a `omp simd` (or similar) simd vectorization pragma (see the compiler vectorization report and descriptions at http://openmp.org).

The processor microarchitecture determines the type of vector instructions that will be supported and thus the size of data the hardware can process efficiently (see http://en.wikipedia.org/wiki/List_of_Intel_CPU_microarchitectures).

For a description of the Intel® Xeon Phi™ coprocessor architecture, visit the Intel® Developer Zone and read such articles as https://software.intel.com/content/www/us/en/develop/articles/intel-xeon-phi-coprocessor-codename-knights-corner.html.

**See Also**

Dependencies Analysis
Suitability Tool Overview
Suitability Report Overview

**Suitability Report**

To access this window in the Result tab, click the **Suitability Report** tab, or run the Suitability tool.

This is the starting point for viewing the annotated sites found by the Intel® Advisor Suitability tool. Use this window to review the parallel sites in the upper right area. Select a site and view its annotations and related characteristics. Use the list of sites as a to-do list: start at the top and work your way down.

**Suitability Report Layout**

1. Workflow Tab
2. Result Tab
3. The main part of the **Suitability Report** window, which displays a summary presentation of the collected data. It also provides modeling parameters you can use to customize the predicted parallel performance based on your program’s target hardware configuration, parallel framework, as well experiment with refactoring loop characteristics.
<table>
<thead>
<tr>
<th>Use This</th>
<th>To Do This</th>
</tr>
</thead>
<tbody>
<tr>
<td>Workflow Tab</td>
<td>Run a tool of your choice and see results in the <strong>Result</strong> tab.</td>
</tr>
<tr>
<td>Result Tab</td>
<td>Select between available reports.</td>
</tr>
</tbody>
</table>

**Upper part of the Suitability Report window**

Any annotation-related error the Suitability tool detects appears at the top of the **Suitability Report** window. If you see such errors, the displayed Suitability data may not be reliable. To view the source location associated with an error, click the *View Source* button. To fix the error, read the displayed error message, modify your source code to fix the problem, rebuild your target executable, and run Suitability tool analysis again.

**Upper-left part of the Suitability Report window**

View the **Maximum Program Gain for All Sites** in the program. Your overall goal of adding parallelism is to increase the **Maximum Program Gain for All Sites** so the parallel program will execute as fast as possible. The measured serial execution runtime, predicted parallel runtime, and measured paused time are displayed below **Maximum Program Gain for All Sites**.

**Upper-right**

Use the upper-right row of modeling parameters to model performance. Choose a hardware configuration and threading model (parallel framework) values from the drop-down lists. If you select a **Target System** for Intel® Xeon Phi™ processors, an additional value for total **Coprocessor Threads** appears.

Below this row is a grid of data that shows the estimated performance of each parallel site detected during program execution. The **Site Label** shows the argument to the site annotation. Examine the predicted **Site Gain** and **Impact to Program Gain** (higher values are better) to estimate how much each site contributes to the **Maximum Program Gain for All Sites** for all sites (described above). To expand the data under **Combined Site Metrics** or **Site Instance Metrics**, click the ☰ icon to the right of that heading; to collapse data, click ☰ to the right of that heading.

To view the source associated with a site in the **Suitability Source** window, double-click its name (or right-click and select View Source from the context menu). To edit the source in your code editor, use the **Edit Source** context menu item.

**Middle-left above the graph**

If you choose a **Target System** of CPU, to view detailed characteristics of the selected site as well as its tasks and locks, click the **Site Details** tab.

**Graph in the lower-left**

The **Scalability of Maximum Site Gain** graph summarizes performance for the selected site. The number of CPU processors or total number of coprocessor threads appears on the horizontal X axis and the target's predicted performance gain appears on the Y axis. To change the default **CPU Count** and the **Maximum CPU Count**, set the Options value.

If you select a **Target System** of Intel Xeon Phi or Offload to Intel Xeon Phi coprocessor (and select the column **Offload to Intel Xeon Phi**), the graph's appearance changes to a gray-green color and the X axis displays...
Use This | To Do This
---|---
**Coprocessor Threads** instead of **CPU Count**. This graph represents the predicted performance of the many-core Intel® Xeon Phi™ parallel coprocessor. For many applications, the number of task instances does not scale enough to fully utilize the many cores of the parallel coprocessor, as indicated by a hover tip. Applications that are not appropriate for a Intel Xeon Phi processing system have values that appear in the gray part of the graph; in this case, consider modeling other types of the **Target System**. Applications that are appropriate for offload to an Intel Xeon Phi processing system have values that appear in the green part of the graph.

The lines between the graph's gray and green areas is a reference baseline, where the reference CPU chosen to calculate the Intel® Xeon® processor peak baseline is a dual-socket 8-core Intel Xeon processor E5-26xx product family (2.7 GHz, 16 cores total). When the **Maximum Site Gain** exceeds this baseline, you might consider using an Intel Xeon Phi coprocessor rather than an Intel Xeon or similar processor.

Near the top of the vertical lines for each CPU number, a box and a circle indicate the minimum and maximum predicted gain values. The color shading indicates the predicted benefit. For example, if the minimum-maximum range appears in the red-shaded area, this site is hurting your program's performance (and power use), so you should significantly modify or remove the site and task annotations.

**Lower-left**

Below the graph is a list of issues that might be preventing better *predicted* performance gains as a summary of serial and predicted parallel time. To expand a line, click the down arrow to the right of the item's name. Most issues are related to the **Runtime Modeling** modeling parameters. Later, you can use other Analyzer tools like Intel® VTune™ Profiler to measure actual performance of your parallel program.

**Lower-middle under Loop Iterations (Tasks) Modeling or Tasks Modeling**

Use the **Loop Iterations (Tasks) Modeling** (or **Tasks Modeling**) modeling parameters to experiment with different loop structures, iteration counts, and instance durations that might improve the predicted parallel performance.

**Lower-right under Runtime impact for this site**

Use the **Runtime Modeling** modeling parameters to learn which parallel overhead categories might have an impact on parallel overhead. If you agree to address a category later by using the chosen parallel framework's capabilities or by tuning the parallel code after you have implemented parallelism, check that category.

**Bottom-right**

If the chosen **Target System** is **Intel Xeon Phi** or **Offload to Intel Xeon Phi**, additional Intel® Xeon Phi™ Advanced Modeling options appear below the **Runtime Modeling** area. To expand this area, click the down arrow to the right of **Intel Xeon Phi Advanced Modeling**.

---

**Suitability Report, Scalability Graph**

View a scalability graph showing the predicted approximate performance characteristics of the selected site. The Scalability graph is automatically updated if you change modeling assumptions in the Suitability Report.

Two types of graphs appear depending upon the selected **Target System**. The X axis of the graph shows either **CPU Count** to model CPU usage or **Coprocessor Threads** to model Intel® Xeon Phi™ coprocessor usage.
Scalability Graph for CPU Count Usage

If you select a **Target System** of **CPU** or select **Offload to Intel Xeon Phi** and uncheck the column **Offload to Intel Xeon Phi** for the selected site, a green-yellow-red graph appears and the X axis shows **CPU Count**.

The number of CPUs appears on the X axis and the program's predicted run-time performance gain appears on the Y axis.

Near the top of each vertical line for a CPU number, you will see a box and a circle that indicate the minimum and maximum predicted gain values. The color shading indicates the likely benefit. For example, if the minimum-maximum range appears in the red-shaded area, this site is hurting your program's performance and you should significantly modify or remove the site annotations.

If the minimum-maximum range appears in the:

- Red-shaded area, this site is hurting your program's performance. Remove or significantly modify the site annotations.
- Yellow-shaded area, this site is helping your program's performance, but maybe the gain may not be enough to justify the effort you needed to refactor and maintain your program. You should investigate how the results can be improved. Also, it might be more energy (power) efficient to increase the serial throughput using program changes or faster processors (parallel machines are designed to run at a lower clock frequency than certain serial machines).
- Green-shaded area, this indicates good results. If you can achieve this level of performance, you will likely have a significant improvement to your program. The green area also indicates energy efficient (power) parallel speedups.

If the minimum-maximum range forms a flat line, this indicates that the maximum number of CPUs specified exceeds the number of task instances.

Scalability Graph for Intel® Xeon Phi™ Coprocessor Threads Usage

If you select a **Target System** of **Intel Xeon Phi** or select **Offload to Intel Xeon Phi** and check the column **Offload to Intel Xeon Phi** for the selected site, a gray-green graph appears.

The X axis shows **Coprocessor Threads** and the program's predicted run-time performance gain appears on the Y axis.

The lines between the graph's gray and green areas is a reference baseline, where the reference CPU chosen to calculate the Intel® Xeon® processor peak baseline is a dual-socket 8-core Intel Xeon processor E5-26xx product family (2.7 GHz, 16 cores total). When the **Maximum Site Gain** exceeds this baseline, you might consider using an Intel Xeon Phi coprocessor rather than an Intel Xeon or similar processor.

Near the top of each vertical line for **Coprocessor Threads**, you will see a box and a circle that indicate the minimum and maximum predicted gain values. The color shading indicates the likely benefit depending on whether the number of predicted threads would saturate the manycore coprocessor.

This graph shows the predicted parallel performance of the manycore parallel coprocessor without accounting for data exchange amongst Intel Xeon Phi coprocessor cores and the host CPU. For many applications, the number of task instances does not scale enough to fully utilize the many cores of the parallel coprocessor, as indicated by a hover tip. Applications that are not appropriate for a Intel Xeon Phi processing system have values that appear in the gray part of the graph; in this case, try modeling other types of the **Target System**.

For Each Site, Decide Whether to Modify or Keep Annotations

Use the **Suitability Report** window to view the predicted parallel performance of each parallel site and its impact on the **Maximum Program Gain for All Sites**. For example, if a site either has a **Site Gain** of less than 1.0 or does not contribute to **Maximum Program Gain for All Sites**, modify or remove its annotations. In contrast, any site that contributes to **Maximum Program Gain for All Sites** should be kept. For most sites, carefully examine the annotations, overhead assumptions, and related code.
Within the upper-right area of the **Suitability Report** window, if multiple parallel sites were detected during execution, select a different Site row to display its details.

**Implementing Modeling Assumptions Later When Adding Parallel Code**

In the lower-right part of the Suitability Report under **Runtime Modeling**, changing a checkmark does not resolve an issue - it configures the modeling of your proposed parallel program execution. To implement the modeled improvements indicated by the check boxes, use specific parallel framework constructs in the **Add Parallel Framework** step of the workflow and consider tuning your parallel code.

**Suitability Report, Site Details, Lock Row**

If you choose a **Target System** of **CPU**, to view detailed characteristics of the selected site as well as its tasks and locks, click the **Site Details** tab. The Lock row shows the statistics for that lock. You can reduce lock contention by using a fine-grain lock when you convert to a parallel framework. Your locks are considered to be fine-grain when you have a different lock for each data-structure or data-element that you need to synchronize. You can also eliminate locks by using other synchronization techniques, such as by using private copies of shared data.

In the upper-right part of the **Suitability Report** pane, if multiple parallel sites were detected during target execution, select a different Site row to display its details.

**Implementing Modeling Assumptions Later When Adding Parallel Code**

In the lower-right part of the Suitability Report under **Runtime Modeling**, changing a checkmark does not resolve an issue - it configures the modeling of your proposed parallel program execution. To implement the modeled improvements indicated by the check boxes, use specific parallel framework constructs in the **Add Parallel Framework** step of the workflow and consider tuning your parallel code.

**Suitability Report, Site Details, Site Row**

If you choose a **Target System** of **CPU**, to view detailed characteristics of the selected site and its tasks and locks, click the **Site Details** tab. The **Selected Site** row shows the overall statistics for the Selected Site. Below the **Selected Site** row, view the Tasks and Locks executed in this Site, including the **Number of Instances** and **Total Serial Time** for the entire site and each task and lock within the site.

You can reduce Site overhead by combining multiple site execution into a single site execution, for example putting a site outside a loop instead of inside a loop (moving a loop surrounding a site to be inside of the site). Under **Runtime Modeling**, you can view the predicted site overhead values and other predicted overhead or lock contention values (see below).

**For Each Site, Decide Whether to Modify or Keep Annotations**

Use the **Suitability Report** window to view the predicted parallel performance of each parallel site and its impact on the **Maximum Program Gain for All Sites**. For example, if a site either has a **Site Gain** of less than 1.0 or does not contribute to **Maximum Program Gain for All Sites**, modify or remove its annotations. In contrast, any site that contributes to **Maximum Program Gain for All Sites** should be kept. For most sites, carefully examine the annotations, overhead assumptions, and related code.

In the upper-right part of the **Suitability Report** pane, if multiple parallel sites were detected during target execution, select a different Site row to display its details.

**Implementing Modeling Assumptions Later When Adding Parallel Code**

In the lower-right part of the Suitability Report under **Runtime Modeling**, changing a checkmark does not resolve an issue - it configures the modeling of your proposed parallel program execution. To implement the modeled improvements indicated by the check boxes, use specific parallel framework constructs in the **Add Parallel Framework** step of the workflow and consider tuning your parallel code.
Suitability Report, Site Details, Task Row

If you choose a Target System of CPU, to view detailed characteristics of the selected site and its tasks and locks, click the Site Details tab. The Task row shows the statistics for that task.

You can reduce Task overhead by combining multiple task executions into a single task execution, for example by:

- Lexically merging two tasks into one.
- Enabling chunking. Chunking means that the parallel framework merges several tasks into a single task, with little or no overhead between them. For instance, if tasks are loop iterations, chunking would execute several iterations together (as a chunk) before heavyweight task control is performed. Chunking is typically implemented when you convert to a parallel framework by using an Intel® Threading Building Blocks (Intel® TBB) parallel_for() instance, an OpenMP* C/C++ pragma #pragma omp parallel for, or the OpenMP* Fortran directive !$omp parallel do.

NOTE
If you see an Annotation Label named _too_deep_, this indicates that the call behavior of your program results in nesting of site and/or task annotations more deeply than Intel Advisor can handle. The deeper occurrences have all been merged into this _too_deep_ pseudo-site. This can indicate a problem with the placement of the annotation pair. For example, deep recursions or not executing site-end or task-end annotations can result in such nesting.

For Each Site, Decide Whether to Modify or Keep Annotations

Use the Suitability Report window to view the predicted parallel performance of each parallel site and its impact on the Maximum Program Gain for All Sites. For example, if a site either has a Site Gain of less than 1.0 or does not contribute to Maximum Program Gain for All Sites, modify or remove its annotations. In contrast, any site that contributes to Maximum Program Gain for All Sites should be kept. For most sites, carefully examine the annotations, overhead assumptions, and related code.

In the upper-right part of the Suitability Report pane, if multiple parallel sites were detected during target execution, select a different Site row to display its details.

Implementing Modeling Assumptions Later When Adding Parallel Code

In the lower-right part of the Suitability Report under Runtime Modeling, changing a checkmark does not resolve an issue - it configures the modeling of your proposed parallel program execution. To implement the modeled improvements indicated by the check boxes, use specific parallel framework constructs in the Add Parallel Framework step of the workflow and consider tuning your parallel code.

See Also

- Suitability Analysis
- Suitability Tool Overview
- Suitability Report Overview
- Choosing Modeling Parameters in the Suitability Report
- Troubleshooting Source Not Available
- Troubleshooting Debug Information Not Available
- Using Partially Parallel Programs with Intel Advisor Tools

Suitability Report, Site Row

The Site rows summarize the predicted performance improvement for each parallel site you have annotated in your application. If only one annotated parallel site was executed, only a single site row will be present.
Selecting a Site row allows you to view the characteristics of the selected parallel site and choose its modeling parameters, including the scalability graph and likely causes, modeling of tasks and task iterations, and runtime impact of overhead and contention items. Based on your current annotations, the check boxes under Runtime Modeling show what would happen if you agree to fix different types of parallel overhead items when you add parallelism.

View the upper-left corner. Your goal is to make the Site Gain as close to the CPU count as possible. Improving each site’s Site Gain will improve the combined Maximum Program Gain for All Sites.

NOTE
If you see an Annotation Label named _too_deep_, this indicates that the call behavior of your program results in nesting of site and/or task annotations more deeply than Intel Advisor can handle. The deeper occurrences have all been merged into this _too_deep_ pseudo-site. This can indicate a problem with the placement of the annotation pair. For example, deep recursions or not executing site-end or task-end annotations can result in such nesting.

For Each Site, Decide Whether to Modify or Keep Annotations

Use the Suitability Report window to view the predicted parallel performance of each parallel site and its impact on the Maximum Program Gain for All Sites. For example, if a site either has a Site Gain of less than 1.0 or does not contribute to Maximum Program Gain for All Sites, modify or remove its annotations. In contrast, any site that contributes to Maximum Program Gain for All Sites should be kept. For most sites, carefully examine the annotations, overhead assumptions, and related code.

Within the upper-right area of the Suitability Report window, if multiple parallel sites were detected during execution, select a different Site row to display its details.

Implementing Modeling Assumptions Later When Adding Parallel Code

In the lower-right part of the Suitability Report under Runtime Modeling, changing a checkmark does not resolve an issue - it configures the modeling of your proposed parallel program execution. To implement the modeled improvements indicated by the check boxes, use specific parallel framework constructs in the Add Parallel Framework step of the workflow and consider tuning your parallel code.

See Also

- Suitability Report Overview
- Suitability Tool Overview

Reducing Parallel Overhead, Lock Contention, and Enabling Chunking

The data collected and analyzed in the Suitability Report window shows data for the selected site. The text that appears below Runtime impact for this site (lower-right area) may recommend that you consider reducing several types of parallel overhead, lock contention, and enable chunking in your parallel program, as explained in Suitability Report Overview. If you agree to address a category later by using the chosen parallel framework’s capabilities or by tuning the parallel code after you have implemented parallelism, check that category.

This group of topics explain site, task, and lock overhead, lock contention, and task chunking.

Reducing Site Overhead

Site overhead is the time spent starting up (and shutting down) parallel execution. This overhead includes creating threads, scheduling those threads onto cores, and waiting for the threads to begin executing. In some parallel framework implementations, real threads are only created once - rather than destroying them at the end of a parallel site; the implementation suspends the threads. In this case, the full site overhead will be experienced only the first time a site is entered.
Site overhead is proportional to the number of times a site is executed. If you have a site that is executed too frequently or where the average time per instance is too small, you should choose a location for your site that encloses a larger amount of computation.

If the Suitability tool recommends that you reduce site overhead, the parallel site is probably too small.

To reduce site overhead, have the site do more work during its execution. You might be able to combine multiple site executions into one. For example, consider putting a site outside a loop instead of inside a loop.

### Reducing Task Overhead

Task overhead is the time spent creating a task and getting it assigned to a thread, and also the time spent stopping or pausing the thread when the task is complete.

Task overhead is proportional to the number of times a task is executed. If you have a task that is executed too frequently or where the average time per task instance is too small, modify your task so it encloses a larger amount of computation. Alternatively, consider using the task chunking feature, which is supported by several parallel frameworks. In this case, the parallel framework groups multiple task executions at run-time.

If the Suitability analysis recommends that you reduce task overhead, the parallel task is probably too small. Often this is because you have chosen an inner loop in a leaf function as the location of your parallel site, where you instead should have chosen a function farther up the call tree.

There are two ways to reduce task overhead:

- Restructure your program to reduce the number of tasks you create. For example, restructure your task annotations and/or code to increase the amount of work that occurs during each task's execution.
- If available for the selected parallel framework, enable the task chunking feature.

You can reduce task overhead by combining multiple task executions into a single task execution. For example, by merging two tasks into one.

### Reducing Lock Overhead

Lock overhead is the time spent in creating, destroying, acquiring, and releasing locks. Lock overhead does not include the time spent waiting for a lock held by another task - that is called lock contention. You can think of lock overhead as the cost of the lock operations themselves assuming the lock is always available.

If possible, restructure you code to reduce Lock overhead by creating a private copy of an object for each task to avoid the need to acquire a lock - see the help topic Problem Solving Strategies.

### Reducing Lock Contention

Lock contention is the time spent waiting in one thread for a lock to be released while another thread is holding that lock.

You can reduce Lock contention by using different locks for unrelated data when you convert to a parallel framework.

### Enabling Task Chunking

Chunking means that the parallel framework will merge several tasks into a single task, with little or no overhead between them. For instance, if tasks are loop iterations, chunking would mean that several iterations are executed together (as a chunk) before heavyweight task control is performed.

Chunking is typically implemented when you convert to a parallel framework:

- With Intel® Threading Building Blocks (Intel® TBB), by using a `parallel_for()` instance.
- With OpenMP®, by using the C/C++ pragma `#pragma omp parallel for` or the Fortran directive `!$omp parallel do`.

You can also restructure your code to enable chunking. This can be done by modifying a single loop to create a new outer loop where the two loops cover the same iteration space. A technique called strip-mining allows the inner loop to use vector operations in small chunks. Loop vectorization allows hardware to process data independently in smaller units (usually 64-byte), such as operations on data arrays.
Once these two loops exist, move the inner loop inside the task annotations so the task begin and end annotations encapsulate the inner loop. The outer loop strides by some chunk size, and the inner loop iterates sequentially through each chunk.

In cases where the CPU time and the elapsed time are about the same, the Suitability Report window under Runtime impact for this site may recommend that you enable task chunking.

If you check an item under to the right of the Scalability of Maximum Site Gain graph (such as Enable Task Chunking), its value will be added to the Site Gain and possibly the Maximum Site Gain for All Sites values.

See Also
Dependencies Analysis
Parallelize Functions - Intel® Threading Building Blocks (Intel® TBB) Tasks
Suitability Tool Overview
Reducing Task Overhead
Suitability Report Overview

Suitability Source Window

To access this window in the Suitability Report window, double-click a row (or use the View Source context menu item) to view the source code associated with a Site, Task, or Lock annotation.

Use this Suitability Source window to examine the source code and the associated call stack for each source line. To modify your source code, use your code editor.

Suitability Source Layout

<table>
<thead>
<tr>
<th></th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

1. Workflow Tab
2. Result Tab
3. File: filename pane
4. Call Stack pane

<table>
<thead>
<tr>
<th>Use This</th>
<th>To Do This</th>
</tr>
</thead>
<tbody>
<tr>
<td>Workflow Tab</td>
<td>Run a tool of your choice and see results in the Result tab.</td>
</tr>
<tr>
<td>Result Tab</td>
<td>Select between available reports.</td>
</tr>
<tr>
<td>File: filename pane</td>
<td>View the sources associated with the selected site or task annotation.</td>
</tr>
<tr>
<td></td>
<td>Double-click a source code line (or right-click and select Edit Source) to open the code editor with the corresponding source file.</td>
</tr>
<tr>
<td></td>
<td>• On Windows* OS:</td>
</tr>
<tr>
<td></td>
<td>• When using Visual Studio, the Visual Studio code editor appears with the file open at the corresponding location.</td>
</tr>
<tr>
<td></td>
<td>• When using the Intel Advisor GUI, the file type association (or Open With dialog box) determines the editor used.</td>
</tr>
</tbody>
</table>
Use This | To Do This
--- | ---
On Linux OS: When using the Intel Advisor GUI, the editor defined by the Options > Editor dialog box appears with the file open at the corresponding location.

**Call Stack** pane

Use this pane to explore your source code. View the call stack for the selected site or task annotation's code region. Click to display related code regions in the **File: filename** pane, or right click to display the context menu.

---

**Call Stack Pane (Suitability Source Window)**

Use this pane in the **Suitability Source** window to explore your source code. The current position on the **Call Stack** pane is the top-most entry.

---

Use This | To Do This
--- | ---
• The row displayed is for a function  .
• Source code is available for viewing and editing. An icon indicates whether source code is available  or not available  .

Click a row in the **Call Stack** pane

Displays source code for the specified location in the call stack tree.

Pane border (drag)

Resize the pane.

Right click a row in the **Call Stack** pane

Customize call stack presentation by using the **Call Stack** context menu.

---

**File: filename Pane (Suitability Source Window)**

Use this pane to view the source code for source lines.
<table>
<thead>
<tr>
<th>Use This</th>
<th>To Do This</th>
</tr>
</thead>
<tbody>
<tr>
<td>Source code lines</td>
<td>You can view and navigate to related source code by using the Call Stack pane.</td>
</tr>
<tr>
<td>Double-click a source</td>
<td>To open the code editor to the corresponding source line.</td>
</tr>
<tr>
<td>line</td>
<td>• On Windows* OS:</td>
</tr>
<tr>
<td></td>
<td>• When using Visual Studio, the Visual Studio code editor appears with the file open at the corresponding location.</td>
</tr>
<tr>
<td></td>
<td>• When using the Intel Advisor GUI, the file type association (or Open With dialog box) determines the editor used.</td>
</tr>
<tr>
<td></td>
<td>• On Linux* OS: When using the Intel Advisor GUI, the editor defined by the Options &gt; Editor dialog box appears with the file open at the corresponding location.</td>
</tr>
<tr>
<td></td>
<td>To return to the Suitability Source or Suitability Report window, click the Result tab.</td>
</tr>
<tr>
<td>Select multiple source</td>
<td>To copy source lines using the context menu.</td>
</tr>
<tr>
<td>lines</td>
<td></td>
</tr>
<tr>
<td>Right click a source</td>
<td>Display a context menu to: open the code editor to the corresponding source file, copy the selected source line(s) to the clipboard, or display context-sensitive help relevant to the selected loop or function.</td>
</tr>
<tr>
<td>line or multiple source</td>
<td></td>
</tr>
<tr>
<td>lines</td>
<td></td>
</tr>
</tbody>
</table>

**Call Stack Pane, Function Node**

The Suitability Source window shows you the code where annotations are found. Use the Suitability Source window to view the selected source code to understand its execution paths and data use.

The Call Stack pane shows you the program context in which the annotation was executed, including call sites and loops. The current position on this Call Stack pane is the top-most entry.

Use the Call Stack pane to understand the context in which annotations are executed. This can help you to understand how often the annotations are executed, which can help to understand the overhead values shown in the Suitability Report window.

To return to the Suitability Report window, click the Suitability Report button.

**See Also**

- Suitability Source Window
- Call Stack Pane (Suitability Source Window)
- Suitability Tool Overview

**Call Stack Pane, Loop Node**

The Suitability Source window shows you the code where annotations are found. Use the Suitability Source window to view the selected source code to understand its execution paths and data use.

The Call Stack pane shows you the program context in which the annotation was executed, including call sites and loops. The current position on this Call Stack pane is the top-most entry.

Use the Call Stack pane to understand the context in which annotations are executed, including executions paths and data use. This can help you to understand how often the annotations are executed, which can help to understand the overhead values shown in the Suitability Report window.

To return to the Suitability Report window, click the Suitability Report button.

**See Also**

- Suitability Source Window
- Call Stack Pane (Suitability Source Window)
Suitability Tool Overview

Source Code Pane File: filename (Suitability Source Window)
The Suitability Source window shows you the code where annotations are executed. Use the Suitability Source window to view the selected source code to understand its execution paths and data use.
The Call Stack pane shows you the program context in which the annotation was executed, including call sites and loops. The current position on this Call Stack pane is the top-most entry.
Double-click a line in this view to open the code editor to that file.

On Windows* OS:
• When using Visual Studio, the Visual Studio code editor appears with the file open at the corresponding location.
• When using the Intel Advisor GUI, the file type association (or Open With dialog box) determines the editor used.

On Linux* OS: When using the Intel Advisor GUI, the editor defined by the Options > Editor dialog box appears with the file open at the corresponding location.

To return to the Suitability Report window, click the Suitability Report button.

For Each Site, Decide Whether to Modify or Keep Annotations
Use the Suitability Report window to view the predicted parallel performance of each parallel site and its impact on the Maximum Program Gain for All Sites. For example, if a site either has a Site Gain of less than 1.0 or does not contribute to Maximum Program Gain for All Sites, modify or remove its annotations. In contrast, any site that contributes to Maximum Program Gain for All Sites should be kept. For most sites, carefully examine the annotations, overhead assumptions, and related code.

See Also
• Copying Annotations and Build Settings Using the Annotation Assistant Pane
• Dependencies Analysis

Intel® Advisor Beta: GPU Roofline

Purpose and Usage | Run | Chart Controls | Chart Data | Limitations

GPU Roofline Analysis Purpose and Usage
To view a Roofline chart of the GPU kernels in your application, run a GPU Roofline analysis.
The GPU Roofline analysis helps estimate and visualize the actual performance of GPU kernels using benchmarks and hardware metric profiling against hardware-imposed performance ceilings, as well as determine the main limiting factor. When you run a GPU Roofline analysis, the Intel® Advisor Beta:

• Collects OpenCL™ kernels timings and memory data using the Survey analysis with GPU profiling.
• Measures the hardware limitations and collects floating-point and integer operations data using the Trip Counts and FLOP analysis with GPU profiling.

NOTE
To run the GPU Roofline analysis from the Intel® Advisor Beta GUI (technical preview), set the ADVIXE_EXPERIMENTAL-beta_gui variable. For more information, see Intel® Advisor User Guide (PDF).

Use the Roofline chart to answer the following questions:
• What is the maximum achievable performance with your current hardware resources?
• Does your application work optimally on current hardware resources?
• If not, what are the best candidates for optimization?
• Is memory bandwidth or compute capacity limiting performance for each optimization candidate?

**Memory Levels**

Dots on the Roofline chart correspond to OpenCL kernels, as described further in the GPU Roofline Analysis Chart Data section. You can choose which memory levels (CARM, L3, SLM, GTI, L3 + SLM) to plot dots for on the Roofline chart:

- **CARM**: Memory traffic generated by all execution units (EUs). Includes traffic between EUs and corresponding GPU cache or direct traffic to main memory. For each retired instruction with memory arguments, the size of each memory operand in bytes is added to this metric.
- **L3**: Data transferred directly between execution units and L3 cache.
- **SLM**: Memory access to/from Shared Local Memory (SLM), a dedicated structure within the L3 cache.
- **GTI**: Represents GTI traffic/GPU memory read bandwidth, the accesses between the GPU, chip uncore (LLC), and main memory. Use this to get a sense of external memory traffic.
- **L3 + SLM**: Summary traffic to/from L3 and Shared Local Memory.

Arithmetic intensity determines the order in which dots are plotted, which can provide some insight into your code’s performance. For example, the CARM dot is typically far to the right of the L3 dot, as read/write access is by cache lines and CARM traffic is the sum of actual bytes used in operations.

**Run a GPU Roofline Analysis**

**Set Up Environment**

<table>
<thead>
<tr>
<th>Environment</th>
<th>Set-Up Tasks</th>
</tr>
</thead>
<tbody>
<tr>
<td>Intel® Advisor Beta/Linux* OS</td>
<td>Important&lt;br&gt;The GPU Roofline analysis is available only on Linux* OS with a kernel version 4.14 or higher.</td>
</tr>
</tbody>
</table>

1. Switch to a root mode to run the GPU Roofline analysis with root privileges.
   Alternatively, for example, if you want to run the analysis as a user without root privileges, add your username to the video group. To check if you are already in the video group, run:

   ```bash
   groups | grep video
   ```
   If you are not part of the video group, add your username to it:

   ```bash
   sudo usermod -a -G video <username>
   ```
   Type `groups` to verify that you successfully added your username to the video group. If video is not listed, log out and log back in.

2. Set the value of the `dev.i915.perf_stream_paranoid` sysctl option to 0:

   ```bash
   sysctl -w dev.i915.perf_stream_paranoid=0
   ```
Run GPU Roofline Analysis

Important

- GPU profiling is applicable only to Intel® Processor Graphics.
- This page explains how to run the GPU Roofline analysis from the Intel® Advisor Beta command-line interface. To run the GPU Roofline analysis from the Intel® Advisor Beta GUI (technical preview), set the ADVIXE_EXPERIMENTAL=beta_gui variable. For more information, see Intel® Advisor User Guide (PDF).

Your target application must use OpenCL™ either directly or indirectly - for example, using the current implementation of Data Parallel C++ (DPC++), SYCL*, or OpenMP* for GPU.

To generate a Roofline visual model for GPU kernel performance estimation:

1. Collect the GPU Roofline data with the --profile-gpu option. The --profile-gpu option enables the analysis of OpenCL and Intel® Media SDK programs running on Intel® Processor Graphics. Do one of the following:
   - Run a shortcut --collect=roofline command, which runs the Survey and Trip Count with FLOP analyses one by one:

     ```bash
     advixe-cl --collect=roofline --profile-gpu --project-dir=<project-dir> --search-dir src:r=<source-dir> -- <target> [target-options]
     ```
   - Run the Survey and Trip Counts with FLOP analyses separately:

     ```bash
     advixe-cl --collect=survey --profile-gpu --project-dir=<project-dir> --search-dir src:r=<source-dir> -- <target> [target-options]
     advixe-cl --collect=tripcounts --flop --profile-gpu --project-dir=<project-dir> --search-dir src:r=<source-dir> -- <target> [target-options]
     ```

2. Generate a GPU Roofline report with the --gpu option:

   ```bash
   advixe-cl --report=roofline --gpu --project-dir=<project-dir> --report-output=<project-dir>/roofline.html
   ```

   **NOTE** To generate a GPU Roofline report for integer data, specify the --data-type=int option.

3. Review the HTML report with the GPU Roofline model for your application.
NOTE
You can use the Intel Advisor Python* API to collect GPU metrics for your project and print them in the terminal:

```
advixe-python <install-dir>/pythonapi/examples/survey_gpu.py <project-dir>
```

GPU Roofline Chart Controls
There are several controls to help you focus on the GPU Roofline chart data most important to you, including the following.

1. **Select by Mouse Rect**: Select one or more kernels by tracing a rectangle with your mouse.
2. **Zoom by Mouse Rect**: Zoom in and out by tracing a rectangle with your mouse. You can also zoom in and out using your mouse wheel.
3. **Move View by Mouse**: Move the chart left, right, up, and down.
4. **Undo** or **Redo**: Undo or redo the previous zoom action.

Select the **Memory Level(s)** to show for each kernel in the chart: CARM, L3, SLM, GTI, L3 + SLM.

Add visual **Guidance** to the GPU Roofline chart to make the interpretation of data easier, including performance limits and whether kernels are memory bound, compute bound, or both.

Use the following checkboxes in the **Guidance** drop-down toolbar:

- **Display roof rulers**: Enable showing a vertical line from a kernel to the nearest and topmost performance ceilings. To view the ruler, hover the cursor over a kernel dot. Where the line intersects with each roof, labels display hardware performance limits for the kernel.
- **Show memory level relationships**: Visually emphasize the relationships among displayed memory levels and roofs for a selected dot by enabling.

- **Show Roofline boundaries**: Color the GPU Roofline zones to make it easier to see if enclosed kernels are fundamentally memory bound, compute bound, or bound by compute and memory roofs.

The preview picture is updated as you select guidance options, allowing you to see how changes will affect the GPU Roofline chart's appearance. Click **Apply** to apply your changes or **Default** to return the GPU Roofline chart to its original appearance.

- **Roofline View Settings**: Change the default scale setting to show:
  - The **optimal** scale (default), which adjusts to a chosen GPU Roofline chart view.
  - A **constant** scale, which adjusts to the tallest or widest view and does not change when a different GPU Roofline chart view is chosen.

- **Roof Settings**: Change the visibility and appearance of roofline representations (lines):
  - Click a **Visible** checkbox to show/hide a roof line.
  - Click a **Selected** checkbox to change a roof line appearance: display the roof line as a solid or a dashed line.
  - Manually fine-tune roof values in the **Value** column to set hardware limits specific to your code.

- **Loop Weight Representation**: Change the appearance of dots:
  - **Point Weight Calculation**: Change the **Base Value** for a point weight calculation.

- **Point Weight Ranges**: Change the **Size**, **Color**, and weight **Range (R)** of a dot. Click the + button to split a point weight range in two. Click the - button to merge a point weight range with the range below.

- **Point Colorization**: Color dots by **weight ranges** or by **type** (vectorized or scalar). You can also change the color of loop with no self time.

---

**NOTE** For a GPU Roofline chart, only **Self Elapsed Time** is available as a base value.
Hover your mouse over a dot to display metrics and, if enabled, a roof ruler for it.

By default, Intel Advisor Beta generates a **GPU Cache-Aware Roofline Model (CARM)**, which reports memory traffic, in bytes, generated by all execution units.

If **Show memory level relationships** is enabled: Double-click a dot or select a dot and press **SPACE** or **ENTER** to display labeled dots representing memory levels for the selected kernel. Lines connect the dots to indicate that they correspond to the selected kernel.

**NOTE** If you have chosen to display only some memory levels in the chart using the **Memory Level** toolbar, unselected memory levels are displayed with **X** marks.

To hide the labeled dots, do one of the following:

- Select another kernel.
- Double-click an empty space in the GPU Roofline chart.
- Press **SPACE** or **ENTER**.
- Right-click a kernel dot or a blank area in the Roofline chart to perform more functions, such as:
  - Further simplify the GPU Roofline chart by filtering out (temporarily hiding a dot), filtering in (temporarily hiding all other dots), and clearing filters (showing all originally displayed dots).
  - Add visual guidance to the GPU Roofline chart to make the interpretation of data easier. These options are the same as in the **Guidance** toolbar.

---

**GPU Roofline Chart Data**

**Learn More About Roofline Charts and Investigate Kernels**
A **Roofline** chart plots an application’s *achieved performance* and *arithmetic intensity* against the machine’s *maximum achievable performance*:

- **Arithmetic intensity** (x axis) - measured in number of floating-point operations (FLOPs) per byte, based on the kernel algorithm, transferred between GPU and memory.
- **Performance** (y axis) - measured in billions of floating-point operations per second (GFLOPS).

In general:

- The size and color of each dot represent relative execution time for each kernel. Large red dots take the most time, so are the best candidates for optimization. Small green dots take less time, so may not be worth optimizing.
- Diagonal lines indicate *memory bandwidth limitations* preventing kernels from achieving better performance without some form of optimization.

- **L3 cache roof**: Represents the maximal bandwidth of the L3 cache for your current graphics hardware. Measured using an optimized sequence of load operations, iterating over an array that fits entirely into L3 cache.
- **SLM cache roof**: Represents the maximal bandwidth of the Shared Local Memory for your current graphics hardware. Measured using an optimized sequence of load and store operations that work only with SLM.
- **GTI roof**: Represents the maximum bandwidth between the GPU and the rest of the SoC. This estimate is calculated via analytical formula based on the maximum frequency of your current graphics hardware.
- **DRAM roof**: Represents the maximal bandwidth of the DRAM memory available to your current graphics hardware. Measured using an optimized sequence of load operations, iterating over an array that does not fit in GPU caches.
- Horizontal lines indicate *compute capacity limitations* preventing kernels from achieving better performance without some form of optimization.
• A dot cannot exceed the topmost rooflines, as these represent the maximum capabilities of the machine. However, not all kernels can utilize maximum machine capabilities.
• The greater the distance between a dot and the highest achievable roofline, the more opportunity exists for performance improvement.

The GPU Roofline chart is based on a CPU Roofline chart, but there are some differences:
• The dots on the chart correspond to OpenCL kernels, while in the CPU version, they correspond to individual loops.
• Some displayed information and controls (for example, thread/core count) are not relevant to GPU Roofline. For more information, see the table below.
• Integrated chart shows multiple dots for a single kernel. These dots correspond to different memory levels used to calculate arithmetic intensity. Hover over a dot to identify its arithmetic intensity. To show or hide certain dots from a chart, use the Memory Level drop-down filter.

**Known Issues and Limitations**

On Windows OS:
• GPU Roofline works only with 64-bit application.
• GPU Roofline does not support Intel® Math Kernel Library (Intel® MKL) applications.

**Minimizing Analysis Overhead**

Running your target application with the Intel Advisor can take substantially longer than running your target application without the Intel Advisor. For example:

<table>
<thead>
<tr>
<th>Runetime Overhead / Analysis</th>
<th>Survey</th>
<th>Trip Counts &amp; FLOP</th>
<th>Roofline</th>
<th>Dependencies</th>
<th>MAP</th>
</tr>
</thead>
<tbody>
<tr>
<td>Target application runtime with Intel Advisor compared to runtime without Intel Advisor</td>
<td>1.1x longer</td>
<td>3 - 8x longer</td>
<td>3.1 - 8.1x longer</td>
<td>5 - 100x longer</td>
<td>5 - 20x longer</td>
</tr>
</tbody>
</table>

The following tables summarize how to minimize overhead while collecting and finalizing Intel Advisor analysis data. The techniques are grouped by:
• Collection Controls
• Loop Markup
• Filtering
• Execution Speed/Duration/Scope Properties
• Miscellaneous Techniques

### Collection Controls

The following table is a summary. For more information, see Collection Controls to Minimize Analysis Overhead.

<table>
<thead>
<tr>
<th>Minimization Technique</th>
<th>Impacted Analyses</th>
<th>Summary</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pause collection/ Resume collection using API methods</td>
<td>• Roofline</td>
<td>Pause collection:</td>
</tr>
<tr>
<td></td>
<td>• Survey</td>
<td>• C++: __itt_pause</td>
</tr>
<tr>
<td></td>
<td>• Trip Counts &amp; FLOP</td>
<td>• Fortran: Use ITTNOTIFY statement to call ITT_PAUSE() subroutine</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Resume collection:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• C++: __itt_resume</td>
</tr>
<tr>
<td>Minimization Technique</td>
<td>Impacted Analyses</td>
<td>Summary</td>
</tr>
<tr>
<td>------------------------</td>
<td>-------------------</td>
<td>---------</td>
</tr>
</tbody>
</table>
| Pause collection/Resume collection using annotations | • Survey  
• Dependencies  
Some analysis types recognize the structural annotations typically used in the Threading Advisor workflow. | • Fortran: Use ITTNOTIFY statement to call ITT_RESUME() subroutine |
| Start target application with collection paused | • Survey  
• Trip Counts & FLOP | Pause collection:  
• C++: ANNOTATE_DISABLE_COLLECTION_PUSH  
• Fortran: call annotate_disable_collection_push()  
• C#: Annotate.DisableCollectionPush();  
Resume collection:  
• C++: ANNOTATE_DISABLE_COLLECTION_POP  
• Fortran: call annotate_disable_collection_pop()  
• C#: Annotate.DisableCollectionPop(); |
| Start target application with collection paused/ Resume collection after N seconds | • Survey  
• Trip Counts & FLOP | Start target application with collection paused:  
• GUI control: Workflow pane > Start paused control  
• advixe-cl CLI action option: -start-paused  

**NOTE**  
You can use different techniques to resume collection. The most common is __itt_resume__ |
| Stop collection after N seconds | All | GUI control: Project Properties > Analysis Target > [Name] Analysis > Advanced > Automatically resume collection after (sec) checkbox  
advixe-cl CLI action option: -resume-after=<integer> |
| Stop collection | All | GUI control: Workflow pane > Stop current analysis control and Site Coverage widget  
advixe-cl CLI action option: -command=stop |
| Manually pause collection/Manually resume collection | • Survey  
• Trip Counts & FLOP | Pause collection:  
• GUI control: Workflow pane > Pause control  
• advixe-cl CLI action: -command=pause  
Resume collection:  
• GUI control: Workflow pane > Resume control  
• advixe-cl CLI action: -command=resume |
**Minimization Technique** | **Impacted Analyses** | **Summary**
--- | --- | ---
Attach to process/ Detach from process | • Survey  
• Trip Counts & FLOP | Attach to process:  
• GUI control: **Project Properties > Analysis Target > [Name]**  
**Analysis > Launch Application** drop-down list > **Attach to Process**  
• `advixe-cl` CLI action options: `-target-pid=<unsigned integer>` and `-target-process=<string>`  
Detach from process:  
• GUI control: **Workflow pane > Stop current analysis** control  
• `advixe-cl` CLI action: `-command=detach`

**Loop Markup**
The following table is a summary. For more information, see [Loop Markup to Minimize Analysis Overhead](#).

<table>
<thead>
<tr>
<th>Minimization Technique</th>
<th>Impacted Intel Advisor Analyses</th>
<th>Summary</th>
</tr>
</thead>
</table>
| Select loops by ID | • Trip Counts & FLOP  
• Dependencies  
• Memory Access Patterns | GUI control: **Survey Report** checkbox(es)  
`advixe-cl` CLI action option: `-mark-up-list=<string>` |
| Select loops by source file/line | • Trip Counts & FLOP  
• Dependencies  
• Memory Access Patterns | GUI control: **Survey Report** checkbox(es)  
`advixe-cl` CLI action: `-mark-up-loops with action option -select=<string>` |
| Select loops by criteria | • Dependencies  
• Memory Access Patterns | GUI control: **Workflow pane > Batch mode** settings  
`advixe-cl` CLI: action `-mark-up-loops or -collect with action option -loops=<string>` |

**Filtering**
The following table is a summary. For more information, see [Filtering to Minimize Analysis Overhead](#).

<table>
<thead>
<tr>
<th>Minimization Technique</th>
<th>Impacted Intel Advisor Analyses</th>
<th>Summary</th>
</tr>
</thead>
</table>
| Filter modules | • Survey  
• Trip Counts & FLOP | GUI control: **Project Properties > Analysis Target > [Name]**  
**Analysis > Modules** options and field  
`advixe-cl` CLI: action option: `-module-filter-mode=include | exclude and -module-filter=<string>` |

**Execution Speed/Duration/Scope Properties**
The following table is a summary. For more information, see [Execution Speed/Duration/Scope Properties to Minimize Analysis Overhead](#).
<table>
<thead>
<tr>
<th>Minimization Technique</th>
<th>Impacted Intel Advisor Analyses</th>
<th>Summary</th>
</tr>
</thead>
<tbody>
<tr>
<td>Change stackwalk mode from offline (after collection) to online (during collection)</td>
<td>Survey</td>
<td>GUI control: Project Properties &gt; Analysis Target &gt; Survey Hotspots Analysis &gt; Advanced &gt; Stack unwinding mode &gt; During collection&lt;br&gt;advixe-cl CLI action option: -stackwalk-mode=online</td>
</tr>
<tr>
<td>Disable stacks collection</td>
<td>• Roofline&lt;br&gt;• Trip Counts &amp; FLOP</td>
<td>GUI controls:&lt;br&gt;• Vectorization Workflow pane &gt; Enable Roofline with Callstacks checkbox&lt;br&gt;• Project Properties &gt; Analysis Target &gt; Trip Counts and FLOP Analysis &gt; Advanced &gt; Collect stacks checkbox&lt;br&gt;advixe-cl CLI action option: -no-stacks (or just ensure the CLI action option -stacks is omitted from the advixe-cl command line)</td>
</tr>
<tr>
<td>Disable stitch stacks</td>
<td>Survey</td>
<td>GUI control: Project Properties &gt; Analysis Target &gt; Survey Hotspots Analysis &gt; Advanced &gt; Stitch stacks checkbox&lt;br&gt;advixe-cl CLI action option: -no-stack-stitching</td>
</tr>
<tr>
<td>Increase sampling interval</td>
<td>Survey</td>
<td>GUI control: Project Properties &gt; Analysis Target &gt; Survey Hotspots Analysis &gt; Advanced &gt; Sampling interval field&lt;br&gt;advixe-cl CLI action option: interval=&lt;integer&gt;</td>
</tr>
<tr>
<td>Limit collected analysis data</td>
<td>Survey</td>
<td>GUI control: Project Properties &gt; Analysis Target &gt; Survey Hotspots Analysis &gt; Advanced &gt; Collection data limit, MB field&lt;br&gt;advixe-cl CLI action option: -data-limit=&lt;integer&gt;</td>
</tr>
<tr>
<td>Limit loop call count</td>
<td>• Dependencies&lt;br&gt;• Memory Access Patterns</td>
<td>GUI control: Project Properties &gt; Analysis Target &gt; [Name] Analysis &gt; Advanced &gt; Loop Call Count Limit field&lt;br&gt;advixe-cl CLI action option: -loop-call-count-limit=&lt;integer&gt;</td>
</tr>
<tr>
<td>Disable additional analysis</td>
<td>Survey</td>
<td>GUI controls: Project Properties &gt; Analysis Target &gt; Survey Hotspots Analysis &gt; Advanced&lt;br&gt;• Analyze MKL loops and functions checkbox&lt;br&gt;• Analyze Python loops and functions checkbox&lt;br&gt;• Analyze loops that reside in non-executed code paths checkbox&lt;br&gt;• Enable register spill/fill analysis checkbox&lt;br&gt;• Enable static instruction mix analysis checkbox&lt;br&gt;advixe-cl CLI action options:&lt;br&gt;• -no-mkl-user-mode&lt;br&gt;• -no-profile-python</td>
</tr>
</tbody>
</table>
### Minimization Technique

#### Impacted Intel Advisor Analyses

**Summary**

- -no-support-multi-isa-binaries
- -no-spill-analysis
- -no-static-instruction-mix

### Miscellaneous Techniques

The following table is a summary. For more information, see Miscellaneous Techniques to Minimize Analysis Overhead.

<table>
<thead>
<tr>
<th>Minimization Technique</th>
<th>Impacted Intel Advisor Analyses</th>
<th>Summary</th>
</tr>
</thead>
<tbody>
<tr>
<td>Disable cache simulation</td>
<td>• Trip Counts &amp; FLOP&lt;br&gt;• Memory Access Patterns</td>
<td>GUI controls:&lt;br&gt;• Project Properties &gt; Analysis Target &gt; Memory Access Patterns Analysis &gt; Advanced &gt; Enable cache simulation checkbox&lt;br&gt;• Project Properties &gt; Analysis Target &gt; Trip Counts and FLOP Analysis &gt; Advanced &gt; Enable cache simulation checkbox&lt;br&gt;advixe-cl CLI action option: -no-enable-cache-simulation</td>
</tr>
<tr>
<td>Limit reported data</td>
<td>Memory Access Patterns</td>
<td>GUI controls:&lt;br&gt;• Project Properties &gt; Analysis Target &gt; Memory Access Patterns Analysis &gt; Advanced &gt; Report stack variables checkbox&lt;br&gt;• Project Properties &gt; Analysis Target &gt; Memory Access Patterns Analysis &gt; Advanced &gt; Report heap allocated variables checkbox&lt;br&gt;advixe-cl CLI action options:&lt;br&gt;• -no-record-stack-frame&lt;br&gt;• -no-record-mem-allocations</td>
</tr>
<tr>
<td>Minimize data set</td>
<td>All, but especially:&lt;br&gt;• Dependencies&lt;br&gt;• Memory Access Patterns</td>
<td>Minimize number of instructions executed within a loop while thoroughly exercising target application control flow paths</td>
</tr>
<tr>
<td>Temporarily disable finalization until opening result in GUI</td>
<td>• Roofline&lt;br&gt;• Survey&lt;br&gt;• Trip Counts &amp; FLOP</td>
<td>GUI control: <strong>Workflow pane &gt; Cancel current analysis</strong> control during finalization&lt;br&gt;advixe-cl CLI action option: -no-auto-finalize</td>
</tr>
</tbody>
</table>
Collection Controls to Minimize Analysis Overhead

Issue
Running your target application with the Intel Advisor can take substantially longer than running your target application without the Intel Advisor. For example:

<table>
<thead>
<tr>
<th>Run-time Overhead / Analysis</th>
<th>Survey</th>
<th>Trip Counts &amp; FLOP</th>
<th>Roofline</th>
<th>Dependences</th>
<th>MAP</th>
</tr>
</thead>
<tbody>
<tr>
<td>Target application runtime with Intel Advisor compared to runtime without Intel Advisor</td>
<td>1.1x longer</td>
<td>3 - 8x longer</td>
<td>3.1 - 8.1x longer</td>
<td>5 - 100x longer</td>
<td>5 - 20x longer</td>
</tr>
</tbody>
</table>

Solutions
Use the following techniques to skip uninteresting parts of your target application, such as the initialization phase, and analyze only interesting parts.

This section contains details on using the Intel® Instrumentation and Tracing Technology (ITT) API to minimize analysis overhead. You can find the ITT API documentation at https://github.com/intel/ittapi.

Pause Collection/Resume Collection Using Annotations
Minimize collection overhead.
Applicable analyses: Survey, Dependencies.
Some analysis types recognize the structural annotations typically used in the Threading Advisor workflow.

Use when:
• Modifying/recompiling your target application is not an issue.
• You do not want to analyze one or more uninteresting parts of your target application.
• The interesting parts of your target application involve large workloads (because Pause/Resume API call frequency is about 1 Hz, and the operations pause and resume data collection in all processes in the analysis run, with the corresponding collection state notification to the GUI).

To pause collection, add the following annotation to your code:
• C++: ANNOTATE_DISABLE_COLLECTION_PUSH
• Fortran: call annotate_disable_collection_push()
• C#: Annotate.DisableCollectionPush();

To resume collection, add the following annotation to your code:
• C++: ANNOTATE_DISABLE_COLLECTION_POP
• Fortran: call annotate_disable_collection_pop()
• C#: Annotate.DisableCollectionPop();

See Pause Collection and Resume Collection Annotations for more information.

Pause Collection/Resume Collection Using API Methods
Minimize collection overhead.
Applicable analyses: Roofline, Survey, Trip Counts and FLOP.
Use when:
• Modifying/recompiling your target application is not an issue.
• You do not want to analyze one or more uninteresting parts of your target application.
• The interesting parts of your target application involve large workloads (because Pause/Resume API call frequency is about 1 Hz, and the operations pause and resume data collection in all processes in the analysis run, with the corresponding collection state notification to the GUI).

Prerequisites:
• Add the following statements to every source file you want to instrument:
  • C/C++: ittnotify.h
  • Fortran: USE ITTNOTIFY

**NOTE**
• The ittnotify header file contains definitions of ITT API routines and important macros that provide the correct logic of API invocation from an application.
• The ITT API is designed to incur almost zero overhead when tracing is disabled. If you need completely zero overhead, you can compile out all ITT API calls from an application by defining the INTEL_NO_ITTNOTIFY_API macro in your project at compile time, either on the compiler command line or in your source file prior to including the ittnotify header file.

• Configure your build system to reach ITT API header file and libraries, where <install-dir> is the Intel Advisor installation directory.
  • Add the appropriate entry to your INCLUDE path:
    • C++: <install-dir>/sdk/include
    • Fortran: <install-dir>/sdk/include/lib32 or <install-dir>/sdk/include/lib64
    • Microsoft Visual Studio* IDE: Project Properties > C/C++ | Fortran > General > Additional Include Directories.
  • Add <install-dir>/sdk/lib32 or <install-dir>/sdk/lib64 to your LIBRARIES path.

**NOTE** The ITT API headers, static libraries, and Fortran modules previously located at <install-dir>/include and <install-dir>/[lib32 | lib64] folders were moved to the <install-dir>/sdk/include and <install-dir>/sdk/[lib32 | lib64] folder starting with the Intel® Advisor Beta 2021.1-beta08 release. Copies of these files are retained at their old locations for backward compatibility and these copies should not be used for new projects.

• Link your target application to the static library libittnotify.a (Linux* OS) or libittnotify.lib (Windows* OS) by passing -littnotify to your compiler. If tracing is enabled, this static library loads the ITT API implementation and forwards ITT API instrument data to the Intel Advisor. If tracing is disabled, the static library ignores ITT API calls, providing nearly zero instrumentation overhead.

**NOTE** The ITT API headers, static libraries, and Fortran modules previously located at <install-dir>/include and <install-dir>/[lib32 | lib64] folders were moved to the <install-dir>/sdk/include and <install-dir>/sdk/[lib32 | lib64] folder starting with the Intel® Advisor Beta 2021.1-beta08 release. Copies of these files are retained at their old locations for backward compatibility and these copies should not be used for new projects.

• Insert _itt_pause (C/C++) or CALL ITT_PAUSE (Fortran) before uninteresting parts of your target application and the _itt_resume (C/C++) or CALL ITT_RESUME (Fortran) before interesting parts of your target application.
Example 1: The following snippet plus the standard run control collects analysis data twice - at the beginning and the middle of the snippet:

```c
#include <ittnotify.h>
int main(int argc, char* argv[]) {
    // Do work here
    __itt_pause();
    // Do uninteresting work here
    __itt_resume();
    // Do work here
    __itt_pause();
    // Do uninteresting work here
    return 0;
}
```

Example 2: The following snippet plus the standard run control collects analysis data only once - in the middle of the snippet:

```c
#include <ittnotify.h>
int main(int argc, char* argv[]) {
    __itt_pause();
    // Do uninteresting work here
    __itt_resume();
    // Do work here
    __itt_pause();
    // Do uninteresting work here
    return 0;
}
```

Example 3: The following snippet plus the standard run control collects analysis data only once - at the end of the snippet:

```c
#include <ittnotify.h>
int main(int argc, char* argv[]) {
    __itt_pause();
    // Do uninteresting work here
    __itt_resume();
    // Do work here
    return 0;
}
```

Example 4: The following snippet plus the Start Paused control collects analysis data only once - at the end of the snippet:

```c
#include <ittnotify.h>
int main(int argc, char* argv[]) {
    // Do uninteresting work here
    __itt_resume();
    // Do work here
    return 0;
}
```

After performing the prerequisites and recompiling, do one of the following:

- Click the standard run control or the Start Paused control on the Workflow pane to run the desired analysis.
• Use the advixe-cl CLI action --collect with or without the CLI action option --start-paused to run the desired analysis. For example:
  
  advixe-cl --collect=survey --project-dir=./myAdvisorProj -- ./bin/myTargetApplication

To attach ITT APIs to a launched application, that is, to collect API data on an application that is already launched, point the target application to the ittnotify_collector library using an environment variable:

• Windows* OS:
  
  set INTEL_LIBITTNOTIFY32=<install_dir>\bin32\runtime\ittnotify_collector.dll
  set INTEL_LIBITTNOTIFY64=<install_dir>\bin64\runtime\ittnotify_collector.dll

• Linux* OS:
  
  export INTEL_LIBITTNOTIFY32=<install_dir>/lib32/runtime/libittnotify_collector.so
  export INTEL_LIBITTNOTIFY64=<install_dir>/lib64/runtime/libittnotify_collector.so

**NOTE**
Use the full path to the library without quotations marks.

After you complete configuration, start the instrumented application in the correct environment. Intel Advisor collects API data even if the application is launched before the Intel Advisor is launched.

**Start Target Application With Collection Paused**

Minimize collection overhead.

Applicable analyses: Survey, Trip Counts and FLOP.

Use when you do not want to analyze the early phase(s) of your target application, such as the initialization phase, but you want analysis in *ready mode*.

To implement, do one of the following:

• Click the associated control on the Workflow pane to run the desired analysis.

• Use the advixe-cl CLI action option --start-paused when you run the desired analysis. For example:
  
  advixe-cl --collect=survey --project-dir=./myAdvisorProj --start-paused -- ./bin/myTargetApplication

**NOTE**
You can use different techniques to resume collection. The most common is __itt_resume.__

**Start Target Application With Collection Paused/Resume Collection After N Seconds**

Minimize collection overhead.

Applicable analyses: Survey, Trip Counts and FLOP.

Use when...

• You do not want to modify/recompile your target application.
• You do not want to analyze the initialization phase of your target application.
• You have a good idea of the time interval of interest, but pinpointing the exact beginning of the interesting part of your target application is not important.
• The interesting part of your target application is more than a few loops.
To implement, do one of the following:

- Enable the **Project Properties > Analysis Target > [Name] Analysis > Advanced > Automatically resume collection after (sec)** checkbox and supply the desired value.

  Click the standard run control on the **Workflow** pane to run the desired analysis. (Collection automatically starts in the paused state.)

- Use the `advixe-cl` CLI action option `--resume-after=<integer>` when you run the desired analysis.

  For example:

  ```bash
  advixe-cl --collect=survey --project-dir=./myAdvisorProj --resume-after=30 -- ./bin/myTargetApplication
  ```

  **NOTE**
  Use a value representing seconds in the GUI field and milliseconds in the integer argument.

---

**Stop Collection After N Seconds**

Minimize collection overhead.

Applicable analyses: Roofline, Survey, Trip Counts and FLOP, Dependencies, Memory Access Patterns.

This is the flip side of the **Start target application with collection paused** technique. Use when...

- You do not want to modify/recompile your target application.
- You do not want to analyze the end of your target application.
- You have a good idea of the time interval of interest, but pinpointing the exact end of the interesting part of your target application is not important.
- The interesting part of your target application is more than a few loops.

To implement, do one of the following:

- Enable **Project Properties > Analysis Target > [Name] Analysis > Advanced > Automatically stop collection after (sec)** checkbox and supply the desired value.

  Click the standard run control on the **Workflow** pane to run the desired analysis.

- Use the `advixe-cl` CLI action option `--stop-after=<integer>` when you run the desired analysis. For example:

  ```bash
  advixe-cl --collect=survey --project-dir=./myAdvisorProj --stop-after=30 -- ./bin/myTargetApplication
  ```

  **NOTE**
  Use a value representing seconds in both the GUI field and integer argument.

---

**Stop Collection**

Minimize collection overhead.

Applicable analyses: Roofline, Survey, Trip Counts and FLOP, Dependencies, Memory Access Patterns.

Use when...

- You do not want to modify/recompile your target application.
- You do not want to analyze the end of your target application.
- You can detect the time interval of interest based on target application output.
- The interesting part of your target application is more than a few loops.

To implement, do one of the following:
Click the associated control on the Workflow pane when running the desired analysis.

**NOTE**
If running a Dependencies or Memory Access Patterns analysis, use the Site Coverage bar to determine when all marked loops are analyzed at least once:

- Use the advixe-cl CLI action --command=stop when you run the desired analysis. For example:
  
  ```bash
  advixe-cl --command=stop --result-dir=./myAdvisorResult
  ```

**Manually Pause Collection/Manually Resume Collection**

Minimize collection overhead.

Applicable analyses: Survey, Trip Counts and FLOP.

Use when...
- You can detect the time interval of interest based on target application output.
- Your need to pause or resume is unplanned and spontaneous.

To implement, do one of the following to pause analysis data collection (the target application continues running, but analysis data collection stops):

- Click the associated control on the Workflow pane when running the desired analysis.
- Use the advixe-cl CLI action --command=pause when you run the desired analysis. For example:
  
  ```bash
  advixe-cl --command=pause --result-dir=./myAdvisorResult
  ```

Do one of the following to resume analysis data collection:

- Click the associated control on the Workflow pane.
- Use the advixe-cl CLI action --command=resume. For example:
  
  ```bash
  advixe-cl --command=resume --result-dir=./myAdvisorResult
  ```

**Attach to Process/Detach from Process**

Minimize collection overhead.

Applicable analyses: Survey, Trip Counts and FLOP without call stacks.

This technique is similar to the Start target application with collection paused technique, except you can attach to an already running process. This is particularly beneficial if:
- The process is a service that runs forever.
- The launching infrastructure is relatively complicated, such as a sequence of scripts that must be modified to embed a launch collection command.

GUI:
1. Choose **Project Properties** > **Analysis Target** > [Name] Analysis > **Launch Application** dropdown list > **Attach to Process**.
2. Disable the **Inherit settings from Survey Hotspots Analysis Type** checkbox.
3. Choose the **Process name** or **PID** option and identify a process.
4. Supply other information as desired and close the **Project Properties** dialog box.
5. Click the standard run control on the **Workflow** pane to run the desired analysis.

CLI: Use the `advixe-cl` CLI action option `--target-pid=<unsigned integer>` or `--target-process=<string>` to attach to a process when running the desired analysis. For example:

```
advixe-cl --collect=survey --project-dir=./myAdvisorProj --result-dir=./myAdvisorResult --target-process=myProcess
```

Do one of the following to stop collecting analysis data on a process (the process continues running but analysis data collection stops):

- Click the associated control on the **Workflow** pane.
- Use the `advixe-cl` CLI action `--command=detach`. For example:

```
advixe-cl --command=detach --result-dir=./myAdvisorResult
```

**NOTE**

- Ensure call stacks are disabled (which is the default setting) if you run the Trip Counts and FLOP analysis:
  - Disable the **Project Properties** > **Analysis Target** > **Trip Counts and FLOP Analysis** > **Advanced** > **Collect stacks** checkbox.
  - Add the `advixe-cl` CLI action option `--no-stacks` to the `--collect` command, or simply omit a `--stacks` action option on the `--collect` command.
  - **Using the `advixe-cl` CLI action `--command=stop` kills the process (which also stops analysis data collection).**

**See Also**

Loop Markup to Minimize Analysis Overhead
Filtering to Minimize Analysis Overhead
Execution Speed/Duration/Scope Properties to Minimize Analysis Overhead
Miscellaneous Techniques to Minimize Analysis Overhead

**Loop Markup to Minimize Analysis Overhead**

**Issue**

Running your target application with the Intel Advisor can take substantially longer than running your target application without the Intel Advisor. For example:

<table>
<thead>
<tr>
<th>Runtime Overhead / Analysis</th>
<th>Survey</th>
<th>Trip Counts &amp; FLOP</th>
<th>Roofline</th>
<th>Dependencies</th>
<th>MAP</th>
</tr>
</thead>
<tbody>
<tr>
<td>Target application runtime with Intel Advisor compared to runtime without Intel Advisor</td>
<td>1.1x longer</td>
<td>3 - 8x longer</td>
<td>3.1 - 8.1x longer</td>
<td>5 - 100x longer</td>
<td>5 - 20x longer</td>
</tr>
</tbody>
</table>
**Solutions**

Use the following techniques to skip *uninteresting* loops and analyze only *interesting* loops.

**Select Loops by ID**

Minimize collection overhead.

Applicable analyses: Roofline, Trip Counts and FLOP, Dependencies, Memory Access Patterns.

Use when...

- You want to perform a deeper analysis on only a few loops.
- CLI environment: You cannot identify source file/line numbers, such as when you are analyzing a target application for which you do not have access to source code.

Prerequisites:

1. Run a Survey analysis.
2. advixe-cl CLI environment: Identify the loop IDs for the loops of interest.

```
advixe-cl -report survey -project-dir ./myAdvisorProj -- ./bin/myTargetApplication
```

**Tip**

Intel Advisor reports tend to be very wide. Do one of the following to generate readable reports:

- Set your console width appropriately to avoid line wrapping.
- Pipe your report using the appropriate truncation command if you care only about the first few report columns.

After performing the prerequisites, do one of the following:

- Mark the loop(s) of interest by enabling the associated checkbox on the Survey Report.
  
  Then run a Trip Counts and FLOP, Dependencies, or Memory Access Patterns analysis.
- Mark the loop(s) of interest using the CLI action option `--mark-up-list=<string>` when running a Trip Counts and FLOP, Dependencies, or Memory Access Patterns analysis. For example:

  ```
  advixe-cl --collect=tripcounts --flop --project-dir=./myAdvisorProj --mark-up-list=5,10,12 -- ./bin/myTargetApplication
  ```

**NOTE**

There is essentially no difference between selecting loops by ID and selecting loops by source file/line in the GUI environment. The difference is in the CLI environment:

- The `advixe-cl CLI action option--mark-up-list=<string>` merely simulates enabling a GUI checkbox; therefore it persists only for the duration of the `--collect` command.
- The `advixe-cl CLI action--mark-up-loops and action option --select=<string>` actually enables a GUI checkbox; therefore it persists beyond the duration of the `--mark-up-loops command and applies to downstream analyses, such as Roofline, Trip Counts and FLOP, Dependencies, Memory Access Patterns.

**Select Loops by Source File/Line Number**

Minimize collection overhead.

Applicable analyses: Roofline, Trip Counts and FLOP, Dependencies, Memory Access Patterns.
Use when...
- You want to perform a deeper analysis on only a few loops.
- CLI environment: You are analyzing a target application for which you have access to source code and can identify source file/line numbers.

Prerequisites:
1. Run a Survey analysis.
2. advixe-cl CLI environment: If necessary, identify the source file and line number for the loops of interest.

```bash
advixe-cl --report=survey --project-dir=./myAdvisorProj -- ./bin/myTargetApplication
```

After performing the prerequisites, do one of the following:
- Mark the loop(s) of interest by enabling the associated checkbox on the Survey Report.
- Mark the loop(s) of interest using the advixe-cl CLI action `--mark-up-loops` and action option `--select=<string>`. For example:

```bash
advixe-cl --mark-up-loops --select=foo.cpp:34,bar.cpp:192 --project-dir=./myAdvisorProj -- ./bin/myTargetApplication
```

Then run a Trip Counts and FLOP, Dependencies, or Memory Access Patterns analysis.

**NOTE**
- You can also append and remove source file/line numbers using the advixe-cl CLI action option `--append=<string>` and `--remove=<string>` respectively.
- There is essentially no difference between selecting loops by ID and selecting loops by source file/line in the GUI environment. The difference is in the advixe-cl CLI environment:
  - The advixe-cl CLI action option `--mark-up-list=<string>` merely simulates enabling a GUI checkbox; therefore it persists only for the duration of the `--collect` command.
  - The advixe-cl CLI action `--mark-up-loops` and action option `--select=<string>` actually enables a GUI checkbox; therefore it persists beyond the duration of the `--mark-up-loops` command and applies to downstream analyses, such as Roofline, Trip Counts and FLOP, Dependencies, and Memory Access Patterns.

**Select Loops by Criteria**

Minimize collection overhead.

Applicable analyses: Dependencies, Memory Access Patterns.

Use when you want to perform a deeper analysis on loops chosen by criteria instead of by human input, such as when you are running the Intel Advisor in batch mode or using automated scripts.

To implement in the GUI environment:
1. Toggle on the **Batch mode** control at the top of the **Workflow** pane.
2. Enable the **Dependencies** and/or **Memory Access Patterns** checkboxes.
3. Choose **Automatic Selection** and enable the appropriate criteria checkbox(es):
   - Dependencies analysis:
     - **Scalar serial loops only** (CLI corollary = scalar)
     - **Innermost loops only** (CLI corollary = loop-height=N; set to 0 for innermost loops only)
- **Above .1% of total CPU time only** (CLI corollary = `total-time>N`, but you can specify percentage)
- **With "Assumed Dependency Present" issue only** (CLI corollary = `has-issue`)
- **Exclude loops without source location** (CLI corollary = `has-source`)
- **Top 10 loops with the biggest Self Time**

**Memory Access Patterns analysis**

- **With "Possible Inefficient Memory Access Pattern" issue only** (CLI corollary = `has-issue`)
- **Above .1% of total CPU time only** (CLI corollary = `total-time>N`, but you can specify percentage)
- **Exclude loops without source location** (CLI corollary = `has-source`)
- **Loop height** (CLI corollary = `loop-height=N`; where innermost loops have `loop-height=0`)
- **Top 10 loops with the biggest Self Time**

4. Click the **Collect** control.

To implement in the advixe-cl CLI environment, create a script similar to the following examples, which produce the same outcome. Use the **--loops** option to select loops by criteria:

- **Example 1:**
  ```bash
  advixe-cl --collect=survey --project-dir=./myAdvisorProj -- ./bin/myTargetApplication
  advixe-cl --mark-up-loops --loops="scalar,has-issue" --project-dir=./myAdvisorProj -- ./bin/myTargetApplication
  advixe-cl --collect=dependencies --project-dir=./myAdvisorProj -- ./bin/myTargetApplication
  ```

- **Example 2:**
  ```bash
  advixe-cl --collect=survey --project-dir=./myAdvisorProj -- ./bin/myTargetApplication
  advixe-cl --collect=dependencies --loops="scalar,has-issue" --project-dir=./myAdvisorProj
  ```

**See Also**
- Collection Controls to Minimize Analysis Overhead
- Filtering to Minimize Analysis Overhead
- Execution Speed/Duration/Scope Properties to Minimize Analysis Overhead
- Miscellaneous Techniques to Minimize Analysis Overhead

## Filtering to Minimize Analysis Overhead

### Issue

Running your target application with the Intel Advisor can take substantially longer than running your target application without the Intel Advisor. For example:

<table>
<thead>
<tr>
<th>Runtime Overhead / Analysis</th>
<th>Survey</th>
<th>Trip Counts &amp; FLOP</th>
<th>Roofline</th>
<th>Dependenc</th>
<th>MAP</th>
</tr>
</thead>
<tbody>
<tr>
<td>Target application runtime with Intel Advisor compared to runtime without Intel Advisor</td>
<td>1.1x longer</td>
<td>3 - 8x longer</td>
<td>3.1 - 8.1x longer</td>
<td>5 - 100x longer</td>
<td>5 - 20x longer</td>
</tr>
</tbody>
</table>

### Solution

Use the following techniques to skip **uninteresting** modules and/or analyze only **interesting** modules.
Filter Modules
Minimize collection and finalization overhead.
Applicable analyses: Survey, Trip Counts and FLOP.
Use to...
• Exclude modules you cannot optimize, such as third-party code.
• Include a small number of modules of interest.
To implement, do one of the following before/while running the desired analysis:
• Set Project Properties > Analysis Target > [Name] Analysis > Modules > Exclude the following module(s) and identify the modules.
• Use the advixe-cl CLI action options --module-filter-mode=exclude and --module-filter=<string>. For example:
  advixe-cl --collect=survey --project-dir=./myAdvisorProj --module-filter-mode=exclude --module-filter=foo1.so,foo2.so -- ./bin/myTargetApplication
• Set Project Properties > Analysis Target > [Name] Analysis > Modules > Include only the following module(s) and identify the modules.
• Use the advixe-cl CLI action options --module-filter-mode=include and --module-filter=<string>. For example:
  advixe-cl --collect=survey --project-dir=./myAdvisorProj --module-filter-mode=include --module-filter=foo1.so,foo2.so -- ./bin/myTargetApplication

See Also
Collection Controls to Minimize Analysis Overhead
Loop Markup to Minimize Analysis Overhead
Execution Speed/Duration/Scope Properties to Minimize Analysis Overhead
Miscellaneous Techniques to Minimize Analysis Overhead

Execution Speed/Duration/Scope Properties to Minimize Analysis Overhead

Issue
Running your target application with the Intel Advisor can take substantially longer than running your target application without the Intel Advisor. For example:

<table>
<thead>
<tr>
<th>Runtime Overhead / Analysis</th>
<th>Survey</th>
<th>Trip Counts &amp; FLOP</th>
<th>Roofline</th>
<th>Dependencies</th>
<th>MAP</th>
</tr>
</thead>
<tbody>
<tr>
<td>Target application runtime with Intel Advisor compared to runtime without Intel Advisor</td>
<td>1.1x longer</td>
<td>3 - 8x longer</td>
<td>3.1 - 8.1x longer</td>
<td>5 - 100x longer</td>
<td>5 - 20x longer</td>
</tr>
</tbody>
</table>

Solutions
Use the following techniques to minimize overhead while collecting Intel Advisor analysis data. The Disabling additional analysis technique also minimizes finalization overhead.

Change Stackwalk Mode from Offline (After collection) to Online (During Collection)
Minimize collection overhead.
Applicable analysis: Survey.
Set to offline/after collection when:
• Survey analysis runtime overhead exceeds 1.1x.
• A large quantity of data is allocated on the stack, which is a common case for Fortran applications or applications with a large number of small, parallel, OpenMP* regions

To implement, do one of the following before/while running a Survey analysis:

• Set Project Properties > Analysis Target > Survey Hotspots Analysis > Advanced > Stack unwinding mode > During collection.
• Use the advixe-cl CLI action option --stackwalk-mode=online. For example:

```bash
advixe-cl --collect=survey --project-dir=./myAdvisorProj --stackwalk-mode=online -- ./bin/myTargetApplication
```

**Disable Stacks Collection**
Minimize collection overhead.

Applicable analyses: Roofline, Trip Counts and FLOP.

To implement, do one of the following before/while running the analysis:

• Disable the Vectorization Workflow pane > Enable Roofline with Callstacks checkbox.
• Disable the Project Properties > Analysis Target > Trip Counts and FLOP Analysis > Advanced > Collect stacks checkbox.
• Ensure the CLI action option --stacks is omitted from the command line. Alternative: Use the CLI action option -no-stacks.

**Disable Stitch Stacks**
Minimize collection overhead.

Applicable analysis: Survey.

The stitch stacks option restores a logical call tree for Intel® Threading Building Blocks (Intel® TBB) or OpenMP* applications by catching notifications from the runtime and attaching stacks to a point introducing a parallel workload.

Disable when Survey analysis runtime overhead exceeds 1.1x.

To implement, do one of the following before/while running the analysis:

• Disable the Project Properties > Analysis Target > Survey Hotspots Analysis > Advanced > Stitch stacks checkbox.
• Use the advixe-cl CLI action option --no-stack-stitching. For example:

```bash
advixe-cl --collect=survey --project-dir=./myAdvisorProj --no-stack-stitching -- ./bin/myTargetApplication
```

**NOTE**
Disabling stack stitching may decrease the overhead for applications using Intel TBB.

**Increase Sampling Interval**
Minimize collection overhead.

Applicable analysis: Survey.

Increase the wait time between each analysis collection sample when your target application runtime is long.

To implement, do one of the following before/while running the analysis:
• Increase the value in the Project Properties > Analysis Target > Survey Hotspots Analysis > Advanced > Sampling interval checkbox.

• Use the advixe-cl CLI action option --interval=<integer> when running a Survey analysis. For example:

```
advixe-cl --collect=survey --project-dir=./myAdvisorProj --interval=20 -- ./bin/myTargetApplication
```

**Limit Collected Analysis Data**

Minimize collection overhead.

Applicable analysis: Survey.

Decrease the amount of collected raw data when exceeding a size threshold could cause issues. For example: You have storage space limitations.

To implement, do one of the following before/while running the analysis:

• Decrease the value in the Project Properties > Analysis Target > Survey Hotspots Analysis > Advanced > Collection data limit, MB field.

• Decrease the value in the advixe-cl CLI action option --data-limit=<integer>. For example:

```
advixe-cl --collect=survey --project-dir=./myAdvisorProj --data-limit=250 -- ./bin/myTargetApplication
```

**Limit Loop Call Count**

Minimize collection overhead.

Applicable analysis: Dependencies, Memory Access Patterns.

Decrease the maximum number of instances each marked loop is analyzed.

To implement, do one of the following before/while running the analysis:

• Supply a non-zero value in the Project Properties > Analysis Target > [Name] Analysis > Advanced > Loop Call Count Limit field.

• Supply a non-zero value in the advixe-cl CLI action option --data-limit=<integer>. For example:

```
advixe-cl --collect=dependencies --project-dir=./myAdvisorProj --loop-call-count-limit=10 -- ./bin/myTargetApplication
```

**Disable Additional Analysis**

Minimize finalization overhead.

Applicable analysis: Survey.

Implement these techniques when the additional data is not important to you.

---

**NOTE**

The default setting for all the properties/options in the table below is disabled.
### CLI Action Options

<table>
<thead>
<tr>
<th>Path: Project Properties &gt; Analysis Target &gt; Survey Hotspots Analysis &gt; Advanced</th>
<th>CLI Action Options</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Disable the <strong>Analyze MKL loops and functions</strong> checkbox.</td>
<td><code>--no-mkl-user-mode</code></td>
<td>Do not show Intel® Math Kernel Library (Intel® MKL) loops and functions in Intel Advisor reports.</td>
</tr>
<tr>
<td>Disable the <strong>Analyze Python loops and functions</strong> checkbox.</td>
<td><code>--no-profile-python</code></td>
<td>Do not show Python* loops and functions in Intel Advisor reports.</td>
</tr>
</tbody>
</table>
| Disable the **Analyze loops that reside in non-executed code paths** checkbox. | `--no-support-multi-isa-binaries` | Do not collect a variety of data for loops that reside in non-executed code paths, including:  
  - Loop assembly code  
  - Instruction set architecture (ISA)  
  - Vector length  
  **NOTE**  
  This capability is available only for binaries compiled using the `-ax` (Linux* OS)/`/Qax` (Windows* OS) option with an Intel® compiler. |
| Disable the **Enable register spill/fill analysis** checkbox. | `--no-spill-analysis` | Do not calculate the number of consecutive load/store operations in registers and related memory traffic. |
| Disable the **Enable static instruction mix analysis** checkbox. | `--no-static-instruction-mix` | Do not statically calculate the number of specific instructions present in the binary. |

---

### See Also

- Collection Controls to Minimize Analysis Overhead
- Loop Markup to Minimize Analysis Overhead
- Filtering to Minimize Analysis Overhead
- Miscellaneous Techniques to Minimize Analysis Overhead

### Miscellaneous Techniques to Minimize Analysis Overhead

#### Issue

Running your target application with the Intel Advisor can take substantially longer than running your target application without the Intel Advisor. For example:
Runtime Overhead / Analysis | Survey | Trip Counts & FLOP | Roofline | Dependencies | MAP
--- | --- | --- | --- | --- | ---
Target application runtime with Intel Advisor compared to runtime without Intel Advisor | 1.1x longer | 3 - 8x longer | 3.1 - 8.1x longer | 5 - 100x longer | 5 - 20x longer

**Solutions**
The following techniques may help minimize overhead without limiting collection scope.

**Disable Cache Simulation**
Minimize collection overhead.

Applicable analyses:
- Memory Access Patterns (base simulation functionality)
- Trip Counts and FLOP (enhanced simulation functionality)

Implement these techniques when cache modeling information is not important to you:

**NOTE**
The default setting for all the properties/options in the table below is disabled.

<table>
<thead>
<tr>
<th>Path: Project Properties &gt; Analysis Target...</th>
<th>CLI Action Options</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Disable the <strong>Memory Access Patterns Analysis</strong> &gt; <strong>Advanced</strong> &gt; <strong>Enable Memory-Level Roofline with cache simulation</strong> checkbox.</td>
<td><code>--no-enable-cache-simulation</code></td>
<td>Do not model cache misses, cache misses and cache line utilization, or cache misses and loop footprint.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Path: Project Properties &gt; Analysis Target...</th>
<th>CLI Action Options</th>
<th>Description</th>
</tr>
</thead>
</table>
| Disable the **Trip Counts and FLOP Analysis** > **Advanced** > **Enable CPU cache simulation** checkbox. | `--no-enable-cache-simulation` | Do not:  
  - Model multiple levels of cache for data, such as counts of loaded or stored bytes for each loop.  
  - Create simulations for specific cache hierarchy configurations. |

**Limit Reported Data**
Applicable analysis: Memory Access Patterns.

Implement these techniques when the additional data is not important to you.

**NOTE**
The default setting for all the properties/options in the table below is enabled.
<table>
<thead>
<tr>
<th><strong>Project Properties &gt; Analysis Target &gt; Memory Access Patterns Analysis &gt; Advanced</strong></th>
<th><strong>CLI Action Options</strong></th>
<th><strong>Description</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>Disable the <strong>Report stack variables</strong> checkbox.</td>
<td>--no-record-stack-frame</td>
<td>Do not report stack variables for which memory access strides are detected.</td>
</tr>
<tr>
<td>Disable the <strong>Report heap allocated variables</strong> checkbox.</td>
<td>--no-record-mem-allocations</td>
<td>Do not report heap-allocated variables for which memory access strides are detected.</td>
</tr>
</tbody>
</table>

**Minimize Data Set**

Minimize collection overhead.

Applicable analyses: All, but especially Dependencies, Memory Access Patterns.

When you run an analysis, the Intel Advisor executes the target against the supplied data set. Data set size and workload have a direct impact on target application execution time and analysis speed.

For example, it takes longer to process a 1000x1000 pixel image than a 100x100 pixel image. A possible reason: You may have loops with an iteration space of 1...1000 for the larger image, but only 1...100 for the smaller image. The exact same code paths may be executed in both cases. The difference is the number of times these code paths are repeated.

You can control analysis cost without sacrificing completeness by minimizing this kind of unnecessary repetition from target application execution.

Instead of choosing large, repetitive data sets, choose small, representative data sets that minimize the number of instructions executed within a loop while thoroughly exercising target application control flow paths.

Your objective: In as short a runtime period as possible, execute as many paths as you can afford, while minimizing the repetitive computation within each task to the bare minimum needed for good code coverage.

Data sets that run in about ten seconds or less are ideal. You can always create additional data sets to ensure all your code is checked.

**Temporarily Disable Finalization**

Minimize finalization overhead.

Applicable analyses: Roofline, Survey, Trip Counts and FLOP.

Use when you plan to view collected analysis data on a different machine. Finalization automatically occurs when a result is opened in the GUI or a report is generated from the result.

To implement, do one of the following while running an analysis:

- When the analysis **Finalizing data...** phase begins, click the associated **Cancel** button.

  ![Finalizing data...](image)

- Use the `advixe-cl` CLI action option `--no-auto-finalize` when you run the desired analysis. For example:

  ```
  advixe-cl --collect=survey --project-dir=./myAdvisorProj --no-auto-finalize -- ./bin/myTargetApplication
  ```

**See Also**

*Collection Controls to Minimize Analysis Overhead*
Loop Markup to Minimize Analysis Overhead
Filtering to Minimize Analysis Overhead
Execution Speed/Duration/Scope Properties to Minimize Analysis Overhead

Analysis Summary

Purpose and Usage | Summary Regions

Analysis Summary Purpose and Usage
After running Intel® Advisor analyses (Survey, Trip Counts, FLOPS, Memory Access Patterns, Dependencies, and Suitability), consider reviewing a results summary that includes the most important information about your code. Analysis Summary data varies by workflow:

- **Vectorization Workflow Analysis Summary** - high-level information about Survey, Trip Counts, FLOPS, MAP, and Dependencies analyses.
- **Threading Workflow Analysis Summary** - high-level information about Survey, Trip Counts, FLOPS, Suitability, and Dependencies analyses.

Analysis Summary Regions

1. Select between result tabs.
2. View Summary for the selected workflow.

Vectorization Advisor - Analysis Summary
Purpose and Usage | Report Regions

Vectorization Summary Purpose and Usage
After running Intel® Advisor analyses (Survey, Trip Counts, FLOPS, Memory Access Patterns, Dependencies, and Suitability), consider reviewing a results summary that includes the most important information about your code. Click the Summary tab after running an analysis to view results.
Vectorization Summary Report Regions

- **Program metrics**: View the main performance metrics of your program, such as execution time statistics, vector instruction set (and whether extensions, such as VNNI, are used), and number of CPU threads utilized. The section is broken down into several sub-sections:
  - **Performance characteristics**: View execution time details, such as total CPU time and time spent in vectorized and scalar code.
    
    If your application uses Intel® Math Kernel Library (Intel® MKL), you will see the **MKL detail** button in the Performance characteristics section, which toggles two additional columns: the **User** column, which reports time spent in your code and corresponding compute metrics, and the **MKL** column, which reports time spent in the MKL code and corresponding compute metrics.
  - **Vectorization Gain/Efficiency**: View average estimated speedup of vectorized loops and total estimated program speedup.

  **NOTE**
  The vectorization efficiency data is available only for vectorized loops in modules compiled with an Intel® compiler version 16 or higher.

- **OP/S and Bandwidth**: View GFLOPS and GINTOPS usage and cache bandwidth metrics compared to hardware peak. Hover the mouse over the **Utilization** column and click the 🌟 button to select single-core or multicore benchmarks utilization metrics.

  **NOTE**
  The OP/S and bandwidth metrics are available after you run the Trip Counts and FLOP or the Roofline analysis.

- **Per program recommendations**: View suggested changes for your program that you might want to apply to achieve better performance.
- **Top time-consuming loops**: View top five time-consuming loops sorted by **self time** with performance metrics, such as execution time statistics and vectorization efficiency with comparison to original scalar loop efficiency.
- **Refinement analysis data**: View details about found dependencies and memory access patterns.

  The **Dependencies** column summarizes the predicted data sharing problems collected by the Dependencies tool. To display the **Dependencies Report** window at the corresponding parallel site location, click a function link in the **Site Location** column.

  The **Strides Distribution** column reports the memory access stride distribution within a loop in the ratio format in %: unit strides, constant strides, and variable strides.

  **NOTE**
  The information in the Refinement analysis data section is available only after you run the Memory Access Patterns or Dependencies analysis.

- **Recommendations**: View suggested changes with high confidence level for first five loops in the code that you might want to apply to achieve better performance. Click a recommendation link to access the recommendations texts.
- **Collection details**: View execution statistics for each of the collectors, as well as the **Collection Log**, **Application Output**, and **Collection Command Line** links that lead to the corresponding report logs, command line and output details.
NOTE
*Application Output* is available if you set output destination to Application Output window. To do this, go to **File > Options > General > Application Output Destination** and choose **Application Output window**.

- **Platform information**: View the system information including software and hardware summary.

**See Also**
- Data Reference

**Threading Advisor - Analysis Summary**

**Purpose and Usage | Report Regions**

**Threading Summary Purpose and Usage**

After running Intel® Advisor analyses (Survey, Trip Counts, FLOPS, Memory Access Patterns, Dependencies, and Suitability), consider reviewing a results summary that includes the most important information about your code. Click the **Summary** tab after running an analysis to view results.

**Threading Summary Report Regions**

- **Program metrics**: View the main performance metrics of your program, such as execution time statistics, vector instruction set (and whether extensions, such as VNNI, are used), and number of CPU threads utilized. The section is broken down into several sub-sections:
  - **Performance characteristics**: View execution time details, such as total CPU time and time spent in vectorized and scalar code.
    
    If your application uses Intel® Math Kernel Library (Intel® MKL), you will see the **MKL detail** button in the **Performance characteristics** section, which toggles two additional columns: the **User** column, which reports time spent in your code and corresponding compute metrics, and the **MKL** column, which reports time spent in the MKL code and corresponding compute metrics.
  - **Vectorization Gain/Efficiency**: View average estimated speedup of vectorized loops and total estimated program speedup.

**NOTE**

The vectorization efficiency data is available only for vectorized loops in modules compiled with an Intel® compiler version 16 or higher.

- **OP/S and Bandwidth**: View GFLOPS and GINTOPS usage and cache bandwidth metrics compared to hardware peak. Hover the mouse over the **Utilization** column and click the 🌟 button to select single-core or multicore benchmarks utilization metrics.

**NOTE**

The OP/S and bandwidth metrics are available after you run the Trip Counts and FLOP analysis.

- **Per program recommendations**: View suggested changes for your program that you might want to apply to achieve better performance.
- **Top time-consuming loops**: View top five time-consuming loops sorted by total time with performance metrics, such as execution time statistics and vectorization efficiency with comparison to original scalar loop efficiency.
• **Suitability and Dependencies analysis data**: View information about predicted sharing problems for annotated parallel sites. The **Maximum Site Gain** column summarizes potential performance improvement achieved through threading. The **Dependencies** column summarizes the predicted data sharing problems. To display the Dependencies Report window at the corresponding parallel site location, click a function link under the Site Location column.

**NOTE**
This information is available only after you run the Suitability or Dependencies analysis.

• **Recommendations**: View suggested changes with high confidence level for first five loops in the code that you might want to apply to achieve better performance. Click a recommendation link to access the recommendations texts.

• **Collection details**: View execution statistics for each of the collectors, as well as the **Collection Log**, Application Output, and Collection Command Line links that lead to the corresponding report logs, command line and output details.

**NOTE**
Application Output is available if you set output destination to Application Output window. To do this, go to File > Options > General > Application Output Destination and choose Application Output window.

• **Platform information**: View the system information including software and hardware summary.

**See Also**
• Data Reference

**Intel® Advisor User Guide: Offload Advisor**

Intel® Advisor Beta is an extended Intel® Advisor: a code modernization, programming guidance, and performance estimation tool that supports the DPC++ language and OpenMP* on CPU and GPU. It provides co-design, performance modeling, analysis, and characterization features for industry-size applications (in C, C++, Fortran, and mixed Python*).

Specifically, Intel® Advisor Beta includes Offload Advisor, a tool that allows you to collect performance predictor data in addition to the profiling capabilities of Intel Advisor. Use Offload Advisor to determine what code can be offloaded to a target device (for example, to a GPU), accelerating the performance of your CPU-based application.

**NOTE**
This section explains how to work with the Offload Advisor tool from the Intel® Advisor Beta command-line interface. To run the Offload Advisor analysis from the GUI (technical preview), set the ADVIXE_EXPERIMENTAL=beta_gui variable. For more information, see Intel® Advisor User Guide (PDF).

Offload Advisor provides metrics and performance data such as projected speedup, a call tree showing offloaded and accelerated regions, and more. It takes into account not only compute and memory limitations, but the time required to transfer data if the code is offloaded.

Follow these steps to set up, use, and interpret results using the Offload Advisor. If you would like to skip to a specific step, click the corresponding link:

1. **Before You Begin**
2. Collect Performance Metrics
3. Run Performance Modeling
4. Review the Performance Predictor Output
5. Review the Command Option Reference
6. (Optional) Once your application includes code that has been offloaded to the target device, you can use Intel Advisor to view a Roofline chart of the GPU code. For steps, see Identify GPU Performance Bottlenecks Using GPU Roofline.

As an extended Intel Advisor, Intel® Advisor Beta includes the Vectorization Advisor, Threading Advisor, and Flow Graph Analyzer tools.

- **Vectorization Advisor** is a vectorization optimization tool that lets you identify high-impact, under-optimized loops, what is blocking vectorization, and where it is safe to force vectorization. It also provides code-specific how-can-I-fix-this-issue recommendations.
- **Roofline Analysis** visualizes actual performance against hardware-imposed performance ceilings (rooflines). It provides insights into where the bottlenecks are, which loops are worth optimizing for performance, what are the likely causes of bottlenecks and what should be the next optimization steps.
- **Threading Advisor** is a fast-track threading design and prototyping tool that lets you analyze, design, tune, and check threading design options without disrupting your normal development.
- **Flow Graph Analyzer** is a visual prototyping tool that lets you represent and analyze performance for applications that use the Intel® Threading Building Blocks (Intel® TBB) flow graph interfaces.
- **Offload Advisor** (Intel® Advisor Beta only) allows you to identify high-impact opportunities to offload to a target device as well as the areas that are not advantageous to offload. It provides performance speedup projection on target devices along with offload overhead estimation and pinpoints performance bottlenecks.

For details about the Vectorization Advisor and Threading Advisor, see the Intel Advisor User Guide. For details about Flow Graph Analyzer, see the Flow Graph Analyzer User Guide.

### Before You Begin

Visit the Release Notes page for known issues and the most up-to-date information.

### Prepare Your Application

The only strict requirement for compilation and linking is full debug information. However, the optimization level is critical for both code generation and interpretation. Intel® Advisor Beta functions at any optimization level, but the following settings are considered the optimal requirements:

<table>
<thead>
<tr>
<th>To Do This</th>
<th>Optimal Compiler Settings</th>
</tr>
</thead>
<tbody>
<tr>
<td>Request full debug information (compiler and linker).</td>
<td>Linux* OS option: –g</td>
</tr>
<tr>
<td></td>
<td>Windows* OS option:</td>
</tr>
<tr>
<td></td>
<td>• /ZI</td>
</tr>
<tr>
<td></td>
<td>• /DEBUG</td>
</tr>
<tr>
<td>Request moderate optimization.</td>
<td>Linux* OS option: –02 or higher</td>
</tr>
<tr>
<td></td>
<td>Windows* OS option: /02 or higher</td>
</tr>
<tr>
<td>Disable interprocedural optimizations that may inhibit the ability of Intel® Advisor Beta to</td>
<td>Linux* OS option: –no-ipo</td>
</tr>
<tr>
<td>collect performance data. Use this with Intel® C++ Compiler Classic and Intel® Fortran Compiler</td>
<td>Windows* OS option: /Qipo-</td>
</tr>
<tr>
<td>Classic only.</td>
<td></td>
</tr>
</tbody>
</table>
Set Up the Intel® Advisor Beta Environment

**NOTE**
On macOS*, you can only view results collected on a Linux* or Windows* OS.

To set up the Intel® Advisor Beta environment, run the **vars** script:

- On Linux* OS:
  ```bash
  source <install-dir>/env/vars.sh
  ```
- On Windows* OS:
  ```bash
  <install-dir>/env/vars.bat
  ```
- On macOS*:
  ```bash
  source <install-dir>/env/vars.sh
  ```

This script sets all required environment variables, including **APM**, which points to `<install-dir>/perfmodels`. This is the location of the Offload Advisor scripts in the Intel® Advisor Beta installation directory.

<install-dir> is the installation path for the Intel® Advisor Beta. By default, it can be found inside the following:

- On Linux OS:
  ```plaintext
  /opt/intel/oneapi for root users
  $HOME/intel/oneapi for non-root users
  ```
- On Windows OS: C:\Program Files (x86)\Intel\oneAPI\n- On macOS*: /opt/intel/oneapi

Collect Performance Metrics

In the Offload Advisor workflow, you must (1) run **performance profiling** to collect data about your application, and (2) run **performance modeling** and generate your report results. These steps are generally performed using the **collect.py** and **analyze.py** scripts, respectively.

The unified data collection method below uses the `run_oa.py` script, which combines both scripts, but if you choose to run data collection separately, you must still use `analyze.py` to obtain report results.

You can use one of the following methods to run performance profiling. The methods vary in simplicity and flexibility. Based on how detailed the analysis needs to be, choose the method that suits you best:

- **Unified data collection and output generation.** This is the simplest, but less flexible method. It runs both performance profiling and performance modeling.
- **Separate data collection using collect.py.** This method is simple and moderately flexible. With this method, you must run the performance modeling script separately to generate output.
- **Separate data collection using advixe-cl.** This method is the most flexible. With this method, you must run the performance modeling script separately to generate output. This method supports MPI applications.

**Unified Data Collection and Output Generation Using run_oa.py**

**NOTE** This script does not support MPI applications.
The recommended method to get started with Intel® Advisor Beta is to use the run_oa.py script, which collects performance predictor data and generates performance predictor output using a set of default recommended settings. To run the script, you can use the following command:

```
advixe-python <APM>/run_oa.py <project-dir> [--options] -- <target> [target-options]
```

where `<APM>` is the Offload Advisor environment variable that points to script directory. Replace it with `$APM` on Linux* OS or `%APM%` on Windows* OS.

By default, run_oa.py marks up all regions and selects only the most profitable ones for analysis. To analyze existing parallel regions only, use `--markup regions` option as follows:

```
advixe-python <APM>/run_oa.py <project-dir> --markup regions [--options] -- <target> [target-options]
```

Once the script has run, you can view your performance projection results, which are located by default in `<advisor-project>/perf_models/mNNNN`. For details about the output generated, see the Performance Predictor Output.

For details about markup rules, refer to Troubleshooting Code Region Is Not Marked Up.

**See Also**

- run_oa.py Options
- Performance Predictor Output

### Separate Data Collection Using collect.py

**NOTE**

This script does not support MPI applications.

By default, collect.py launches Survey, Trip Counts and FLOP, and Dependencies profiling steps one by one. It performs a smart region selection and applies the analyses selectively to the regions of interest. To run the script, you can use the following command:

```
advixe-python <APM>/collect.py <project-dir> [--options] -- <target> [target-options]
```

where `<APM>` is the Offload Advisor environment variable that points to script directory. Replace it with `$APM` on Linux* OS or `%APM%` on Windows* OS.

For details about markup rules, refer to Troubleshooting Code Region Is Not Marked Up.

To disable dependencies collection, for example, to decrease collection overhead, you can use the following command:

```
advixe-python <APM>/collect.py <project-dir> --collect basic [--options] -- <target> [target-options]
```

Once you have run collect.py, continue to run performance modeling with analyze.py. For steps, see Run Performance Modeling.

**See Also**

- collect.py Options
- Run Performance Modeling

### Separate Data Collection Using advixe-cl

Use separate data collection with advixe-cl when collect.py is not applicable, for instance, for multi-rank MPI workloads.
To get the advixe-cl commands appropriate for your application, you can run the collect.py script with the --dry-run option. It prints advixe-cl command lines ready for copying and pasting. For example:

```
advice-python <APM>/collect.py <project-dir> --config <config-file> --dry-run -- <target> [target-options]
```

where <APM> is the Offload Advisor environment variable that points to script directory. Replace it with $APM on Linux* OS or %APM% on Windows* OS.

For details about advixe-cl options, refer to Command Line Interface Reference in the Intel® Advisor User Guide.

**Important** After running the advixe-cl --collect=survey [options] command, mark up regions for analysis with the --markup generic option of collect.py.

To collect data with advixe-cl, run the printed commands one by one. For example, the typical workflow might be as follows:

1. Run Survey collection:
   ```
advice-cl --collect=survey --auto-finalize --stackwalk-mode=online --static-instruction-mix --project-dir=<project-dir> -- <target> [target-options]
```

2. Run the collect.py script to mark up specific regions with --markup <markup_type> option. For example, use --markup generic to mark up all regions and select the most profitable for analysis:
   ```
advice-python <APM>/collect.py <project-dir> --markup generic
```

3. Run Trip Counts and FLOP collection:
   ```
```

4. [Optional]: Run Dependencies collection. This improves model accuracy but increases analysis overhead. Dependencies collection is not required for analyzing existing parallel regions but strongly recommended for generic analysis, which is the default mode. For example, you can use the following command to run Dependencies collection:
   ```
advice-cl --collect=dependencies --filter-reductions --loop-call-count-limit=16 --project-dir=<project-dir> -- <target> [target-options]
```

To collect data for MPI workloads with Intel® Advisor Beta, you need to adjust the command examples above using the mpiexec with the advixe-cl command. The general form of the mpiexec command is:

```
$ mpiexec -n <N> "advixe-cl --collect=analysis-type --project-dir=<project-dir> --search-dir src:r=<sources-dir> [options]" <target> [target-options]
```

For example, to collect survey data for all ranks into the shared ./advi project directory, run the following command:

```
$ mpirun -n 4 "advixe-cl --project-dir=./advi --collect=survey" myApplication
```

To collect survey data for a single rank only, run the following command:

```
$ mpirun -n 4 -gtool "advixe-cl --project-dir=./advi --collect=survey :0" myApplication
```

For more information about analyzing MPI applications, refer to MPI Workloads.
NOTE

- For multi-process workloads that include a parent script, Python* or other process and a child process with usercode of interest: if you want to analyze a child process, you must additionally specify its name with the --executable-of-interest option when running the Dependencies collection:

```bash
advixe-cl --collect=dependencies --executable-of-interest=<child-binary> ...
```

This is not required for the Survey and Trip Counts and FLOP steps because, by default, Survey and Trip Counts and FLOP profile all processes.

- Attaching to a process is possible for Survey, but is not supported by Trip Counts and FLOP and Dependencies collections.

- Using ittnotify marked-up binaries and launching analysis in a start-paused mode are supported for Survey and Trip Counts and FLOP steps only. For more information, refer to Pause/Resume Collection Using API Methods.

Once you have run collect.py, continue to run performance modeling with analyze.py. For steps, see Run Performance Modeling.

See Also

- Run Performance Modeling
- Pause/Resume Collection Using API Methods
- MPI Workloads
- Command Line Interface Reference

Run Performance Modeling

Run `analyze.py` to predict your application performance on a target device (for example, on a GPU) and generate Intel® Advisor Beta output results:

```bash
advixe-python <APM>/analyze.py <project-dir> [--options]
```

where `<APM>` is the Offload Advisor environment variable that points to script directory. Replace it with `$APM` on Linux* OS or `%APM%` on Windows* OS.

NOTE

- Traditional, not just-in-time (JIT) compiled, Intel® Math Kernel Library (Intel® MKL) code is supported by all analysis types and can be an offload candidate like any other code assuming it is compiled with a general-purpose compiler.

- Intel® Math Kernel Library for Deep Neural Networks (Intel® MKL-DNN) code is JIT compiled and is not supported by the Dependencies analysis. It can be an offload candidate assuming it is compiled with a general-purpose compiler.

Performance Modeling Configuration

By default, Intel® Advisor Beta collections and analysis are optimized to model performance on an integrated Intel® Processor Graphics Gen11. You can switch to integrated Intel® Processor Graphics Gen9 configuration by providing an alternate hardware definition model .toml file as an input to the `analyze.py` and `collect.py` / `run_oa.py`. For example, for data collection with `collect.py` and performance modeling with `analyze.py` for the Intel® Processor Graphics Gen9 architecture:
advixe-python <APM>/collect.py <project-dir> --config gen9 [--options] -- <target> [target-options]

advixe-python <APM>/analyze.py <project-dir> [--options]

You can also use the sliders in a report.html file to generate a custom hardware configuration file. Refer to Performance Predictor Report.

**View Performance Projection Results**

Once you have run performance modeling with analyze.py, your results can be found by default in <advisor-project>/perf_models/mNNNN. For an overview of the results provided, see Performance Predictor Output Overview.

**See Also**

- analyze.py Options
- Troubleshooting Reference

**Manage Invocation Taxes**

When Offload Advisor detects high call count value for a potentially profitable code region, it assumes that the kernel invocation tax is paid as many times as the kernel is launched. This results in high invocation tax and cost of offloading, which means that this code region cannot benefit from offloading. This is a pessimistic assumption.

However, for simple applications where there is no need to wait for a kernel instance to finish, this cost can be hidden every time except the very first one.

To reflect these two approaches, the kernel invocation tax is reported in two columns in the Offload Advisor HTML report:

- **Invocation Tax** reports the estimated time, in seconds, spent on launching each kernel instance with no taxes hidden.
- **Configuration Tax** reports the estimated time, in seconds, spent on launching only the first instance of a kernel with all other taxes hidden.

You can tell Offload Advisor how to handle invocation taxes for your application when modeling its performance on a target device:

**Hide All Taxes**

For simple applications, you are recommended to enable the optimistic approach for estimating invocation taxes. In this approach, Offload Advisor assumes the invocation tax is paid only for the first time the kernel executes.

To enable this approach, use the --assume-hide-taxes option with the analyze.py, for example:

```
advixe-python <APM>/analyze.py ./advi --assume-hide-taxes [--options]
```

where <APM> is the Offload Advisor environment variable that points to script directory. Replace it with $APM on Linux* OS or %APM% on Windows* OS.

With this option, the tax is reported in the Configuration Tax column only, and the Invocation Tax column reports 0.

**Do Not Hide Taxes**

By default, Offload Advisor estimates invocation taxes using the pessimistic approach and assumes the invocation tax is paid each time the kernel is launched.
You can use the `--assume-never-hide-taxes` option with the `analyze.py` to explicitly tell the Offload Advisor not to hide any invocation taxes. For example:

```
advixe-python <APM>/analyze.py ./advi --assume-never-hide-taxes [--options]
```

where `<APM>` is the Offload Advisor environment variable that points to script directory. Replace it with `$APM` on Linux* OS or `%APM%` on Windows* OS.

With this option, the tax is reported in the **Invocation Tax** column only, and the **Configuration Tax** column reports 0.

**Fine-Tune the Number of Hidden Taxes**

You can fine-tune the number of invocation taxes to hide by specifying the `Invoke_tax_ratio` parameter and a fraction of invocation taxes to hide in a TOML configuration file.

1. Create a new TOML file, for example, `my_config.toml`. Copy and paste the following text there:

```
[scale]
# Fraction of invocation taxes to hide.
# Note: The invocation tax of the first kernel instance is not scaled.
# Possible values: 0.0--1.0
# Default value: 0.0
Invoke_tax_ratio = <float>
```

Where `<float>` is a fraction of invocation taxes to hide, for example, `Invoke_tax_ratio = 0.95`, which means that 95% of invocation taxes will be hidden and only 5% of the taxes will be estimated.

2. Save and close the file.

3. Run the performance projection with the new TOML file:

```
advixe-python <APM>/analyze.py ./advi --config my_config.toml [--options]
```

**Important** If you use the configuration parameter to control the number of taxes to hide, **do not use** the `--assume-hide-taxes` or `--assume-never-hide-taxes` option. These options overwrite the value of the configuration parameter.

In the generated HTML report, the **Configuration Tax** column reports the tax paid only for the first time the kernel is executed in the specified fraction, and the **Invocation Tax** column reports the rest of the taxes assuming it is paid each time the kernel executes.

**Tip** If you want to model performance for a specific accelerator using a pre-defined configuration file and apply the invocation tax configuration parameter to it, you can specify several configuration files. For example, to model performance on an integrated Intel® Processor Graphics Gen9 configuration with the custom configuration tax, use the following command:

```
advixe-python <APM>/analyze.py ./advi --config gen9_gt2 --config my_config.toml [--options]
```

**Minimize Collection Overhead**

Running your target application with the Offload Advisor tool of the Intel® Advisor Beta can take substantially longer than running your target application without the Offload Advisor. Use the following techniques to minimize overhead when collecting data.
Pause/Resume Collection Using API Methods

**NOTE**
This is applicable only to Survey, Trip Counts, and FLOP analyses.

To minimize collection overhead, you can pause/resume collection using the corresponding `__itt_pause` and `__itt_resume` methods of the Instrumentation and Tracing Technology API (ITT API). This is particularly useful when:

- You do not want to analyze one or more uninteresting parts of your target application.
- The interesting parts of your target application involve large workloads.
- Modifying/recompiling your target application is not an issue.

To attach ITT APIs to a launched application, that is, to collect API data on an application that is already launched, be sure to point the target application to the `ittnotify_collector` library using an environment variable:

- **Windows* OS:**
  ```
  set INTEL_LIBITTNOTIFY32=<install_dir>\bin32\runtime\ittnotify_collector.dll
  set INTEL_LIBITTNOTIFY64=<install_dir>\bin64\runtime\ittnotify_collector.dll
  ```
- **Linux* OS:**
  ```
  export INTEL_LIBITTNOTIFY32=<install_dir>/lib32/runtime/libittnotify_collector.so
  export INTEL_LIBITTNOTIFY64=<install_dir>/lib64/runtime/libittnotify_collector.so
  ```

  **NOTE**
  Use the full path to the library without quotations marks.

To use pause/resume collection controls when analyzing your application:

1. Add the following statements to every source file you want to instrument to enable the ITT API:
   - **C/C++:** `#include <ittnotify.h>
   - **Fortran***: `USE ITTNOTIFY`

2. Insert `__itt_pause (C/C++)` or `CALL ITT_PAUSE (Fortran)` before uninteresting parts of your target application and the `__itt_resume (C/C++)` or `CALL ITT_RESUME (Fortran)` before interesting parts of your target application.

   For example, the following snippet starts collection in paused mode and skips initialization, collects data in the middle, then pauses collections and skips finalization:

   ```
   #include <ittnotify.h>
   int main(int argc, char* argv[])
   {
     __itt_pause();
     // Do uninteresting work here
     __itt_resume();
     // Do work here
     __itt_pause();
     // Do uninteresting work here
     return 0;
   }
   ```
3. Run the desired analysis using the CLI action `--collect=<analysis>` with the `--start-paused` option to enable the `__itt_pause` and `__itt_resume` methods in your code. For example, for the Survey analysis:

```
advixe-cl --collect=survey --project-dir=./myAdvisorProj --start-paused -- ./bin/myTargetApplication
```

For the details about using ITT API methods to pause/resume collection, refer to Collection Controls to Minimize Analysis Overhead section in the Intel® Advisor User Guide.

You can find the open source version of ITT API source code in the official Intel® ITT repository at https://github.com/intel/ittapi.

**Disable Stacks Collection**

To minimize collection overhead, you can disable collecting data distribution over stacks with the `--no-stacks` option, with the potential tradeoff of modeling accuracy. You have two ways to use the `--no-stacks` option:

- **Disable collecting data distribution over stacks.** Run `run_oa.py`, `collect.py`, or `advixe-cl` (with `--collect=tripcounts`) with the `--no-stacks` option to minimize collection overhead. This may reduce modeling accuracy.

  **NOTE**
  
  If you collect data with `collect.py` or `advixe-cl`, you do not need to use this options with `analyze.py`, as it will be used automatically because the collection will not include callstack-attributed data.

- **Run data analysis without callstacks data.** If you collected callstack-attributed data with `collect.py` or `advixe-cl` (with `--collect=tripcounts`), but callstack attribution went wrong, disable using callstacks data for analysis with `analyze.py` to avoid using the wrong data. This is a possible fallback when data with stacks is broken.

  **NOTE**
  
  This reduces modeling accuracy.

**See Also**

Command Options Reference. This reference section describes the command line options available for each of the Offload Advisor Python* scripts.

advixe-cl Command Option Reference: stacks | no-stacks

**Analyze DPC++, OpenCL™, and OpenMP* Target Applications with Offload Advisor**

If your application is written in DPC++, OpenCL™, or OpenMP* with `pragma omp target` (for C++) or `directive omp target` (for Fortran), and the application contains code that is already offloaded to a target device, you may want to analyze it with the Offload Advisor and model potential performance gains from offloading to a different target device. For example, you have an application with code offloaded to an integrated GPU, but wish to model performance on a discrete GPU.

To do this, make use of CPU offload profiling. In this approach, code is temporarily offloaded to the CPU to project application performance on different hardware. You can use this approach to profile your code with Advisor, or make performance projections with Offload Advisor. **Enabling GPU profiling is not required when using the CPU offload feature.**
For OpenMP*

1. Disable offloading or set the offload target to the CPU. To do this, use environment variables:
   - Set the control default device using `OMP_DEFAULT_DEVICE`, which determines the device number used in device constructs. Non-negative integer values are accepted.
   - Set the execution mode using the following environment variables:
     - `OMP_TARGET_OFFLOAD=MANDATORY` and `LIBOMPTARGET_DEVICETYPE=CPU` enable offloading the target region code to run on a CPU.
     - `OMP_TARGET_OFFLOAD=DISABLED` disables code offloading; the code runs natively on the CPU.

   For example, you can ensure that kernels are ‘offloaded’ to the CPU using the commands:
   ```bash
   export OMP_TARGET_OFFLOAD=DISABLED
   export LIBOMPTARGET_DEVICETYPE=CPU
   ```

   **NOTE** By default, the environment variables are set to `OMP_TARGET_OFFLOAD=MANDATORY` and `LIBOMPTARGET_DEVICETYPE=GPU` for OpenMP target applications to be offloaded to a GPU.

2. Set the `INTEL_JIT_BACKWARD_COMPATIBILITY` environment variable to 1.

3. When collecting performance metrics and modeling performance with Offload Advisor, use the `--jit` option.

   **NOTE** The `--jit` option automatically enables `--assume-hide-taxes` for performance modeling, which hides all invocation taxes except the first one in the report. For details, see Manage Invocation Taxes.

You can then execute the application as usual. For sample commands to run performance modeling, see Collect Performance Metrics.

For additional information on environment variables, see Get Started Using the OpenMP* Offload to GPU Feature.

For DPC++

1. Specify the CPU as the target device in the application source code by using `sycl::host_selector`.

2. Set the `INTEL_JIT_BACKWARD_COMPATIBILITY` environment variable to 1.

3. When collecting performance metrics and modeling performance with Offload Advisor, use the `--jit` option.

   **NOTE** The `--jit` option automatically enables `--assume-hide-taxes` for performance modeling, which hides all invocation taxes except the first one in the report. For details, see Manage Invocation Taxes.

You can then execute the application as usual. For sample commands to run performance modeling, see Collect Performance Metrics.

For additional information about SYCL, see https://www.khronos.org/registry/SYCL/specs/sycl-1.2.1.pdf.

For OpenCL™

1. Set the `INTEL_JIT_BACKWARD_COMPATIBILITY` environment variable to 1.

2. Configure your OpenCL code to be offloaded to a CPU. Refer to the OpenCL documentation at https://www.khronos.org/registry/OpenCL/ for specific instructions.
3. When collecting performance metrics and modeling performance with Offload Advisor, use the `--jit` option.

**NOTE** The `--jit` option automatically enables `--assume-hide-taxes` for performance modeling, which hides all invocation taxes except the first one in the report. For details, see Manage Invocation Taxes.

You can then execute the application as usual. For sample commands to run performance modeling, see Collect Performance Metrics.

**Performance Predictor Output Overview**

The default Intel® Advisor Beta output with performance predictions includes:

- The main report in HTML format, named as `report.html`.
- Two comma-separated CSV files, named as `report.csv` and `whole_app_metric.csv`.
- A graphical representation of the call tree showing the offloadable and accelerated regions, named as `program_tree.dot`.
- A graphical representation of the call tree, named as `program_tree.pdf`, which is generated if a DOT utility is installed on your system. It is the result of 1:1 conversion from the `program_tree.dot` file.

**NOTE** The DOT utility is usually a part of a GraphViz® package.

- JSON and LOG files that contain data used to generate the HTML report, as well as logs. These files are primarily used for debugging and when reporting bugs and issues.

**Call Tree Graphical Representation**

The `program_tree.pdf` file is a graphical representation of the application call tree.

- **Circles** represent loops and functions.
- **Dotted circles** represent loops and functions excluded from analysis. This usually happens because of missing information, or because something prevents their direct offload. Each dotted circle includes a reason why this loop or functions was not analyzed.
- **Dotted rectangle regions** represent offloadable regions, which are candidates for offloading to a target device.
- **Rectangle regions with grey background** represent accelerated regions, which are optimal offloads within offloadable regions. These grey rectangles also list metrics for each particular offload, such as original execution time, execution time on a target device, estimated speedup.

The figure below shows a selected fraction of a complete file. In this section, there are three offload regions marked by grey boxes. All three regions correspond to triply-nested loops. The offload region number 13 (on the right) comes from the file `stream.cpp`, while the other two come from the file `collision.cpp`. Focusing on the region on the right of the figure, you can see the following reported metrics:

- The speedup for this region's offload is 31x.
- The loop nest head is at line 21.
- The child loops start at lines 22 and 26 respectively.

Similar information is available for all the marked regions.
Performance Predictor Report

Report Overview | Summary | Offloaded Regions | Non-Offloaded Regions | Call Tree | Configuration | Logs

Performance Predictor Report Overview

The report.html file contains multiple sections: Summary, Offloaded Regions, Non-Offloaded Regions, Call Tree, Configuration, Logs. You can switch between different sections using the links in the top left box. The top right box highlights total speedup, number of loops and functions offloaded, and a fraction of code accelerated.

By default, the report opens the Summary section.

Summary Section

The Summary section includes the most important information about your program and regions analyzed and allows you to adjust the configuration parameters. It is broken down into several panes:
• **Program metrics**: View the main performance metrics of your program, such as the number of offloads, speedup, and time spent on a target device. The bar chart above visually compares the original execution time spent on a base platform (for example, on a CPU) with the estimated execution time on a target device. The chart on the right breaks down the estimated time by specific tasks, including offload taxes, in percentages.

• **Offloads bounded by**: View the fractions of offloaded code bounded by specific limitations, in percentages.

• **Configuration**: View the hardware configuration used to generate performance prediction for your application.

  • To customize the target device configuration for performance projection, move the sliders (shown in the image above, in the top right).
  • To use your custom configuration, save the configuration file by clicking the **Download configuration** file icon and re-run the `analyze.py` or `run_oa.py` performance projection script with this file using the `--config <CONFIG>` option. The file will be saved as `scaler.toml`. For example, to run `analyze.py` with the custom configuration file:

    ```bash
    advixe-python <APM>/analyze.py ./advi --config <path-to-config>/scalers.toml
    ```

    where `<APM>` is the Offload Advisor environment variable that points to script directory. Replace it with `$APM` on Linux* OS or `%APM%` on Windows* OS.

**NOTE**

Some of the parameters require running performance collection (`collect.py`) in addition to the performance projection with `analyze.py`. Refer to the comments in the Configuration section of the report or in the downloaded configuration file for more information.

• To reset a slider position to default, click an arrow icon below a specific slider. To reset all sliders to default, click the **Reset to default** icon.

  • To go to the **Configuration** section of the report, click the **Open configuration page** icon.

• **Top Offloaded**: View the top five offloaded code regions sorted by speedup and performance metrics. Hover over a histogram in the Bounded By column to see time breakdown with color legend. Click a code region to open it in the **Offloaded Regions** section.

• **Top Non-Offloaded**: View the top five non-offloaded code regions, performance metrics, and the reason why they were not offloaded. Hover over a histogram in the Execution Time column to see time breakdown with color legend. Click a code region to open it in the **Non-Offloaded Regions** section. For details about reasons for not offloading and possible solutions, refer to Troubleshooting Why Not Offloaded.

**Offloaded Regions Section**

The **Offloaded Regions** section provides detailed information on all of the offloaded regions of the code. By default, the code regions are sorted by total execution time.

• Review the performance metrics and useful information about offloaded loops in the **metrics table**. For details about metrics and the pane controls, see Reported Metrics.

• View the source code associated with a specific function or loop in the **Source Code** pane (on the right). To see source code, if available, click a specific loop in the **Hierarchy** column of the metrics table.

• Review the **memory objects** used in a code region. To see details about tracked memory objects, if any, click a specific code region in the **Hierarchy** column of the metrics table. The memory objects pane includes the following:
• The table reports details about each memory object tracked in the selected code region. By default, the table is sorted by memory object size. Hover over a column header and click the menu icon to filter values or autosize columns.
• The memory objects histogram shows a distribution of memory objects by size.
• View a hierarchical listing of functions and loops in your code in the Hierarchy pane (on the right). To locate a specific loop, click its name in the Hierarchy column of the metrics table.

Non-Offloaded Regions Section
The Non-Offloaded Regions section provides detailed information on all of the regions of the code that were not offloaded and the reason why they were not offloaded. By default, the code regions are sorted by total execution time.
• Review the performance metrics and useful information about non-offloaded loops in the metrics table. For details about metrics and the pane controls, see Reported Metrics.
• View the source code associated with a specific function or loop in the Source Code pane (on the right). To see source code, if available, click a specific loop in the Hierarchy column of the metrics table.
• Review the memory objects used in a code region. To see details about tracked memory objects, if any, click a specific code region in the Hierarchy column of the metrics table. The memory objects pane includes the following:
  • The table reports details about each memory object tracked in the selected code region. By default, the table is sorted by memory object size. Hover over a column header and click the menu icon to filter values or autosize columns.
  • The memory objects histogram shows a distribution of memory objects by size.
  • View a hierarchical listing of functions and loops in your code in the Hierarchy pane (on the right). To locate a specific loop, click its name in the Hierarchy column of the metrics table.

Call Tree Section
The Call Tree Regions section provides a top-down view of the code and performance metrics for loops and functions. The page contains several sections:
• Review the performance metrics and useful offload information about all code regions in the metrics table. For the details about metrics and the pane controls, see Reported Metrics.
• View the source code associated with a specific function or loop in the Source Code pane (on the right). To see source code, if available, click a specific loop in the Hierarchy column of the metrics table.
• Review the memory objects used in a code region. To see details about tracked memory objects, if any, click a specific code region in the Hierarchy column of the metrics table. The memory objects pane includes the following:
  • The table reports details about each memory object tracked in the selected code region. By default, the table is sorted by memory object size. Hover over a column header and click the menu icon to filter values or autosize columns.
  • The memory objects histogram shows a distribution of memory objects by size.
  • View a hierarchical listing of functions and loops in your code in the Hierarchy pane (on the right). To locate a specific loop, click its name in the Hierarchy column of the metrics table.

Configuration Section
The Configuration section shows the source code of a commented configuration file in a read-only format. Review the comments for detailed description and possible values of the parameters. To change the values of the configuration parameters, use the sliders in the Summary section.
Logs Section
The Logs section provides a command line used to run the analysis and all output messages reported in console during the script(s) execution. This section reports four types of messages: Error, Warning, Info, and Debug.

- By default, only Error, Warning, and Info messages are shown. To control types of messages shown, hover over the Severity column header and click the menu button to open filters pane.
- To filter output messages, hover over the Message column header, click the menu button to open filters pane, and type a filter text.
- By default, the messages are reported in the order of their appearance in console during the script(s) execution. To group the messages by type, click the table header.

Command Option Reference
This reference section describes the command line options available for each of the Offload Advisor Python* scripts.

To use the Offload Advisor, run one or two of the following scripts, depending on a method you chose:
- run_oa.py Options
- collect.py Options
- analyze.py Options

Command Syntax
The syntax for Offload Advisor commands is as follows:

```
advixe-python <APM>/<script-name>.py <project-dir> [--options] [-- <target> [target-options]]
```

where:

- advixe-python A call to the Intel Advisor Python* command line tool.
  `advixe-python` is recommended to run the Offload Advisor Python scripts. The main advantage of using this command line tool is that it does not require you to install a specific Python version on your system because it calls to an internal Python version of the Intel Advisor.

- `<APM>` The environment variable that points to the directory with the Intel® Advisor Beta scripts. Replace it with:
  - `$APM` on Linux* OS
  - `%APM%` on Windows* OS

- `<script-name>` A script name to run: run_oa.py, collect.py, or analyze.py.

- `<project-dir>` The path to a Intel® Advisor Beta project directory.

- `<--options>` Options to modify behavior specific to the script. You can specify several options per script. Using an option not supported by the script causes a usage error.
A target application to analyze.

<table>
<thead>
<tr>
<th>Syntax Rules and Alternatives</th>
</tr>
</thead>
<tbody>
<tr>
<td>• An option can be preceded by one or two dashes. This section uses two dashes before long version of options and one dash before short version of options. For example, the following commands are equivalent:</td>
</tr>
<tr>
<td>```bash</td>
</tr>
<tr>
<td>advixe-python $APM/run_oa.py -h</td>
</tr>
<tr>
<td>advixe-python $APM/run_oa.py --help</td>
</tr>
<tr>
<td>• The path to a project directory must always follow after a script name. For example:</td>
</tr>
<tr>
<td>```bash</td>
</tr>
<tr>
<td>advixe-python $APM/analyze.py ./advi</td>
</tr>
<tr>
<td>• If an option accepts values, they can be separated by a space or by an equal sign (=). This document uses space for all such options. For example, the following are equivalent:</td>
</tr>
<tr>
<td>```bash</td>
</tr>
<tr>
<td>advixe-python $APM/analyze.py ./advi --out-dir ./report</td>
</tr>
<tr>
<td>advixe-python $APM/analyze.py ./advi --out-dir=./report</td>
</tr>
<tr>
<td>• The target executable must be preceded by two dashes and a space. For example:</td>
</tr>
<tr>
<td>```bash</td>
</tr>
<tr>
<td>advixe-python $APM/run_oa.py ./advi -- myApplication</td>
</tr>
<tr>
<td>advixe-python $APM/collect.py ./advi -- myApplication</td>
</tr>
<tr>
<td>• If you have Python 3.6 or 3.7 installed and it is the default Python version on your machine, you can run Offload Advisor with your system Python instead of the advixe-python tool:</td>
</tr>
<tr>
<td>```bash</td>
</tr>
<tr>
<td>python $APM/run_oa.py ./advi -- myApplication</td>
</tr>
<tr>
<td>python3.6 $APM/run_oa.py ./advi -- myApplication</td>
</tr>
<tr>
<td>python3.7 $APM/run_oa.py ./advi -- myApplication</td>
</tr>
<tr>
<td>run_oa.py Options</td>
</tr>
<tr>
<td>Collect basic data, do markup, and collect refinement data. Then proceed to run analysis on profiling data. This script combines the separate scripts collect.py and analyze.py.</td>
</tr>
</tbody>
</table>

Usage
advixe-python <APM>/run_oa.py <project-dir> [--options] -- <target> [target-options]

NOTE Replace `<APM>` with $APM on Linux* OS or %APM% on Windows* OS.
## Options

The following table describes options that you can use with the `run_oa.py` script. The target application to analyze and application options, if any, must be preceded by two dashes and a space and placed at the end of a command.

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>&lt;project-dir&gt;</code></td>
<td>Required. Specify the path to the Intel® Advisor Beta project directory.</td>
</tr>
<tr>
<td>-h</td>
<td>Show help message and exit.</td>
</tr>
<tr>
<td>--help</td>
<td></td>
</tr>
<tr>
<td>-v <code>&lt;verbose&gt;</code></td>
<td>Specify output verbosity level:</td>
</tr>
<tr>
<td>--verbose <code>&lt;verbose&gt;</code></td>
<td>• 1 - Show only error messages. This is the least verbose level. &lt;br&gt;• 2 - Show warning and error messages. &lt;br&gt;• 3 (default) - Show information, warning, and error messages. &lt;br&gt;• 4 - Show debug, information, warning, and error messages. This is the most verbose level.</td>
</tr>
<tr>
<td>--no-cachesim</td>
<td>Disable cache simulation during collection. The model assumes 100% hit rate for cache.</td>
</tr>
<tr>
<td>--config <code>&lt;config&gt;</code></td>
<td>Specify a configuration file by absolute path or name. If you choose the latter, the model configuration directory is searched for the file first, then the current directory. You can specify several configurations by using the option more than once.</td>
</tr>
<tr>
<td>-o <code>&lt;output-dir&gt;</code></td>
<td>Specify the directory to put all generated files into. By default, results are saved in <code>&lt;advisor-project&gt;/perf_models/mNNNN</code>. If you specify an existing directory or absolute path, results are saved in this directory. The new directory is created if it does not exist. If you only specify the directory <code>&lt;name&gt;</code>, results are stored in <code>&lt;advisor-project&gt;/perf_models/&lt;name&gt;</code>.</td>
</tr>
<tr>
<td>--out-dir <code>&lt;output-dir&gt;</code></td>
<td></td>
</tr>
<tr>
<td>-p <code>&lt;output-name-prefix&gt;</code></td>
<td>Specify a string to be prepended to output result filenames.</td>
</tr>
<tr>
<td>--out-name-prefix <code>&lt;output-name-prefix&gt;</code></td>
<td></td>
</tr>
<tr>
<td>Option</td>
<td>Description</td>
</tr>
<tr>
<td>--------</td>
<td>-------------</td>
</tr>
<tr>
<td><code>--no-stacks</code></td>
<td>Run data collection without collecting data distribution over stacks. You can use this option to reduce overhead at the potential expense of accuracy.</td>
</tr>
<tr>
<td><code>-c {basic, refinement, full}</code></td>
<td>Specify the type of data to collect for the application:</td>
</tr>
</tbody>
</table>
| `--collect {basic, refinement, full}` | • basic - Collect basic data (Hotspots and FLOPs).  
• refinement - Collect refined data (Dependencies) for marked loops only.  
• full (default) - Collect both basic data for application and refined data for marked loops. |
<p>| <code>-b</code> | | [Deprecated; use <code>--collect basic</code> instead.] |
| <code>--collect-basic</code> | | [Deprecated; use <code>--collect refinement</code> instead.] |
| <code>-r</code> | | [Deprecated; use <code>--collect full</code> instead.] |
| <code>--collect-refinement</code> | | [Deprecated; use <code>--collect refinement</code> instead.] |
| <code>-f</code> | | [Deprecated; use <code>--collect full</code> instead.] |
| <code>--collect-full</code> | | [Deprecated; use <code>--collect full</code> instead.] |
| <code>--executable-of-interest &lt;executable-name&gt;</code> | Specify an executable process name to profile if it is not the same as the application to run. Use this option if you run your application via script or other binary. |
| <code>--model-system-calls (default)</code> | Analyze regions with system calls inside. The actual presence of system calls inside a region may reduce model accuracy. |
| <code>--no-model-system-calls</code> | Do not analyze regions that contain system calls. |
| <code>--jit</code> | Enable data collection and analysis for applications with DPC++, OpenMP* target, and OpenCL™ code on a base platform. |
| <code>--no-profile-jit</code> | Disable JIT functions analysis. |
| <code>--enable-slm</code> | Enable SLM modeling in the memory hierarchy model. Must be used both with <code>collect.py</code> and <code>analyze.py</code>. |
| <code>-m [{all, generic, regions, omp, dpcpp, daal, tbb}]</code> | Mark up loops after survey or other data collection. Use this option to limit the scope of further collections by selecting loops according to a provided parameter: |
| <code>--markup [{all, generic, regions, omp, dpcpp, daal, tbb}]</code> | | [NOTE] Specify the <code>name</code> only, not the full path. |</p>
<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>• <strong>all</strong> - Get lists of loop IDs to pass as the option for further collections.</td>
</tr>
<tr>
<td></td>
<td>• <strong>generic</strong> <em>(default)</em> - Mark up all regions and select the most profitable ones.</td>
</tr>
<tr>
<td></td>
<td>• <strong>regions</strong> - Select already existing parallel regions.</td>
</tr>
<tr>
<td></td>
<td>• <strong>omp</strong> - Select outermost loops in OpenMP* regions.</td>
</tr>
<tr>
<td></td>
<td>• <strong>dpcpp</strong> - Select outermost loops in DPC++ regions.</td>
</tr>
<tr>
<td></td>
<td>• <strong>daal</strong> - Select outermost loops in Intel® Data Analytics Acceleration Library (Intel® DAAL) regions.</td>
</tr>
<tr>
<td></td>
<td>• <strong>tbb</strong> - Select outermost loops in Intel® Threading Building Blocks (Intel® TBB) regions.</td>
</tr>
</tbody>
</table>

**NOTE**

This option increases collection overhead.

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>--no-cache-sources</td>
<td>Enable keeping source code cache within a project.</td>
</tr>
<tr>
<td>--dry-run</td>
<td>Show the Intel® Advisor Beta CLI commands for advixe-cl appropriate for the specified configuration. No actual collection is performed.</td>
</tr>
<tr>
<td>--data-transfer <em>(default)</em></td>
<td>Enable data transfer analysis.</td>
</tr>
<tr>
<td></td>
<td><strong>NOTE</strong> This option increases collection overhead.</td>
</tr>
<tr>
<td>--no-data-transfer</td>
<td>Disable data transfer analysis. Use this option to reduce collection overhead.</td>
</tr>
<tr>
<td>--track-heap-objects <em>(default)</em></td>
<td>Attribute heap-allocated objects to the analyzed loops that accessed these objects. Enabling can increase collection overhead.</td>
</tr>
<tr>
<td>--no-track-heap-objects</td>
<td>Disable attributing heap-allocated objects to the analyzed loops that accessed the objects. Disabling can decrease collection overhead.</td>
</tr>
<tr>
<td>--track-stack-accesses <em>(default)</em></td>
<td>Track accesses to stack memory.</td>
</tr>
<tr>
<td>--no-track-stack-accesses</td>
<td>Disable tracking accesses to stack memory.</td>
</tr>
</tbody>
</table>
Examples

- Collect full data on myApplication, run analysis with default configuration, and save the project to the .advi directory. The generated output is saved to the default advi/perfmodels/mNNNN directory.

  advixe-python $APM/run_oa.py ./advi -- myApplication

- Collect full data on myApplication, run analysis with default configuration, save the project to the .advi directory, and save the generated output to the advi/perf_models/report directory.

  advixe-python $APM/run_oa.py ./advi --out-dir report -- myApplication

- Collect refinement data for DPC++ code regions on myApplication, run analysis with a custom configuration file config.toml, and save the project to the .advi directory. The generated output is saved to the default advi/perf_models/mNNNN directory.

  advixe-python $APM/run_oa.py ./advi --collect refinement --markup dpcpp --config ./config.toml -- myApplication

See Also

- Unified Data Collection and Output Generation

collect.py Options

Depending on options specified, collect basic data, do markup, and collect refinement data. By default, execute all steps. For any step besides markup, you must specify an application argument.

Usage

advixe-python <APM>/collect.py <project-dir> [--options] -- <target> [target-options]

NOTE Replace <APM> with $APM on Linux* OS or %APM% on Windows* OS.

Options

The following table describes options that you can use with the collect.py script. The target application to analyze and application options, if any, must be preceded by two dashes and a space.

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;project-dir&gt;</td>
<td>Required. Specify the path to the Intel® Advisor Beta project directory.</td>
</tr>
<tr>
<td>-h</td>
<td>Show help message and exit.</td>
</tr>
<tr>
<td>--help</td>
<td></td>
</tr>
<tr>
<td>-v &lt;verbose&gt;</td>
<td>Specify output verbosity level:</td>
</tr>
<tr>
<td>--verbose &lt;verbose&gt;</td>
<td>• 1 - Show only error messages. This is the least verbose level.</td>
</tr>
<tr>
<td></td>
<td>• 2 - Show warning and error messages.</td>
</tr>
<tr>
<td></td>
<td>• 3 (default) - Show information, warning, and error messages.</td>
</tr>
<tr>
<td></td>
<td>• 4 - Show debug, information, warning, and error messages. This is the most verbose level.</td>
</tr>
<tr>
<td>Option</td>
<td>Description</td>
</tr>
<tr>
<td>----------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>--no-cachesim</td>
<td>Disable cache simulation during collection. The model assumes 100% hit rate for cache.</td>
</tr>
<tr>
<td>--config &lt;config&gt;</td>
<td>Specify a configuration file by absolute path or name. If you choose the latter, the model configuration directory is searched for the file first, then the current directory. You can specify several configurations by using the option more than once.</td>
</tr>
<tr>
<td>-o &lt;output-dir&gt;</td>
<td>Specify the directory to put all generated files into. By default, results are saved in &lt;advisor-project&gt;/perf_models/mNNNN. If you specify an existing directory or absolute path, results are saved in this directory. The new directory is created if it does not exist. If you only specify the directory &lt;name&gt;, results are stored in &lt;advisor-project&gt;/perf_models/&lt;name&gt;.</td>
</tr>
<tr>
<td>-p &lt;output-name-prefix&gt;</td>
<td>Specify a string to be prepended to output result filenames.</td>
</tr>
</tbody>
</table>
| -c {basic, refinement, full} | Specify the type of data to collect for an application:  
  - basic - Collect basic data (Hotspots and FLOPs).  
  - refinement - Collect refined data (Dependencies) for marked loops only.  
  - full (default) - Collect both basic data for application and refined data for marked loops. |
<p>| -b                   | Deprecation: use --collect basic instead. |
| --collect-basic      | Deprecation: use --collect refinement instead. |
| -r                   | Deprecation: use --collect full instead. |</p>
<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
</table>
| `--executable-of-interest <executable-name>` | Specify the executable process name to profile if it is not the same as the application to run. Use this option if you run your application via script or other binary.  

**NOTE**  
Specify the name only, not the full path. |
| `--model-system-calls (default)` | Analyze regions with system calls inside. The actual presence of system calls inside a region may reduce model accuracy. |
| `--no-model-system-calls` | Do not analyze regions that contain system calls. |
| `--jit` | Enable data collection and analysis for applications with DPC++, OpenMP* target, and OpenCL™ code on a base platform. |
| `--enable-slm` | Enable SLM modeling in the memory hierarchy model. Must be used both with `collect.py` and `analyze.py` |
| `--no-profile-jit` | Disable analysis of JIT functions. |
| `-m [{all, generic, regions, omp, dpcpp, daal, tbb}]` | Mark up loops after survey or other data collection. Use this option to limit the scope of further collections by selecting loops according to a provided parameter:  
- **all** - Get lists of loop IDs to pass as the option for further collections.  
- **generic** *(default)* - Mark up all regions and select the most profitable ones.  
- ** regions** - Select already existing parallel regions.  
- **omp** - Select outermost loops in OpenMP* regions.  
- **dpcpp** - Select outermost loops in DPC++ regions.  
- **daal** - Select outermost loops in Intel® Data Analytics Acceleration Library (Intel® DAAL) regions.  
- ** tbb** - Select outermost loops in Intel® Threading Building Blocks (Intel® TBB) regions.  

`omp, dpcpp, or generic` selects loops in the project so that the corresponding collection can be run without loop selection options.  
You can specify several parameters in a comma-separated list. Loops are selected if they fit any of specified parameters. |
<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>--no-cache-sources</td>
<td>Enable keeping source code cache within a project.</td>
</tr>
<tr>
<td>--dry-run</td>
<td>Show the Intel® Advisor Beta CLI commands for advixe-cl appropriate for the specified configuration. No actual collection is performed.</td>
</tr>
<tr>
<td>--data-transfer (default)</td>
<td>Enable data transfer analysis.</td>
</tr>
<tr>
<td>--no-data-transfer</td>
<td>Disable data transfer analysis. Use this option to reduce collection overhead.</td>
</tr>
<tr>
<td>--track-heap-objects (default)</td>
<td>Attribute heap-allocated objects to the analyzed loops that accessed these objects. Enabling can increase collection overhead.</td>
</tr>
<tr>
<td>--no-track-heap-objects</td>
<td>Disable attributing heap-allocated objects to the analyzed loops that accessed the objects. Disabling can decrease collection overhead.</td>
</tr>
<tr>
<td>--no-stacks</td>
<td>Run data collection without collecting data distribution over stacks. You can use this option to reduce overhead at the potential expense of accuracy.</td>
</tr>
<tr>
<td>--track-stack-accesses (default)</td>
<td>Track accesses to stack memory.</td>
</tr>
<tr>
<td>--no-track-stack-accesses</td>
<td>Disable tracking accesses to stack memory.</td>
</tr>
</tbody>
</table>

**Examples**

- Collect full data on myApplication with default configuration and save the project to the ./advi directory.
  
  advixe-python $APM/collect.py ./advi -- myApplication

- Collect refinement data for OpenMP* and DPC++ loops on myApplication with a custom configuration file config.toml and save the project to the ./advi directory.
  
  advixe-python $APM/collect.py ./advi --collect refinement --markup [omp,dpcpp] --config ./config.toml -- myApplication

- Get commands appropriate for a custom configuration specified in the config.toml file to collect data separately with advixe-cl. The commands are ready to copy and paste.
  
  advixe-python $APM/collect.py ./advi --dry-run --config ./config.toml

**See Also**

- Separate Data Collection Using collect.py

**analyze.py Options**

This script allows you to run an analysis on profiling data and generate report results.

**Usage**

advixe-python <APM>/analyze.py <project-dir> [--options]
Options
The following table describes options that you can use with the `analyze.py` script.

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>&lt;project-dir&gt;</code></td>
<td>Required. Specify the path to the Intel® Advisor Beta project directory.</td>
</tr>
<tr>
<td>-h --help</td>
<td>Show help message and exit.</td>
</tr>
<tr>
<td>--version</td>
<td>Display Intel® Advisor Beta version information.</td>
</tr>
<tr>
<td>-v <code>&lt;verbose&gt;</code></td>
<td>Specify output verbosity level:</td>
</tr>
<tr>
<td>--verbose <code>&lt;verbose&gt;</code></td>
<td>• 1 - Show only error messages. This is the least verbose level.</td>
</tr>
<tr>
<td></td>
<td>• 2 - Show warning and error messages.</td>
</tr>
<tr>
<td></td>
<td>• 3 (default) - Show information, warning, and error messages.</td>
</tr>
<tr>
<td></td>
<td>• 4 - Show debug, information, warning, and error messages. This is the most verbose level.</td>
</tr>
<tr>
<td>--no-cachesim</td>
<td>Disable cache simulation during collection. The model assumes 100% hit rate for cache.</td>
</tr>
<tr>
<td></td>
<td>Usage decreases analysis overhead.</td>
</tr>
<tr>
<td>--config <code>&lt;config&gt;</code></td>
<td>Specify a configuration file by absolute path or name. If you choose the latter, the model configuration directory is searched for the file first, then the current directory. You can specify several configurations by using the option more than once.</td>
</tr>
<tr>
<td>-o <code>&lt;output-dir&gt;</code></td>
<td>Specify the directory to put all generated files into. By default, results are saved in <code>&lt;advisor-project&gt;/perf_models/mNNNN</code>. If you specify an existing directory or absolute path, results are saved in this directory. The new directory is created if it does not exist. If you only specify the directory <code>&lt;name&gt;</code>, results are stored in <code>&lt;advisor-project&gt;/perf_models/&lt;name&gt;</code>.</td>
</tr>
<tr>
<td>Option</td>
<td>Description</td>
</tr>
<tr>
<td>-------------------------------------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>-p &lt;output-name-prefix&gt;</td>
<td>Specify a string to be prepended to output result filenames.</td>
</tr>
<tr>
<td>--out-name-prefix &lt;output-name-prefix&gt;</td>
<td></td>
</tr>
<tr>
<td>--assume-parallel</td>
<td>Assume that a loop is parallel if the loop type is not known.</td>
</tr>
<tr>
<td>--no-assume-parallel (default)</td>
<td>Assume that a loop has a dependency if the loop type is not known.</td>
</tr>
<tr>
<td>--set-parallel [&lt;IDs/source-locations&gt;]</td>
<td>Assume loops are parallel if they have IDs or source locations from a specified comma-separated list. If the list is empty, assume all loops are parallel.</td>
</tr>
<tr>
<td><strong>NOTE</strong></td>
<td>--set-dependency option takes precedence over --set-parallel, so if a loop is listed in both, it is considered as having a dependency.</td>
</tr>
<tr>
<td>--set-dependency [&lt;IDs/source-locations&gt;]</td>
<td>Assume loops have dependencies if they have IDs or source locations from the specified comma-separated list. If the list is empty, assume all loops have dependencies.</td>
</tr>
<tr>
<td><strong>NOTE</strong></td>
<td>--set-dependency option takes precedence over --set-parallel, so if a loop is listed in both, it is considered as having a dependency.</td>
</tr>
<tr>
<td>--non-accel-time-breakdown</td>
<td>Provide a detailed breakdown of non-offloaded parts of offloaded regions.</td>
</tr>
<tr>
<td>-l [&lt;file-name&gt;:&lt;line-number&gt;]</td>
<td>Limit the analysis to specified loop nests determined by passing a topmost loop. The parameter must be a comma-separated list of source locations in the following format: &lt;file-name&gt;:&lt;line-number&gt;.</td>
</tr>
<tr>
<td>--select-loops [&lt;file-name&gt;:&lt;line-number&gt;]</td>
<td></td>
</tr>
<tr>
<td>--loop-filter-threshold &lt;threshold&gt;</td>
<td>Specify the loop filter threshold in seconds. The default is 0.02. Loop nests with total time less than the threshold are ignored.</td>
</tr>
<tr>
<td>--small-node-filter &lt;threshold&gt;</td>
<td>Specify the total time threshold, in seconds, to filter out nodes in the program_tree.dot and program_tree.pdf that fall below this value. The default is 0.0.</td>
</tr>
<tr>
<td>--evaluate-min-speedup</td>
<td>Enable offload fraction estimation that reaches minimum speedup defined in a configuration file. Disabled by default.</td>
</tr>
<tr>
<td>Option</td>
<td>Description</td>
</tr>
<tr>
<td>--------</td>
<td>-------------</td>
</tr>
<tr>
<td><code>--mpi-rank &lt;mpi-rank&gt;</code></td>
<td>Use results for the specified MPI rank if multiple ranks were analyzed.</td>
</tr>
<tr>
<td><code>--model-children (default)</code></td>
<td>Analyze child loops of the region head to find if some of the loops provide more profitable offload.</td>
</tr>
<tr>
<td><code>--no-model-children</code></td>
<td>Do not analyze child loops of the region head.</td>
</tr>
<tr>
<td><code>--check-profitability (default)</code></td>
<td>Check the profitability of offloading regions. Only regions that can benefit from the increased speed are added to a report.</td>
</tr>
<tr>
<td><code>--no-check-profitability</code></td>
<td>Add all evaluated regions to a report, regardless of the profitability of offloading specific regions.</td>
</tr>
<tr>
<td><code>--use-collect-configs</code></td>
<td>Use configuration files from collection phase in addition to default and custom configuration files.</td>
</tr>
<tr>
<td><code>--no-use-collect-configs (default)</code></td>
<td>Do not use configuration files from collection phase. Use only default and custom configuration files.</td>
</tr>
<tr>
<td><code>--model-system-calls (default)</code></td>
<td>Analyze regions with system calls inside. The actual presence of system calls inside a region may reduce model accuracy.</td>
</tr>
<tr>
<td><code>--no-model-system-calls</code></td>
<td>Do not analyze regions that contain system calls.</td>
</tr>
<tr>
<td><code>--jit</code></td>
<td>Enable data collection and analysis for applications with DPC++, OpenMP* target, and OpenCL™ code on a base platform.</td>
</tr>
<tr>
<td><code>--enable-slm</code></td>
<td>Enable SLM modeling in the memory hierarchy model. Must be used both with <code>collect.py</code> and <code>analyze.py</code>.</td>
</tr>
<tr>
<td><code>--track-heap-objects (default)</code></td>
<td>Attribute heap-allocated objects to the analyzed loops that accessed these objects. Enabling can increase collection overhead.</td>
</tr>
<tr>
<td><code>--no-track-heap-objects</code></td>
<td>Disable attributing heap-allocated objects to the analyzed loops that accessed the objects. Disabling can decrease collection overhead.</td>
</tr>
<tr>
<td><code>--model-extended-math (default)</code></td>
<td>Model calls to math functions such as EXP, LOG, SIN, and COS as extended math instructions, if possible.</td>
</tr>
<tr>
<td><code>--no-model-extended-math</code></td>
<td>Disable modeling calls to math functions such as EXP, LOG, SIN, and COS as extended math instructions.</td>
</tr>
<tr>
<td><code>--search-n-dim (default)</code></td>
<td>Enable search for optimal N-dimensional offload.</td>
</tr>
<tr>
<td><code>--no-search-n-dim</code></td>
<td>Disable search for optimal N-dimensional offload.</td>
</tr>
<tr>
<td><code>--force-32bit-arithmetics</code></td>
<td>Force all arithmetic operations to be considered single-precision FPs or int32.</td>
</tr>
<tr>
<td>Option</td>
<td>Description</td>
</tr>
<tr>
<td>--------------------------------------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>--force-64bit-arithmetics</td>
<td>Force all arithmetic operations to be considered double-precision FPs or int64.</td>
</tr>
<tr>
<td>--enable-batching</td>
<td>Enable job batching for top-level offloads. Emulate the execution of more than one instance simultaneously. This option is equivalent to --threads=total-EU-count*threads-per-EU.</td>
</tr>
<tr>
<td>--disable-batching (default)</td>
<td>Disable job batching for top-level offloads.</td>
</tr>
<tr>
<td>--threads &lt;number-of-threads&gt;</td>
<td>Specify the number of parallel threads to use for offload heads.</td>
</tr>
<tr>
<td>-e</td>
<td>Skip the profitability check, disable analyzing child loops and functions, and ensure that the rows marked for offload are offloaded even if offloading child rows is more profitable.</td>
</tr>
<tr>
<td>--enforce-offloads</td>
<td>Enable the profitability check and analyzing child loops and functions to find the most profitable offload entries.</td>
</tr>
<tr>
<td>--no-enforce-offloads (default)</td>
<td>Enable the profitability check and analyzing child loops and functions to find the most profitable offload entries.</td>
</tr>
<tr>
<td>--count-memory-instructions (default)</td>
<td>Use the projection of x86 instructions with memory to GPU SEND/SENDs instructions.</td>
</tr>
<tr>
<td>--no-count-memory-instructions</td>
<td>Do not use the projection of x86 instructions with memory to GPU SEND/SENDs instructions.</td>
</tr>
<tr>
<td>--assume-ndim-dependency (default)</td>
<td>When searching for an optimal N-dimensional offload, assume there are dependencies between inner and outer loops.</td>
</tr>
<tr>
<td>--no-assume-ndim-dependency</td>
<td>When searching for an optimal N-dimensional offload, assume there are no dependencies between inner and outer loops.</td>
</tr>
<tr>
<td>-m &lt;markup&gt;</td>
<td>Select markup_analyze, affecting which regions to mark up for data collection and analysis.</td>
</tr>
<tr>
<td>--markup &lt;markup&gt;</td>
<td>Select markup_analyze, affecting which regions to mark up for data collection and analysis.</td>
</tr>
<tr>
<td>--count-mov-instructions</td>
<td>Use the projection of x86 MOV instructions to GPU MOV instructions.</td>
</tr>
<tr>
<td>--no-count-mov-instructions (default)</td>
<td>Do not use the projection of x86 MOV instructions to GPU MOV instructions.</td>
</tr>
<tr>
<td>--disable-fp64-math-optimization</td>
<td>Disable accounting for optimized traffic for transcendentals on the GPU.</td>
</tr>
<tr>
<td>--no-stacks</td>
<td>Run data analysis without using callstacks data. You can use this option to avoid bad callstacks attributed data at the expense of accuracy.</td>
</tr>
<tr>
<td>--data-transfer-histogram (default)</td>
<td>Estimate fine-grained data transfer and latencies for each object transferred and add a memory object histogram to a report.</td>
</tr>
<tr>
<td>Option</td>
<td>Description</td>
</tr>
<tr>
<td>---------------------------------------------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td><strong>--no-data-transfer-histogram</strong></td>
<td>Disable fine-grained data transfer.</td>
</tr>
<tr>
<td><strong>--assume-hide-taxes</strong></td>
<td>Use an optimistic approach to estimate invocation taxes: hide all invocation taxes except the first one.</td>
</tr>
<tr>
<td><strong>--assume-never-hide-taxes (default)</strong></td>
<td>Use a pessimistic approach to estimate invocation taxes: do not hide invocation taxes.</td>
</tr>
<tr>
<td><strong>--assume-single-data-transfer (default)</strong></td>
<td>Assumed data is transferred once for each offload, and all instances share the data.</td>
</tr>
<tr>
<td><strong>--no-assume-single-data-transfer</strong></td>
<td>Assume each data object is transferred for every instance of an offload that uses it. This method assumes no data re-use between calls to the same kernel.</td>
</tr>
</tbody>
</table>

**Important** This option requires `track-heap-objects` and `track-stack-accesses` to be enabled during collection.

### Examples

- Run analysis with default configuration on the project in the `./advi` directory. The generated output is saved to the default `advi/perf_models/mNNNN` directory.

  ```
  advixe-python $APM/analyze.py ./advi
  ```

- Run analysis using the Intel® Processor Graphics Gen9 configuration for the specific loops of the `./advi` project. Add both analyzed loops to the report regardless of their offloading profitability. The generated output is saved to the default `advi/perf_models/mNNNN` directory.

  ```
  advixe-python $APM/analyze.py ./advi --config gen9 --select-loops [foo.cpp:34,bar.cpp:192] --no-check-profitability
  ```

- Run analysis for a custom configuration on the `./advi` project. Mark up regions for analysis and assume a code region is parallel if its type is unknown. Save the generated output to the `advi/perf_models/report` directory.

  ```
  advixe-python $APM/analyze.py ./advi --config ./myConfig.toml --markup --assume-parallel --outdir report
  ```

**See Also**

- Run Performance Modeling

**Reported Metrics Reference**

Enabled Metrics | Hidden Metrics

The **Offloaded Regions**, **Non-Offloaded Regions**, and **Call Tree** sections report performance metrics and important information about certain code regions. The metric tables are broken down into several column groups, which are collapsed by default and show only critical metrics for a section. Double-click a title to expand a group and see additional columns.
To customize a metric table or a column, use the following controls:

- Use the tabs in the right sidebar of the metrics table:
  - Click Column configurator tab and select columns to show or hide from the report. Some columns are hidden by default, you can enable them from this tab. Select a column group to show/hide all columns in it.
  - Click Custom filter tab and expand a column title to filter rows reported by custom value criteria.
  - Hover over a column header and click the menu button to open filters pane with the following tabs:
    - Size tab - Autosize a column(s) or column group(s).
    - Filter tab - Filter rows by column value criteria or by selecting/deselecting code regions from the Hierarchy column. This is the same as Custom filter, but for individual columns only.
    - Menu tab - Show or hide certain columns from the report. Some columns are hidden by default, you can enable them from this tab. This is the same as Column configurator.

- Right-click a table cell to open the context menu, from which you can expand a loop nest, adjust a column view, locate a loop in the call tree, copy the value from the focused cell, or export the whole metrics table as a .csv, .xlsx, or .xml file.

**Enabled Metrics**
Metrics enabled by default give a general performance overview and are helpful when investigating reasons why certain regions are profitable for offload and other are not offloaded. These metrics can be visible or collapsed. Double-click a column group title to expand it and see additional columns.

**Loop/Function**
The Loop/Function section reports basic information about loop and function execution and hierarchy.

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Description</th>
<th>Reported in</th>
</tr>
</thead>
<tbody>
<tr>
<td>Elapsed Time (s)</td>
<td>Elapsed time, in seconds, for the offload head of a code region.</td>
<td>Offloaded Regions</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Non-Offloaded Regions</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Call Tree</td>
</tr>
<tr>
<td>Total Time</td>
<td>Time in the loop nest hierarchy on the platform where the application binary is profiled. The time is reported in seconds and in percentage of total time.</td>
<td>Offloaded Regions</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Non-Offloaded Regions</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Call Tree</td>
</tr>
<tr>
<td>Dependency Type</td>
<td>Type of loop dependency: proved (such as Dependency: raw or Dependency: waw), assumed, or parallel.</td>
<td>Offloaded Regions</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Non-Offloaded Regions</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Call Tree</td>
</tr>
<tr>
<td>Column Name</td>
<td>Description</td>
<td>Reported in</td>
</tr>
<tr>
<td>-----------------------------</td>
<td>------------------------------------------------------------------------------</td>
<td>----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>Execution Target</td>
<td>Target execution platform. For offloaded regions, execution target is <strong>ACC</strong> (a target device).</td>
<td>Offloaded Regions (hidden and collapsed)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Non-Offloaded Regions (collapsed)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Call Tree</td>
</tr>
<tr>
<td>Index</td>
<td>Collapsed. Unique ID assigned to loops and functions by Intel® Advisor Beta.</td>
<td>Offloaded Regions</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Non-Offloaded Regions</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Call Tree</td>
</tr>
<tr>
<td>CLI Loop ID</td>
<td>Collapsed. Unique loop ID used for filtering purposes in Intel® Advisor Beta.</td>
<td>Offloaded Regions</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Non-Offloaded Regions</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Call Tree</td>
</tr>
<tr>
<td>Source Location</td>
<td>Collapsed. Place of a loop or function in a source files in the format <code>&lt;file name&gt;:&lt;line number&gt;</code>.</td>
<td>Offloaded Regions</td>
</tr>
<tr>
<td></td>
<td>You can use the source location in CLI commands.</td>
<td>Non-Offloaded Regions</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Call Tree</td>
</tr>
<tr>
<td>Offload Index</td>
<td>Collapsed. Unique offload region ID assigned by Intel® Advisor Beta.</td>
<td>Offloaded Regions</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Call Tree</td>
</tr>
<tr>
<td>Top Node in Offload</td>
<td>Collapsed. Location of the offload head.</td>
<td>Offloaded Regions</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Non-Offloaded Regions</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Call Tree</td>
</tr>
<tr>
<td>Location</td>
<td>Collapsed. Location of a loop or function in the source file.</td>
<td>Offloaded Regions</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Non-Offloaded Regions</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Call Tree</td>
</tr>
<tr>
<td>Node Position in Offload</td>
<td>Collapsed. Position of a node: offload head, child loop, or child function.</td>
<td>Offloaded Regions (hidden and collapsed)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Non-Offloaded Regions (hidden and collapsed)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Call Tree</td>
</tr>
</tbody>
</table>

**Offload Information**

The **Offload Information** reports performance details about code regions offloaded to a target device.

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Description</th>
<th>Reported in</th>
</tr>
</thead>
<tbody>
<tr>
<td>Why Not Offloaded?</td>
<td>Reason why offloading a code region was unprofitable or impossible. For details and</td>
<td>Non-Offloaded Regions</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Call Tree</td>
</tr>
<tr>
<td>Column Name</td>
<td>Description</td>
<td>Reported in</td>
</tr>
<tr>
<td>------------------------------------------</td>
<td>-----------------------------------------------------------------------------</td>
<td>-------------------------------------------------</td>
</tr>
<tr>
<td>Possible solutions, refer to Troubleshooting Why Not Offloaded.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Estimated Speedup</td>
<td>Estimated speedup after offloading to a target device in comparison to the original elapsed time.</td>
<td>Offloaded Regions</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Call Tree</td>
</tr>
<tr>
<td>Estimated Time on Accelerator</td>
<td>Total estimated time spent on a target device. The time is reported in seconds and in percentage of total offload plus non-offload time.</td>
<td>Offloaded Regions</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Call Tree</td>
</tr>
<tr>
<td>Total Execution Time by Compute</td>
<td>Execution time, in seconds, assuming the workload is bound only by compute throughput.</td>
<td>Offloaded Regions</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Non-Offloaded Regions (hidden and collapsed)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Call Tree</td>
</tr>
<tr>
<td>Total Execution Time by Memory BW Time (s)</td>
<td>Execution time, in seconds, assuming the workload is bound only by memory bandwidth time.</td>
<td>Offloaded Regions</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Non-Offloaded Regions (hidden and collapsed)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Call Tree</td>
</tr>
<tr>
<td>Total Execution Time by LLC BW (s)</td>
<td>Execution time, in seconds, assuming the workload is bound only by LLC bandwidth.</td>
<td>Offloaded Regions</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Non-Offloaded Regions (hidden and collapsed)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Call Tree</td>
</tr>
<tr>
<td>Total Execution Time by L3 BW (s)</td>
<td>Execution time, in seconds, assuming the workload is bound only by L3 cache bandwidth.</td>
<td>Offloaded Regions</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Non-Offloaded Regions (hidden and collapsed)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Call Tree</td>
</tr>
<tr>
<td>Parallel Threads</td>
<td>Number of parallel threads in an offloaded code region.</td>
<td>Offloaded Regions</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Call Tree</td>
</tr>
<tr>
<td>Bounded by</td>
<td>Limitations that an offloaded region is bounded by.</td>
<td>Offloaded Regions</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Non-Offloaded Regions (hidden)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Call Tree</td>
</tr>
<tr>
<td>Max Speedup for This Node (without offload tax)</td>
<td>Collapsed. Maximum possible speedup of offloaded code regions without the cost of offloading them to a target device.</td>
<td>Offloaded Regions</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Non-Offloaded Regions</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Call Tree</td>
</tr>
</tbody>
</table>
### Column Name | Description | Reported in
--- | --- | ---
Estimated Execution Time on Accelerator (+ Host) (s) | Collapsed. Total estimated time, in seconds, spent on a target device and a base platform after offloading. | Non-Offloaded Regions Call Tree

Fraction Offloaded (%) | Collapsed. Percentage of code regions offloaded to a target device. | Non-Offloaded Regions Call Tree (hidden)

Is Offload Candidate? | Collapsed. Indicates if a code region was analyzed to check its profitability for offloading. | Non-Offloaded Regions Call Tree

Estimated Non-Accelerable Time (s) | Collapsed. Total estimated time, in seconds, spent on serial execution on the host. | Offloaded Regions Call Tree

Whole Loop or Function Fits on Accelerator? | Collapsed. Indicates if a whole loop/function fits on a target device or only some part(s) of it. | Call Tree

Total Time Spent in MPI Calls (s) | Collapsed. Total time, in seconds, spent in MPI calls. | Offloaded Regions Non-Offloaded Regions (hidden and collapsed) Call Tree

Global Size | Collapsed. Total number of work items in a kernel executed. | Offloaded Regions Call Tree

---

**Overhead**

The **Overhead** section reports costs of offloading to a target device.

### Column Name | Description | Reported in
--- | --- | ---
Offload Taxes | Cost of offloading a code region from host to a target device. | Offloaded Regions Non-Offloaded Regions Call Tree

Data Transfer Tax (s) | Cost of transferring data between host and target device, in seconds. | Offloaded Regions Non-Offloaded Regions Call Tree

Invocation Tax (s) | Cost of offloading region to a target device assuming the tax is paid each time a kernel is invoked, in seconds. Does not include data transfer costs. | Offloaded Regions Non-Offloaded Regions Call Tree

Configuration Tax (s) | Cost of offloading region to a target device assuming the tax is paid only for the first time a | Offloaded Regions Non-Offloaded Regions Call Tree
<table>
<thead>
<tr>
<th>Column Name</th>
<th>Description</th>
<th>Reported in</th>
</tr>
</thead>
<tbody>
<tr>
<td>Kernel Code Transfer Tax (s)</td>
<td>Collapsed. Cost of transferring kernel code to a target device, in seconds.</td>
<td>Offloaded Regions</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Non-Offloaded Regions</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Call Tree</td>
</tr>
</tbody>
</table>

**Data Transfer**

The **Data Transfer** section reports data transfers to and from a target device.

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Description</th>
<th>Reported in</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total Data Transferred (MB)</td>
<td>Sum of the total incoming traffic to an accelerator and the total outgoing traffic from a target device, for an offload region, in megabytes. It is calculated as (MappedTo + MappedFrom + 2*MappedToFrom).</td>
<td>Offloaded Regions</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Non-Offloaded Regions</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Call Tree</td>
</tr>
<tr>
<td>Total Data Transferred from CPU to GPU (MB)</td>
<td>Data transferred from a base platform to a target device, for an offload region, in megabytes.</td>
<td>Offloaded Regions</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Non-Offloaded Regions</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Call Tree</td>
</tr>
<tr>
<td>Total Data Transferred from GPU to CPU (MB)</td>
<td>Data transferred from a target device to a base platform, for an offload region, in megabytes.</td>
<td>Offloaded Regions</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Non-Offloaded Regions</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Call Tree</td>
</tr>
<tr>
<td>Memory Mapped To Device (MB)</td>
<td>Collapsed. Data transferred to shared memory, for a loop or function, in megabytes.</td>
<td>Offloaded Regions</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Non-Offloaded Regions</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Call Tree</td>
</tr>
<tr>
<td>Memory Mapped From Device (MB)</td>
<td>Collapsed. Data transferred from shared memory, for a loop or function, in megabytes.</td>
<td>Offloaded Regions</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Non-Offloaded Regions</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Call Tree</td>
</tr>
<tr>
<td>Memory Mapped ToFrom Device (MB)</td>
<td>Collapsed. Data transferred both to and from shared memory, for the loop or function, in megabytes.</td>
<td>Offloaded Regions</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Non-Offloaded Regions</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Call Tree</td>
</tr>
</tbody>
</table>

**Trip Counts**

The **Trip Counts** section reports the number of loop iterations and calls.

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Description</th>
<th>Reported in</th>
</tr>
</thead>
<tbody>
<tr>
<td>Average Trip Count</td>
<td>Number of times a loop iterates on average.</td>
<td>Offloaded Regions</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Non-Offloaded Regions (hidden)</td>
</tr>
<tr>
<td>Column Name</td>
<td>Description</td>
<td>Reported in</td>
</tr>
<tr>
<td>-----------------------------</td>
<td>------------------------------------------------------------------------------</td>
<td>----------------------</td>
</tr>
<tr>
<td>Call Count</td>
<td>Number of times a loop or function is called.</td>
<td>Offloaded Regions</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Non-Offloaded Regions</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Call Tree</td>
</tr>
<tr>
<td>Total Trip Count</td>
<td>Collapsed. Total number of times a loop iterates.</td>
<td>Offloaded Regions</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Non-Offloaded Regions</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Call Tree</td>
</tr>
</tbody>
</table>

**L3 Cache**

The **L3 Cache** section reports details on L3 cache utilization.

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Description</th>
<th>Reported in</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total L3 Traffic (GB)</td>
<td>Total data, in gigabytes, that accessed the L3 cache.</td>
<td>Offloaded Regions</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Non-Offloaded Regions</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Call Tree</td>
</tr>
<tr>
<td>Read L3 Traffic (Bytes)</td>
<td>Collapsed. Total number of bytes read from L3 cache.</td>
<td>Offloaded Regions</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Non-Offloaded Regions</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Call Tree</td>
</tr>
<tr>
<td>Write L3 Traffic (Bytes)</td>
<td>Collapsed. Total number of bytes written to the L3 cache.</td>
<td>Offloaded Regions</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Non-Offloaded Regions</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Call Tree</td>
</tr>
<tr>
<td>Read L3 Bandwidth (Bytes/ck)</td>
<td>Collapsed. Number of bytes read from L3 cache during one cycle of a target device.</td>
<td>Offloaded Regions</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Non-Offloaded Regions</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Call Tree</td>
</tr>
<tr>
<td>Write L3 Bandwidth (Bytes/ck)</td>
<td>Collapsed. Number of bytes written to L3 cache during one cycle of a target device.</td>
<td>Offloaded Regions</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Non-Offloaded Regions</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Call Tree</td>
</tr>
</tbody>
</table>

**LLC**

The **LLC** reports details on last level cache (LLC) utilization.

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Description</th>
<th>Reported in</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total LLC Access (GB)</td>
<td>Total data, in gigabytes, that accessed the last level cache (LLC).</td>
<td>Offloaded Regions</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Non-Offloaded Regions</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Call Tree</td>
</tr>
<tr>
<td>Read LLC Traffic (Bytes)</td>
<td>Collapsed. Total number of bytes read from the LLC.</td>
<td>Offloaded Regions</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Non-Offloaded Regions</td>
</tr>
</tbody>
</table>
## Memory

The **Memory** section reports details on memory utilization.

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Description</th>
<th>Reported in</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total Memory Traffic (GB)</td>
<td>Total data accessed from the memory, in gigabytes.</td>
<td>Offloaded Regions</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Non-Offloaded Regions</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Call Tree</td>
</tr>
<tr>
<td>Write Memory Traffic per tile (Bytes)</td>
<td>Collapsed. Total number of bytes written to memory per tile.</td>
<td>Offloaded Regions</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Non-Offloaded Regions</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Call Tree</td>
</tr>
<tr>
<td>Read Memory Traffic per tile (Bytes)</td>
<td>Collapsed. Total number of bytes read from memory per tile.</td>
<td>Offloaded Regions</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Non-Offloaded Regions</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Call Tree</td>
</tr>
<tr>
<td>Write Memory Bandwidth per tile (Bytes/clk)</td>
<td>Collapsed. Total number of bytes written to memory per tile during one cycle of a target device.</td>
<td>Offloaded Regions</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Non-Offloaded Regions</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Call Tree</td>
</tr>
<tr>
<td>Read Memory Bandwidth per tile (Bytes/clk)</td>
<td>Collapsed. Total number of bytes read from memory per tile during one cycle of a target device.</td>
<td>Offloaded Regions</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Non-Offloaded Regions</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Call Tree</td>
</tr>
</tbody>
</table>

## Instruction & Traffic Counts

The **Instruction & Traffic Counts** reports details about floating-point traffic and execution of specific instructions.
**NOTE**
This column group is hidden in the **Non-Offloaded Regions** tab. You need to manually enable it from the **Column Configurator**.

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Description</th>
<th>Reported in</th>
</tr>
</thead>
<tbody>
<tr>
<td>FPU Util (GFLOP/s)</td>
<td>Number of billions of floating-point operations (GFLOP) transferred per second.</td>
<td>Offloaded Regions</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Non-Offloaded Regions (hidden)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Call Tree</td>
</tr>
<tr>
<td>FLOP per Cycle</td>
<td>Number of floating-point operations (FLOP) transferred per one cycle of a target device.</td>
<td>Offloaded Regions</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Non-Offloaded Regions (hidden)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Call Tree</td>
</tr>
<tr>
<td>ABS / Iteration</td>
<td>Collapsed. Number of ABS instructions per iteration.</td>
<td>Offloaded Regions</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Non-Offloaded Regions (hidden and collapsed)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Call Tree</td>
</tr>
<tr>
<td>Add / Iteration</td>
<td>Collapsed. Number of Add instructions per iteration.</td>
<td>Offloaded Regions</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Non-Offloaded Regions (hidden and collapsed)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Call Tree</td>
</tr>
<tr>
<td>Div / Iteration</td>
<td>Collapsed. Number of Div instructions per iteration.</td>
<td>Offloaded Regions</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Non-Offloaded Regions (hidden and collapsed)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Call Tree</td>
</tr>
<tr>
<td>FMA / Iteration</td>
<td>Collapsed. Number of FMA instructions per iteration.</td>
<td>Offloaded Regions</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Non-Offloaded Regions (hidden and collapsed)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Call Tree</td>
</tr>
<tr>
<td>MAX / Iteration</td>
<td>Collapsed. Number of MAX instructions per iteration.</td>
<td>Offloaded Regions</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Non-Offloaded Regions (hidden and collapsed)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Call Tree</td>
</tr>
<tr>
<td>MIN / Iteration</td>
<td>Collapsed. Number of MIN instructions per iteration.</td>
<td>Offloaded Regions</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Non-Offloaded Regions (hidden and collapsed)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Call Tree</td>
</tr>
<tr>
<td>MUL / Iteration</td>
<td>Collapsed. Number of ABS instructions per iteration.</td>
<td>Offloaded Regions</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Non-Offloaded Regions (hidden and collapsed)</td>
</tr>
<tr>
<td>Column Name</td>
<td>Description</td>
<td>Reported in</td>
</tr>
<tr>
<td>----------------</td>
<td>----------------------------------------------------------</td>
<td>---------------------------------------</td>
</tr>
<tr>
<td>RECCP / Iteration</td>
<td>Collapsed. Number of RECCP instructions per iteration.</td>
<td>Offloaded Regions</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Non-Offloaded Regions (hidden and collapsed)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Call Tree</td>
</tr>
<tr>
<td>SAD / Iteration</td>
<td>Collapsed. Number of SAD instructions per iteration.</td>
<td>Offloaded Regions</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Non-Offloaded Regions (hidden and collapsed)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Call Tree</td>
</tr>
<tr>
<td>SCALE / Iteration</td>
<td>Collapsed. Number of SCALE instructions per iteration.</td>
<td>Offloaded Regions</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Non-Offloaded Regions (hidden and collapsed)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Call Tree</td>
</tr>
<tr>
<td>SIGN / Iteration</td>
<td>Collapsed. Number of SIGN instructions per iteration.</td>
<td>Offloaded Regions</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Non-Offloaded Regions (hidden and collapsed)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Call Tree</td>
</tr>
<tr>
<td>SQRT / Iteration</td>
<td>Collapsed. Number of SQRT instructions per iteration.</td>
<td>Offloaded Regions</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Non-Offloaded Regions (hidden and collapsed)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Call Tree</td>
</tr>
<tr>
<td>SUB / Iteration</td>
<td>Collapsed. Number of SUB instructions per iteration.</td>
<td>Offloaded Regions</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Non-Offloaded Regions (hidden and collapsed)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Call Tree</td>
</tr>
</tbody>
</table>

**Diagnostics**

The **Diagnostics** reports detailed compiler diagnostics.

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Description</th>
<th>Reported in</th>
</tr>
</thead>
<tbody>
<tr>
<td>Diagnostics</td>
<td>Compiler diagnostics messages about situations that can affect model accuracy. For details, refer to Troubleshooting Diagnostics.</td>
<td>Offloaded Regions</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Non-Offloaded Regions</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Call Tree</td>
</tr>
<tr>
<td>No Execution Count</td>
<td>Collapsed. Time spent in parts of an offload not modeled because there is no execution count.</td>
<td>Offloaded Regions</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Non-Offloaded Regions</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Call Tree</td>
</tr>
<tr>
<td>Column Name</td>
<td>Description</td>
<td>Reported in</td>
</tr>
<tr>
<td>--------------------------</td>
<td>-----------------------------------------------------------------------------</td>
<td>--------------------------------------------------</td>
</tr>
<tr>
<td>No Static Mixes</td>
<td>Collapsed. Time spent in parts of an offload not modeled because there are no static mixes.</td>
<td>Offloaded Regions</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Non-Offloaded Regions</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Call Tree</td>
</tr>
<tr>
<td>Not in Bottom-Up Table</td>
<td>Collapsed. Time spent in parts of an offload not modeled because they are not in a bottom-up table.</td>
<td>Offloaded Regions</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Non-Offloaded Regions</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Call Tree</td>
</tr>
<tr>
<td>System Module</td>
<td>Collapsed. Time spent in parts of an offload not modeled because they are system modules.</td>
<td>Offloaded Regions</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Non-Offloaded Regions</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Call Tree</td>
</tr>
<tr>
<td>Zero Execution Count</td>
<td>Collapsed. Time spent in parts of an offload not modeled because of zero execution count.</td>
<td>Offloaded Regions</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Non-Offloaded Regions</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Call Tree</td>
</tr>
</tbody>
</table>

**Hidden Metrics**

Hidden metrics can be useful for advanced performance analysis or for debug purposes. To see the hidden metrics:

1. Open the **Column Configurator** pane on the right.
2. Select required metrics from the list.

**NOTE** Some of these metrics require that you expand the column group they are reported in first.

**Loop/Function**

The **Loop/Function** section reports basic information about loop and function execution and hierarchy.

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Description</th>
<th>Reported in</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parent Index</td>
<td>Hidden. Parent's unique ID assigned by Intel® Advisor Beta.</td>
<td>Offloaded Regions (hidden and collapsed)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Non-Offloaded Regions (hidden and collapsed)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Call Tree</td>
</tr>
<tr>
<td>Total Time (%)</td>
<td>Hidden. Percentage of total application time in the loop nest hierarchy.</td>
<td>Offloaded Regions</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Non-Offloaded Regions</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Call Tree</td>
</tr>
<tr>
<td>Total Application Time (s)</td>
<td>Hidden. Total application time, in seconds.</td>
<td>Offloaded Regions</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Non-Offloaded Regions</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Call Tree</td>
</tr>
</tbody>
</table>

**Offload Information**

The **Offload Information** reports performance details about code regions offloaded to a target device.
<table>
<thead>
<tr>
<th>Column Name</th>
<th>Description</th>
<th>Reported in</th>
</tr>
</thead>
<tbody>
<tr>
<td>Whole Loop or Function Offloaded?</td>
<td>Indicates if a whole loop/function is offloaded on a target device or only some part(s) of it.</td>
<td>Offloaded Regions</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Call Tree</td>
</tr>
<tr>
<td>Estimated Time on Accelerator (%)</td>
<td>Hidden. Estimated time spent on a target device as a percentage of total offload plus non-offload time.</td>
<td>Offloaded Regions</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Call Tree</td>
</tr>
<tr>
<td>Topmost Node of Offload</td>
<td>Hidden. Location of the offload head.</td>
<td>Offloaded Regions</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Non-Offloaded Regions</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Call Tree</td>
</tr>
<tr>
<td>Head of Regions Offloaded Together</td>
<td>Hidden and collapsed. Head of several child loops/functions if they are offloaded to a target device together.</td>
<td>Call Tree</td>
</tr>
<tr>
<td>Node Position in Offload</td>
<td>Hidden and collapsed. Position of a node: child or head node.</td>
<td>Offloaded Regions</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Non-Offloaded Regions</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Call Tree</td>
</tr>
<tr>
<td>Global Size Model Error (%)</td>
<td>Hidden and collapsed. Percentage of work items executed in a kernel with a possible error of loop compute time caused by different approaches of converting single-dimension loops to multi-dimension.</td>
<td>Offloaded Regions</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Call Tree</td>
</tr>
</tbody>
</table>

**Potential Offload**

The **Potential Offload** reports details about *not offloaded* code regions. This section is hidden by default.

**NOTE**

This column group is available only in the **Non-Offloaded Regions** and **Call Tree** tabs.

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Description</th>
<th>Reported in</th>
</tr>
</thead>
<tbody>
<tr>
<td>Non-Offloaded Weight</td>
<td>Hidden and collapsed. Weight of a non-offloaded code region in the code tree.</td>
<td>Non-Offloaded Regions</td>
</tr>
</tbody>
</table>

**Data Transfer**

The **Data Transfer** section reports data transfers to and from a target device.
<table>
<thead>
<tr>
<th>Column Name</th>
<th>Description</th>
<th>Reported in</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data Transferred (Shared) (MB)</td>
<td>Hidden and collapsed. Sum of data transferred to a target device, from the target device, and in both directions (to and from the target device).</td>
<td>Offloaded Regions</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Non-Offloaded Regions</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Call Tree</td>
</tr>
<tr>
<td>Memory Objects</td>
<td>Hidden and Collapsed. Details about memory objects used in a region in the following format: &lt;object size in bytes&gt;/&lt;memory allocation type&gt;/&lt;allocation address&gt;/&lt;source location file and line number&gt;/&lt;transfer direction&gt;.</td>
<td>Offloaded Regions</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Non-Offloaded Regions</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Call Tree</td>
</tr>
</tbody>
</table>

**Instruction & Traffic Counts**

The **Instruction & Traffic Counts** reports details about floating-point traffic and execution of specific instructions.

**NOTE**

This column group is hidden in the **Non-Offloaded Regions** tab. You need to manually enable it from the **Column Configurator**.

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Description</th>
<th>Reported in</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bytes / Iteration</td>
<td>Hidden and collapsed. Average number of bytes transferred per iteration.</td>
<td>Offloaded Regions</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Non-Offloaded Regions</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Call Tree</td>
</tr>
<tr>
<td>Bytes Read / Iteration</td>
<td>Hidden and collapsed. Average number of bytes read per iteration.</td>
<td>Offloaded Regions</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Non-Offloaded Regions</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Call Tree</td>
</tr>
<tr>
<td>Bytes Written / Iteration</td>
<td>Hidden and collapsed. Average number of bytes written per iteration.</td>
<td>Offloaded Regions</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Non-Offloaded Regions</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Call Tree</td>
</tr>
<tr>
<td>Floating Operations / Iteration</td>
<td>Hidden and collapsed. Average number of floating-point operations per iteration.</td>
<td>Offloaded Regions</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Non-Offloaded Regions</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Call Tree</td>
</tr>
<tr>
<td>Compute</td>
<td>Hidden and collapsed. Number of compute operations.</td>
<td>Offloaded Regions</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Non-Offloaded Regions</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Call Tree</td>
</tr>
<tr>
<td>Compute with Memory</td>
<td>Hidden and collapsed. Number of compute with memory instructions.</td>
<td>Offloaded Regions</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Non-Offloaded Regions</td>
</tr>
<tr>
<td>Column Name</td>
<td>Description</td>
<td>Reported in</td>
</tr>
<tr>
<td>-----------------------------------</td>
<td>------------------------------------------------------------------------------</td>
<td>--------------</td>
</tr>
<tr>
<td>Vector Compute</td>
<td>Hidden and collapsed. Number of vector compute instructions.</td>
<td>Offloaded Regions</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Non-Offloaded Regions</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Call Tree</td>
</tr>
<tr>
<td>Vector Compute with Memory</td>
<td>Hidden and collapsed. Number of vector compute with memory instructions.</td>
<td>Offloaded Regions</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Non-Offloaded Regions</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Call Tree</td>
</tr>
<tr>
<td>Compute Other / Iteration</td>
<td>Hidden and collapsed. Average number of compute other instructions per iteration.</td>
<td>Offloaded Regions</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Non-Offloaded Regions</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Call Tree</td>
</tr>
<tr>
<td>Non-Compute Other / Iteration</td>
<td>Hidden and collapsed. Average number of non-compute other instructions per iteration.</td>
<td>Offloaded Regions</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Non-Offloaded Regions</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Call Tree</td>
</tr>
</tbody>
</table>

**Troubleshooting Reference**

This section describes known problems and questions you may encounter when analyzing your application with the Intel® Advisor Beta and suggests solutions or steps for further investigation.

**Broken Call Tree**

**Symptoms**

In the Call Tree section of the HTML report, you see one of the following:

- A code region is duplicated.
- A code region is located at a wrong place.
- A code region has incorrect number of trip counts reported in any column of the Trip Counts column group.
- A code region with your code has a *System Module* message in the Diagnostics column and *Cannot be modeled: System Module* message in the Why Not Offloaded column.

Any of these symptoms mean that the Offload Advisor detected the application call tree incorrectly during Survey.

**Details**

A broken call tree often happens if you use a program model with Data Parallel C++ (DPC++) or Threading Building Block (TBB). These program models run code in many threads using a complicated scheduler, and the Intel® Advisor Beta sometimes cannot correctly detect their call stacks. As a result, some code instances might have no metrics or incorrect metrics in a report and a call tree is broken.

**Cause**

This can happen due to the following reasons:

- Call stacks were detected incorrectly.
- A heavy optimization was used.
• Debug information has issues.

**Possible Solution**

**NOTE**
This is not an issue if all hotspots and code you are interested in are outside of the broken part of the call tree. You can ignore it in this case.

To fix a broken call tree, do the following:

• Make sure you compiled binary with `-g` option.

**NOTE**
You can recompile it with the `-debug inline-debug-info` option to get enhanced debug information.

• Recompile the binary with a lower optimization level: use `-O2`.

• **If you collect performance metrics with advixe-cl:** When running the Survey analysis, try the following:
  • Remove `--stackwalk-mode=online` option.
  • Add `--no-stack-stitching` option.
  • Offload only specific code regions if their estimated execution time on a target device is greater than or equal to the original execution time. Rerun the performance modeling with `--select-loops` to specify loops of interest and `--enforce-offloads` to make sure all of them are offloaded. For example:

```
advixe-python <APM>/analyze.py <project-dir> --select-loops=[<file-name1>:<line-number1>,<file-name1>:<line-number2>,<file-name2>:<line-number3>] --enforce-offloads
```

**NOTE** Replace `<APM>` with `$APM` on Linux* OS or `%APM%` on Windows* OS.

• If you model a multithreaded code that runs with a complicated scheduler, you might see a code region with suspiciously low trip counts and multiple instances of the same region loop present in the scheduler. This means that the Offload Advisor could not correctly detect the call stacks. Use the `--enable-batching` option to artificially increase the number of trip counts by using total number of executions instead of average number trip counts.

**See Also**
Command Line Interface Reference
`analyze.py` Options

**Code Region is not Marked Up**

**Symptoms**
A code region of interest is not analyzed and has **Outside of Marked Region** message in the Why Not Offloaded column of the HTML report.

**Details**
To limit the scope of collections, the Intel® Advisor Beta selects loops that match certain criteria and marks them up for analysis. By default, the Intel® Advisor Beta performs a smart region selection using the **generic** markup.
If a code region does not satisfy the markup criteria, you should see the *Outside of Marked Region* message in the Why Not Offloaded column or the *System Module* message in the Diagnostics column in the HTML report.

**Cause**

Your code region does not satisfy one or more markup rules for a specified markup mode. If you use the default generic mark-up strategy, make sure your loop of interest satisfies the following rules:

- It is not a system module or a system function.
- It has instruction mixes.
- It is executed.
- Its execution time is not less than 0.02 seconds, which is a sampling interval of the Intel® Advisor Beta.

For more information about execution time limitations, see **Total Time is Too Small for Reliable Modeling**.

**Possible Solution**

If a code region does not satisfy the generic markup rules, but you want to analyze it, do one of the following:

- You can change the markup strategy by using a `--markup=<markup_mode>` option with a parameter other than *generic*. The following parameters select only loops inside regions that are already parallel:
  - `regions`: Select loops in OpenMP*, Data Parallel C++ (DPC++), Intel® Data Analytics Acceleration Library (Intel® DAAL), Threading Building Blocks (TBB) parallel regions.
  - `omp`: Select loops only in OpenMP parallel regions.
  - `dpcpp`: Select loops only in DPC++ parallel regions.
  - `daal`: Select loops only in Intel DAAL parallel regions.
  - `tbb`: Select loops only in TBB parallel regions.

  **NOTE**
  
  *omp*, *dpcpp*, and *generic* select loops in the project so you can run another collection or performance modeling without markup or loop selection options.

- If your loops of interest are not marked up because they have no static instruction mixes or not executed, you can limit the analysis to these specific loops by using the `--select-loops` option with the `analyze.py` script. With this option, the Offload Advisor analyzes only the loops specified and ignores other code regions. For example:

  ```bash
  advixe-python <APM>/analyze.py <project-dir> --select-loops=[<file-name1>:<line-number1>,<file-name1>:<line-number2>,<file-name2>:<line-number3>]
  ```

  **NOTE** Replace `<APM>` with `$APM` on Linux* OS or `%APM%` on Windows* OS.

**See Also**

analyze.py Options

**Error Message: Memory Model Cache Hierarchy Incompatible**

**Cause**

You may get the *Error: Memory model cache hierarchy incompatible* error message if you run performance modeling with `analyze.py` on the results collected with a previous release of the Intel® Advisor Beta. This happens because the cache configuration file from a previous release is incompatible with the higher versions of the Intel® Advisor Beta.
Possible Solution
Delete the `perf_models` directory from the results and re-run `analyze.py`.

Run Performance Modeling

High Dependencies Bound-by

Symptoms
You see one or both of the following symptoms:

- In the **Summary** tab of the HTML report, there is a high percentage of **Dependencies** in the **Offloads Bounded-by** pane.
- A lot of code regions have the **Dependency: Assumed** value in the **Dependency Type** column of the metric table.

Cause
If you do not run the Dependencies analysis, Intel® Advisor Beta uses information about a loop dependencies from a compiler or from explicit code mark-up, for example, with a programming model (OpenMP*, DPC++, Threading Building Blocks). If there is not information in the compiler or the loop is not explicitly marked as parallel, Intel® Advisor Beta assigns **Dependency: assumed** type to it.

Loops with dependencies are not recommended for offloading.

Possible Solution
Do one of the following:

- Run the Dependencies analysis:
  - Use the **--markup generic** option with `collect.py`.
  - Re-run the collection with the Dependencies analysis enabled.
- If the Dependencies overhead is still high, you can skip the Dependencies analysis and use the **--assume-parallel** option with `analyze.py` to mark all code regions with assumed dependencies as **Parallel**.

See Also
- `analyze.py` Options
- `collect.py` Options

Troubleshooting Why Not Offloaded

Symptoms
A code region of interest is not recommended for offloading. It is not present in the Offloaded Regions section, but is listed in the Non-Offloaded Regions section with a reason in the **Why Not Offloaded** column.

Details
The **Why Not Offloaded** column in the Non-Offloaded Regions tab (**Offload Information** column group) and the corresponding **Top Non-Offloaded** pane in the Summary tab list reasons why certain code regions are not recommended to be offloaded to a target device. In this column, you can see different reasons for not offloading a code region:

**Why Not Offloaded: Cannot Be Modeled**

Symptoms
A code region of interest has **Cannot Be Modeled** as a reason why it is not offloaded.
### Cause and Solution
The *Cannot be modeled* reason is always followed by one of the following messages that clarify what makes offloading impossible.

<table>
<thead>
<tr>
<th>Message</th>
<th>Cause and Details</th>
<th>Solution</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cannot be modeled: Outside of Marked Region</td>
<td>Offload Advisor cannot model performance for a code region because it is not marked up for analysis.</td>
<td>Make sure a code region satisfies all markup rules or use a different markup strategy. For more information, see Code Region is not Marked Up.</td>
</tr>
</tbody>
</table>
| Cannot be modeled: Not Executed        | A code region is in the call tree, but the Offload Advisor detected no calls to it for a dataset used during Survey. | This can happen if execution time of the loop is very small and close to the sampling interval of the Intel® Advisor. Such loops can have significant inaccuracies in time measurement. By default, the sampling interval for Survey is 0.01 seconds. You can try to decrease the sampling interval of the Intel Advisor: 1. Run the `collect.py` script with the `--dry-run` option to generate `advixe-cl` command lines appropriate for your application. 2. Copy the first command with `--collect=survey`, add the `--interval=<interval>` option with a value less than 0.01, and run the command. For example:  
   ```bash
   advixe-cl --collect=survey --project-dir=<project-dir> --interval=0.005 --return-app-exitcode --auto-finalize --static-instruction-mix -- <target>
   ``` 3. Copy and paste all other commands generated by `--dry-run` and run them as they are. 4. Rerun performance modeling with `analyze.py` to generate performance prediction output. |
<table>
<thead>
<tr>
<th>Message</th>
<th>Cause and Details</th>
<th>Solution</th>
</tr>
</thead>
</table>
| Cannot be modeled: Internal Error           | *Internal Error* means incorrect data or lack of data because the Offload Advisor encountered issues when collecting or processing data. For example, the internal error can happen if:  
  - The instances of a code region are in the Top-Down view, but some of the instances are not aggregated to the region in the Bottom-Up view.  
  - Code region does not have the Instruction Mix data. | You might try to rerun the collection and performance projection to fix the metrics attribution problem. If this does not help, please use the Analyzers community forum for technical support.                                                                 |
| Cannot be modeled: System Module           | This code region is a system function/loop.                                                                                                                                                                                                                                 | This is not an issue. If this code region is inside an offload region, or a runtime call, its execution time is added to execution time of offloaded regions.                                                                                                                    |
| Cannot be modeled: No Execution Count      | The Offload Advisor detected no calls to a loop during Trip Count analysis and no information about the Execution Counts is available for this loop.  
This can happen if the Offload Advisor incorrectly attributed metrics collected to the application call tree.                                                                                   | Verify the issue:  
1. Locate your loop of interest in the Call Tree section of the HTML report.  
2. Scroll to *Trip Counts* column group and check values reported in the *Call Count* and *Average Trip Counts* (for loops) columns. If there are no execution counts for the loop, these columns do not contain a value or report zero.  
3. If *Call Count* and *Average Trip Counts* report zero or no value, make sure the call tree is not broken. For more information, see the Broken Call Tree.  
If the call tree is not broken and the *Call Count* and *Average Trip Counts* report zero or no value, there is a problem with metrics attribution.  
Try to rerun the collection and performance projection to fix the metrics attribution problem. |
Broken Call Tree
Total Time is Too Small for Reliable Modeling
Reported Metrics

Why Not Offloaded: Less or Equally Profitable Than Children/Parent Offloads

Symptoms
A code region has *less or equally profitable than children offloads* or *less or equally profitable than parent offload* as a reason why it is not offloaded.

Cause and Solution

<table>
<thead>
<tr>
<th>Message</th>
<th>Details and Cause</th>
<th>Solution</th>
</tr>
</thead>
</table>
| Less or equally profitable than children offloads | Offloading child loops/functions of this code region is more profitable than offloading the whole region with all its children. This means that the *Estimated Time on Target Device (+Host)* for the region of interest is greater than or equal to the sum of *Estimated Time on Target Device (+Host)* of its child regions profitable for offloading. See the following metrics to identify a specific reason that prevents offloading:
  - Total execution time metrics reported in the *Offload Information* column group
  - Taxes in the *Overhead* column group
  - Information about trip counts in the *Trip Counts* column group
  - Dependencies in the *Dependency Type* column of the *Loop/Function* column group | **Solution 1.** Disable analyzing child loops of all region heads using the `--no-model-children` option with `analyze.py`. With this option, the Offload Advisor only considers the region heads for potential offloading. **Solution 2.** You can tell Offload Advisor to model offloading for only specific code regions even if they are not profitable. Rerun the performance modeling with `--select-loops` to specify loops of interest and `--enforce-offloads` to make sure all of them are offloaded. For example: `advixe-python <APM>/analyze.py <project-dir> --select-loops=[<filename1>:<linenumber1>,<filename1>:<linenumber2>,<filename2>:<linenumber3>] --enforce-offloads` |
| Less or equally profitable than parent offload | Offloading a whole parent code region of the region of interest is more profitable than offloading any of its child regions | You can tell Offload Advisor to model offloading for only specific code regions even if they are not profitable. |

*NOTE* In the commands below, replace `<APM>` with `$APM` on Linux* OS or `%APM%` on Windows* OS.
<table>
<thead>
<tr>
<th>Message</th>
<th>Details and Cause</th>
<th>Solution</th>
</tr>
</thead>
<tbody>
<tr>
<td>separately. This means that the Estimated Time on Target Device (+Host) for the region of interest is greater than or equal to the Estimated Time on Target Device (+Host) of its parent region. Offloading a child code region might be limited by high offload taxes.</td>
<td>Rerun the performance modeling with --select-loops to specify loops of interest and --enforce-offloads to make sure all of them are offloaded. For example: advixe-python &lt;APM&gt;/analyze.py &lt;project-dir&gt; --select-loops=[&lt;filename1&gt;:&lt;line-number1&gt;,&lt;filename1&gt;:&lt;line-number2&gt;,&lt;filename2&gt;:&lt;line-number3&gt;] --enforce-offloads</td>
<td></td>
</tr>
</tbody>
</table>
or Dependency: waw) or assumed (Dependency: assumed) dependency.

3. If your code region of interest has a real dependency, rewrite your code to get rid of the dependency limitations and rerun both the metrics collection and performance modeling.

You can refer to the Dependencies Problem and Message Types Reference for details on potential dependencies problems.

4. If your code region of interest has an assumed dependency, rerun the performance modeling with --assume-parallel or --set-parallel=[<IDs/source-locations>] option to reduce the number of code regions with dependencies.

For example:
advixe-python <APM>/analyze.py <project-dir> --set-parallel=[<file-name1>:<line-number1>,<file-name1>:<line-number2>,<file-name2>:<line-number3>]

**NOTE** These options do not resolve dependencies issues, they tell the Offload Advisor to assume that certain code regions do not have dependencies.

**Solution 2.**

You can tell Offload Advisor to model offloading for only specific code regions even if they are not profitable.
Not profitable: The Number of Loop Iterations is not enough to fully utilize Target Device capabilities

The number of loop iterations is not enough to fully utilize Target Device capabilities and benefit from offloading.

For example, if a target device can execute up to 1024 threads in parallel, but a loop has only 100 iterations, offloading of this loop can result in up to 924 threads being inactive. This may significantly reduce the benefit of using the target device.

See the Estimated Time on Target Device (+Host) and Baseline Elapsed Time columns of a metrics table.

In most cases, such code regions cannot benefit from offloading. If you still want to offload such loops, you can try to use one of the following workarounds:

**Solution 1.**

If a loop is broken down into several chunks by a compiler or a program model, use --enable-batching or --threads=<number-of-threads> option with analyze.py. For the --threads option, specify the number of parallel threads equal to the target device capacity.

**Solution 2.**

You can tell Offload Advisor to model offloading for only specific code regions even if they are not profitable.

Rerun the performance modeling with --select-loops to specify loops of interest and --enforce-offloads to make sure all of them are offloaded. For example:

```
advixe-python <APM>/analyze.py <project-dir> --select-loops=[<filename1>:<linenumber1>,<filename1>:<linenumber2>,<filename2>:<linenumber3>] --enforce-offloads
```
Not profitable: Data Transfer Tax is greater than Computation Time and Memory Bandwidth Time

Message | Details and Cause | Solution
---|---|---
Not profitable: Data Transfer Tax is greater than Computation Time and Memory Bandwidth Time | Time spent on transferring data to a target device (Data Transfer Tax) is greater than Computation Time and Memory Bandwidth Time. The resulting Estimated Time on Target Device (+Host) with data transfer tax is greater than or equal to the Baseline Elapsed Time. | Check metrics in the Data Transfer column group. Large values in any of the columns mean that this code region cannot benefit from offloading. If you still want to offload such loops, you can try to use one of the following workarounds:

- Rerun the collection with the --no-data-transfer option to disable data transfer analysis and use only estimated execution time for speedup and profitability calculation.

For example:
```bash
advixe-python <APM>/collect.py <project-dir> --no-data-transfer -- <target> [target-options]
```

**NOTE** This option disables data transfer analysis for all loops. You may get different performance modeling results for all loops.

- Rerun the performance modeling with --select-loops and --enforce-offloads to offload only specific code regions even if they are not profitable.

For example:
```bash
advixe-python <APM>/analyze.py <project-dir> --select-loops=[<filename1>:<linenumber1>,<filename2>:<linenumber2>] --enforce-offloads
```
Not profitable: Computation Time is high despite the full use of Target Device capabilities

The code region uses the Target Device capabilities to the full, but the Computation Time is still high. As a result, the Estimated Time on Target Device (+Host) is greater than or equal to the Baseline Elapsed Time.

Solution

Check the value in the *Total Time by Compute* column in the *Offload Information* column group.

- High value means that this code region cannot benefit from offloading.
- Unexpectedly high value indicates a problem with a programming model used.

If you still want to offload such loops: You can tell Offload Advisor to model offloading for only specific code regions even if they are not profitable. Rerun the performance modeling with

--select-loops to specify loops of interest and

--enforce-offloads to make sure all of them are offloaded.

For example:

```
advixe-python <APM>/analyze.py <project-dir> --select-loops= [<file-name1>:<line-number1>,<file-name2>:<line-number2>,<file-name3>:<line-number3>] --enforce-offloads
```

Not profitable: Cache/Memory Bandwidth Time is greater than other execution time components on Target Device

The *Total Execution Time* spent in Cache or Memory Bandwidth takes a big part of the Estimated Time on Target Device (+Host). As a result, it is greater than or equal to the Baseline Elapsed Time.

In the actual report, the Cache/Memory is replaced with a specific cache or memory level that prevents offloading, for example, L3 or SLM.

Solution 1.

1. Local your loop of interest in the Call Tree section of the HTML report.
2. Scroll to the *Offload Information* column group and expand it by double-clicking the title.
3. Examine the values in one of the Total Execution Time by Cache/Memory columns for the memory level reported in
### Solution 1.

1. Locate your loop of interest in the Call Tree section of the HTML report.
2. Scroll to the Overhead column group and expand it by double-clicking the title.
3. Examine values in the columns of the Overhead group. Big value in any of the columns means that this code region cannot benefit from offloading because cost of offloading is high.

### Solution 2.

You can tell Offload Advisor to model offloading for only specific code regions even if they are not profitable.

Rerun the performance modeling with `--select-loops` to specify loops of interest and `--enforce-offloads` to make sure all of them are offloaded.

For example:

```
advixe-python <APM>/ analyze.py <project-dir> --select-loops=[<file-name1>:<line-number1>,<file-name1>:<line-number2>,<file-name2>:<line-number3>] -- enforce-offloads
```

### Not profitable because of offload overhead (taxes)

Not profitable because of offload overhead (taxes) is due to high invocation, kernel, code transfer, and data transfer taxes that make the total estimated time on target device (+Host) greater than or equal to the baseline elapsed time.
You can tell Offload Advisor to model offloading for only specific code regions even if they are not profitable.

Rerun the performance modeling with `--select-loops` to specify loops of interest and `--enforce-offloads` to make sure all of them are offloaded.

For example:

```
advixe-python <APM>/analyze.py <project-dir> --select-loops=[<filename1>:<linenumber1>,<filename1>:<linenumber2>,<filename2>:<linenumber3>] --enforce-offloads
```

### See Also

- Reported Metrics
- `analyze.py` Options

### Why Not Offloaded: N/A - Part of Offload

#### Symptoms

A code region of interest has **n/a - part of offload** as a reason why it is not offloaded.

#### Cause

This happens if offloading a code region is less profitable than offloading its outer loop.

#### Possible Solution

This is not an issue. The code region of interest is located inside of an offloaded loop.

### Why Not Offloaded: Total Time is Too Small for Reliable Modeling

#### Symptoms

A code region of interest has **Total time is too small for reliable modeling** as a reason why it is not offloaded.

#### Cause

This can happen if the execution time of a code region or a whole loop nest is less than 0.02 seconds. In this case, Offload Advisor cannot estimate the speedup correctly and say if it is worth to offload the code regions because its execution time is close to the sampling interval of the Intel® Advisor Beta.
Possible Solution
If you want to check the profitability of offloading code regions with total time less than 0.02 seconds, use the `--loop-filter-threshold=0` option when running performance modeling with `analyze.py` to model such small offloads.

For example:
```
advixe-python <APM>/analyze.py <project-dir> --loop-filter-threshold=0 [options]
```

**NOTE** Replace `<APM>` with `$APM` on Linux* OS or `%APM%` on Windows* OS.

See Also
analyze.py Options

Troubleshooting Diagnostics

Symptoms
A code region has a message in the *Diagnostics* column of the *Diagnostics* column group.

Details
The *Diagnostics* column in the Offloaded Regions, Non-Offloaded Regions, and Call Tree tabs lists diagnostics messages that report details that can affect model accuracy.

**NOTE**
For offloaded region heads, diagnostic messages, if any, are reported for a whole loopnest.

You can see different messages in this column:

*Diagnostics: Aggregated Execution Count was Used*

**Symptoms**
A code region of interest has *Aggregated execution counts were used* diagnostics message in the *Diagnostics* column.

**Cause**
This means that execution counts in the Top-Down are not available or incorrect, so Offload Advisor aggregated the loop/function to the Bottom-Up and used the aggregated execution counts to estimate performance.

**Possible Solution**
- This may not be a problem if your kernel is called on a single call path. No correction required.
- If your kernel is called on multiple call paths, you might try to rerun the collection and performance projection to fix the metrics attribution problem. If this does not help, please use the [Analyzers community forum](https://www.analyzers.com) for technical support.

*Diagnostics: No Execution Counts*

**Symptoms**
A code region of interest has *No execution counts message* in the *Diagnostics* column.
Details
With this diagnostic message, you also get the Cannot be modeled: No execution count reason in the Why Not Offloaded column.

Cause
This means that a code region cannot be modeled because information about the execution count is not available or the execution count is zero.

Possible Solution
See the Why Not Offloaded: Cannot Be Modeled topic for details on possible ways to fix this.

See Also
Why Not Offloaded: Cannot Be Modeled
Diagnostics: System Module

Symptoms
A code region of interest has System module diagnostics message in the Diagnostics column.

Details
With this diagnostic message, you also get the Cannot be Modeled: System Module reason in the Why Not Offloaded column.

Cause
This means that a code region cannot be modeled because it is a system module or a system function.

Possible Solution
No correction is required. If this code region is inside an offload region, or a runtime call, its execution time is added to execution time of offloaded regions.

See Also
Why Not Offloaded: Cannot Be Modeled

MPI Workloads

Analyzing MPI Workloads
Intel® Advisor allows you to analyze parallel tasks running on a cluster, so you can examine your MPI application for opportunities of adding vectorization or threading parallelism. Use the Intel® MPI gtool with mpiexec to invoke the advixe-cl command and spawn MPI processes across the cluster.

MPI analysis can be performed only through the command line interface, but the result can be viewed through the standalone GUI, as well as the command line.

Tips
Consider the following when running collections for an MPI application:
• Only homogenous clusters are supported.
• Analysis data can be saved to a shared partition, or to local directories on the cluster.
• Only one processes' data can be viewed at a time.
• Intel® Advisor saves collection results into a subdirectory under the Intel Advisor project directory. If you wish to collect and then view (in a separate session) data for more than one process, specify a new project directory when running new collection.
• Specify one and the same project directory when running various Intel Advisor collections for the selected process.

**MPI Implementations Support**

You can use the Intel Advisor with the Intel® MPI Library and other MPI implementations, but be aware of the following details:

• You may need to adjust the command examples in this section to work for non-Intel MPI implementations. For example, adjust commands provided for process ranks to limit the number of processes in the job.
• An MPI implementation needs to operate in cases when there is the Intel Advisor process (advixe-cl) between the launcher process (mpiexec) and the application process. This means that the communication information should be passed using environment variables, as most MPI implementations do. Intel Advisor does not work on an MPI implementation that tries to pass communication information from its immediate parent process.

**Intel® MPI Command Syntax**

You can use Intel Advisor to generate the command line for collecting results on multiple MPI ranks. To do that,

1. In Intel Advisor user interface go to **Project Properties > Analysis Target tab > Survey Analysis Types > Survey Hotspots Analysis**.
2. Select the **Use MPI Launcher** checkbox.
3. Specify the MPI run parameters, then copy the command line from **Get command line** text box to your clipboard.

Use the `-gtool` option of mpiexec with Intel® MPI Library 5.0.2 and higher:

```
$ mpiexec -gtool "advixe-cl -collect <analysis_type> -project-dir <project_PATH>:<ranks_set>" -n <N> <application_name> [myApplication_options]
```

where:

• `<analysis_type>` is one of the Intel Advisor tools:
  - `survey` runs the target process and collects basic information about the hotspots.
  - `tripcounts` collects data on the loop trip counts.
  - `dependencies` collects information possible dependencies in your application, requires
  - Either loop ID as an additional parameter (`-mark-up-list=<loop_ID>`). Find the loop ID in the Survey report (`--report survey`) or using the **Command Line** link in the Intel® Advisor GUI **Workflow tab** (right under the **Collect** button).
  - Or annotations in the source code.
  - `map` collects information about memory access patterns for the selected loops. Also requires loop IDs for the analysis.
  - `suitability` checks suitability of the parallel site that you want to insert into your target application. Requires annotations to be added into the source code of your application, and also requires recompilation in Debug mode.
  - `<ranks_set>` is the set of MPI ranks to run the analysis for. Separate ranks with a comma, or use a dash "-" to set a range of ranks. Use all to analyze all the ranks.

For detailed syntax, refer to the *Intel® MPI Library Reference Manual*. 

---

264
Generic MPI Command Syntax

Use mpiexec with the advixe-cl command to spawn processes across the cluster and collect data about the application.

Each process has a rank associated with it. This rank is used to identify the result data.

To collect performance or dependencies data for an MPI program with Intel Advisor, the general form of the mpiexec command is:

```sh
$ mpiexec -n <N> advixe-cl -project-dir <project_PATH> -collect <analysis_type> -search-dir src:r=<sources_PATH> -- myApplication [myApplication_options]
```

where:

- `<N>` is the number of MPI processes to launch
- `<project_PATH>` specifies the path/name of the project directory
- `<analysis_type>` is survey, suitability or dependencies
- `<sources_PATH>` is the path to the directory where annotated sources are stored

Viewing Results via Intel Advisor

As a result of collection, Intel Advisor creates a number of result directories in the directory specified with `-project-dir`. The nested result directories are named as `rank.0`, `rank.1`, ... `rank.n`, where the numeric suffix `n` corresponds to the MPI process rank.

To view the collected results for a specific rank, you can either open a result project file (`*.advixeproj`) that resides in the `-project-dir` via the Intel Advisor GUI, or run the Intel Advisor CLI report:

```sh
$ advixe-cl -report <analysis_type> -project-dir <project_PATH>:<ranks>
```

You can view only one rank’s results at a time.

Additional MPI Resources

For more details on analyzing MPI applications, see the Intel MPI Library and online MPI documentation on the Intel® Developer Zone at https://software.intel.com/content/www/us/en/develop/tools/mpi-library/get-started.html

Other Intel® Developer Zone online resources that discuss usage of the Intel® Parallel Studio XE Cluster Edition with the Intel MPI Library:


MPI Workflow Example

In this example, analysis is performed for an application that is run in four processes.

Collect survey data for all ranks into the shared `./advi` project directory.

```sh
$ mpirun -n 4 advixe-cl -project-dir ./advi -collect survey -- <PATH>/mpi-sample/1_mpi_sample_serial
```

If you need to copy the data to the development system, do so now.

Import the survey result from rank 3 to the development system and finalize it.

```sh
$ advixe-cl -project-dir ./new-advi -import-dir ./advi -mpi-rank 3 -search-dir src:=<PATH>/mpi_sample
```

Open the survey in the Intel® Advisor standalone GUI for viewing.

```sh
$ advixe-gui ./new-advi
```
Run a Suitability analysis for all ranks.
$ mpirun -n 4 advixe-cl -project-dir ./advi -collect suitability -- <PATH>/mpi_sample/2_mpi_sample_annotated

Run a Dependencies analysis for all ranks.
$ mpirun -n 4 advixe-cl -project-dir ./advi -collect dependencies -- <PATH>/mpi_sample/2_mpi_sample_annotated_debug

If you need to copy the data to the development system, do so now.
Import and finalize your data.
$ advixe-cl -project-dir ./new-advi -import-dir ./advi -mpi-rank 3 -search-dir src:=<PATH>/mpi_sample

Open the GUI to view your result.
$ advixe-gui ./new-advi

**NOTE**
When using a shared partition on Windows*, either the network paths must be used to specify the project and executable location, or the MPI options `mapall` or `map` can be used to specify these locations on the network drive.

For example:
$ mpiexec -gwdir \\
\<host1>\mpi -hosts 2 <host1> 1 <host2> 1 advixe-cl -collect survey -project-dir \\
\<host1>\mpi\advi -- \<host1>\mpi\mpi_sample.exe

$ advixe-cl -project-dir \\
\<host1>\mpi\new-advi -import-dir \\
\<host1>\mpi\advi -search-dir src:=\<host1>\mpi\mpi_sample.exe

$ advixe-cl -project-dir \\
\<host1>\mpi\new-advi -report survey

Or:
$ mpiexec -mapall -gwdir z:\ -hosts 2 <host1> 1 <host2> 1 advixe-cl -collect survey -project-dir z:\advi -- z:\mpi_sample.exe

Or:
$ mpiexec -map z:\<host1>\mpi -gwdir z:\ -hosts 2 <host1> 1 <host2> 1 advixe-cl -collect survey -project-dir z:\advi -- z:\mpi_sample.exe

**See Also**
MPI Workloads

**Results**
Data collected by running Intel Advisor tools is stored in a result. When you run one of its tools, the Intel Advisor executes a target, identifies issues that may need handling, collects the results and shows it in the Results subdirectory in the **Solution Explorer** in Microsoft Visual Studio* or in the **Project Navigator** in the Intel Advisor Standalone GUI.

View the data in the Result tab to help you choose the best places to add parallelism. There is one result for each project or each project in the Solution (on Windows* OS). If you run an Intel Advisor tool on the same project, any previously collected results are overwritten.
**Result Locations**
The Intel Advisor saves results in the `Results` directory in a solution directory on Windows OS (the default Microsoft Visual Studio* location) or a subdirectory of the specified Intel Advisor project location in the Project Navigator on Linux* OS. You can specify a custom location for saving results.

**Results and the Solution Explorer**
You can view results associated with a project in the Solution Explorer in Visual Studio or the Project Navigator in the Intel Advisor GUI. However, sometimes other considerations outweigh accessibility. For example, in Visual Studio, do not display results in the Solution Explorer if you use a source code control system and you do not want to check in your Solution Explorer with embedded results.

**Opening a Result**
There is one result for each project. To open a previously collected result for one project:

<table>
<thead>
<tr>
<th>To Do This</th>
<th>Do This</th>
</tr>
</thead>
<tbody>
<tr>
<td>After opening its Solution, open a result from the Microsoft Visual Studio* IDE or the Intel Advisor GUI.</td>
<td>After you open the Visual Studio solution or the Intel Advisor GUI project, do one of the following:</td>
</tr>
<tr>
<td></td>
<td>• From the Microsoft Visual Studio* Solution Explorer, double-click the result in the Results program folder.</td>
</tr>
<tr>
<td></td>
<td>• From the Microsoft Visual Studio* menu:</td>
</tr>
<tr>
<td></td>
<td>1 Choose Open &gt; File.</td>
</tr>
<tr>
<td></td>
<td>2 In the Open File dialog box, browse to and double-click the result. By default the <code>ennn.advixe</code> file in <code>project name</code> folder in the solution or project directory.</td>
</tr>
<tr>
<td></td>
<td>• From the Intel Advisor GUI menu:</td>
</tr>
<tr>
<td></td>
<td>1 Choose either File &gt; Open &gt; Result... or File &gt; Recent Results.</td>
</tr>
<tr>
<td></td>
<td>2 In the Project Navigator, browse to and double-click the result. By default the <code>ennn.advixe</code> file in <code>project name</code> folder in the solution or project directory.</td>
</tr>
</tbody>
</table>

**NOTE** You can open multiple results from the same project only,

| Open a result from the Solution Explorer or the Intel Advisor product GUI or its Project Navigator. | After you open the Visual Studio solution, browse to and double-click the result. By default, the results are in the Advisor Result directory in the Solution Explorer. |
| | From the product GUI, click File > Open > Result... or File > Recent Results > name. |
| | When using the Project Navigator in the product GUI, navigate to the project and click its result name, such as `ennn`. |

**NOTE** You can open multiple results from the same project only,

| View a specific result | If you have opened multiple results for different projects and you would like to view a result that is not displayed, click its tab to view that result. The result appears showing the last report that you viewed. |
| | To view a specific result, click its tab name: |

![Welcome e000 x e001](image-url)
To Do This | Do This
---|---

NOTE You can open multiple results from the same project only.

To rearrange the order of the displayed tabs, drag a tab to the desired location.

You can create a snapshot of the active result and save as a read-only result (see the help topic Creating a Read-only Result Snapshot).

See Also
Renaming Existing Results
Creating a Read-only Result Snapshot
Opening a Result as a Read-only File

Renaming Existing Results

NOTE
If you rename results using the file system or Windows* Explorer software instead of the Intel Advisor Standalone GUI or the Microsoft Visual Studio* IDE, you may create an error condition.

NOTE
If you change the .advixe extension, you will not be able to open the result file in Intel Advisor.

To rename an existing result

1. Right-click the result folder in the Solution Explorer or the result in the Intel Advisor GUI to display a context menu, then choose Rename.
2. Type the new name.
3. Press the Enter key.

The result is renamed in the project and on your system.

See Also
Deleting a Result
Saving Results to a Custom Location

Deleting a Result

To delete a result:

1. Right-click the result in the Solution Explorer in Visual Studio or the Project Navigator in the Intel Advisor GUI to display a context menu.
2. When using Visual Studio, choose Remove. In the resulting dialog box, click the Delete button.
3. When using the Intel Advisor GUI Project Navigator, click Delete.

The result is removed from the project and deleted from your system.

See Also
Creating a Read-only Result Snapshot
Saving Results to a Custom Location
Creating a Read-only Result Snapshot

Only the active result for a project can be modified by collecting new data. You can create a read-only result snapshot after you collect data from one or more Intel Advisor tools. To create a snapshot of a result and save its data in a read-only result:

1. Display the active result (not a read-only result snapshot).
2. Click the button next the window caption. The Create a Result Snapshot dialog box appears.
3. Type the Result name of the Intel Advisor read-only result. Provide a unique name, perhaps by adding an identifying suffix within the result name.
4. Click OK. With a large result, you may need to wait as the read-only result gets created.

You can visually compare saved read-only results against the current active result or other read-only results. The read-only result appears in the Intel Advisor GUI Project Navigator and the Visual Studio Solution Explorer (for Windows* OS only). The words (read-only) appear after the name of a read-only result in the result tab. The icon of a read-only result differs from that of the active result.

Although you can only collect data to update the active project, you can update your sources regardless of what type of result (or project) you have open.

See Also
Saving Results to a Custom Location

Create a Result Snapshot Dialog Box

Dialog Box Purpose and Usage | Dialog Box Access | Dialog Box Controls

Create a Result Snapshot Dialog Box Purpose and Usage

Intel Advisor stores only the most recent analysis result. Use this dialog box to save a read-only result snapshot you can view any time.

Tip
- Visually comparing one or more snapshots to each other or to the most recent analysis result can be an effective way to judge performance improvement progress.
- To view a snapshot, choose File > Open > Result...
- Snapshots are identified by a different icon in the Visual Studio Solution Explorer and the Intel Advisor Project Navigator. The words (read-only) appear after the snapshot name in a result tab.

Create a Result Snapshot Dialog Box Access

1. Display an active result (not a snapshot).
2. Click the button in the Filters pane.

Create a Result Snapshot Dialog Box Controls

<table>
<thead>
<tr>
<th>Use This</th>
<th>To Do This</th>
</tr>
</thead>
<tbody>
<tr>
<td>Result name</td>
<td>Specify the name of the read-only result snapshot. Provide a unique name, perhaps by adding an identifying suffix within the result name.</td>
</tr>
<tr>
<td>Cache sources</td>
<td>Enable source code availability in the resulting snapshot.</td>
</tr>
<tr>
<td>Use This</td>
<td>To Do This</td>
</tr>
<tr>
<td>--------------------------</td>
<td>---------------------------------------------------------------------------</td>
</tr>
<tr>
<td><strong>Cache binaries</strong> checkbox</td>
<td>Enable binary availability in the resulting snapshot.</td>
</tr>
<tr>
<td><strong>Pack into archive</strong> checkbox</td>
<td>Create a one-file archive with all snapshot data inside.</td>
</tr>
<tr>
<td><strong>Result path</strong> text box</td>
<td>Specify the path to the resulting snapshot archive. Use the <strong>Browse...</strong> button to specify the address. Disabled by default. Enable by selecting the <strong>Pack into archive</strong> checkbox.</td>
</tr>
</tbody>
</table>

**Saving Results to a Custom Location**

The Intel Advisor saves a result in a subdirectory of each project's directory. The project directory is in the default Visual Studio location or the directory specified when creating the Intel Advisor Standalone GUI project. Instead of saving results within each project's directory, you can specify a custom, central location for saving all new results.

To save results to a custom location when using the Microsoft Visual Studio* or Intel Advisor GUI:

1. From the Microsoft Visual Studio menu, choose **Tools > Options...**
2. From the Intel Advisor GUI menu, choose **File > Options...**
3. In the **Options** dialog box, expand the **Intel Advisor** program folder and choose the **Result Location** page.
4. Select **Save all results in this directory**:.
5. Click **Browse** to select the custom location.
6. Click **OK**.

The Intel Advisor saves future results to the custom location. The subdirectory name is the result name, such as e000.

**See Also**

Opening a Result as a Read-only File

**Opening a Result as a Read-only File in Visual Studio**

**NOTE**

This topic applies only to the Microsoft Visual Studio* IDE. Alternatively, you can also take a snapshot of the active result and save it as a read-only result using either Visual Studio or the Intel Advisor GUI.

To open a previously collected result file as a read-only file when its corresponding Visual Studio* Solution is **not** open:

<table>
<thead>
<tr>
<th>To Do This</th>
<th>Do This</th>
</tr>
</thead>
<tbody>
<tr>
<td>Open a result from the Microsoft Visual Studio* IDE.</td>
<td>Before opening the Solution, from the Microsoft Visual Studio* <strong>File</strong> menu:</td>
</tr>
<tr>
<td></td>
<td>1. Choose <strong>Open &gt; File</strong>.</td>
</tr>
<tr>
<td></td>
<td>2. In the <strong>Open File</strong> dialog box, browse to and double-click the Intel Advisor result file (file suffix .advixe). By default, Intel Advisor results are saved in the enn.n.advixe file in project name directory in the solution or project directory.</td>
</tr>
</tbody>
</table>

**NOTE**

This action launches the Microsoft Visual Studio* IDE if it is not already open.
Open a result from the Windows* Explorer (or comparable) environment.

From the Windows* Explorer (or comparable) environment:

1. Open the Windows* Explorer (or comparable) environment.
2. Browse to and double-click the result. By default, Intel Advisor results are saved in the `ennn.advice` file in `project name` folder in the solution or project directory.

**NOTE**
This action launches the Microsoft Visual Studio* IDE if it is not already open.

Verify that the result is open as a read-only file.

A result opened as a read-only file contains (read-only) after its result name.

View a specific report for the result

Each result contains several different reports. To view a specific report, click the button for the report you want to view, such as Survey Report, Suitability Report, or Dependencies Report.

When opening a result as a read-only file, you are only viewing the previously collected data as a read-only file. The following limitations apply:

- The Annotation Report window does not contain any data. To view the Annotation Report, you need to open a result after opening its Solution.
- The Suitability Report window shows data, but you cannot change the modeling parameters, such as the Target System or the Threading Model.
- If you modify your sources or Intel Advisor annotations, you need to open the solution to collect data and perform analysis (see Opening a Result) to update the result and view Annotation Report data.

**See Also**
Creating a Read-only Result Snapshot
Opening a Result  (after opening a solution)

**Flow Graph Analyzer**

This section explains how to use Intel® Advisor – Flow Graph Analyzer (FGA), a graphical tool for constructing, analyzing, and visualizing applications that use the Threading Building Blocks (TBB) flow graph interfaces. Through a graphical interface, the Flow Graph Analyzer lets you:

- Start with a blank canvas and construct a flow graph application by interactively adding nodes and edges.
- Collect events during the execution of an existing application. These events allow you to explore the topology and performance of the flow graphs used by that application.

**Where to Find the Flow Graph Analyzer**

The Flow Graph Analyzer package consist of a Flow Graph Analyzer tool and an associated collector to capture traces from Threading Building Blocks (TBB) flow graph and OpenMP* applications.

This package is part of Intel® Advisor and is installed in the following directory when the Intel® Advisor is installed: `<advisor-install-dir>/fga`

Under this directory, you can find the Flow Graph Analyzer tool, the collector, and supporting documentation.

- Flow Graph Analyzer: `<advisor-install-dir>/fga/fga`
- Flow Graph Collector: `<advisor-install-dir>/fga/fgt`
- Documentation: `<advisor-install-dir>/fga/doc`
Launching the Flow Graph Analyzer

The Flow Graph Analyzer executable and all its supporting files are located in the directory `<advisor-install-dir>/fga/fga`. To launch it, go to this directory and do the following:

<table>
<thead>
<tr>
<th>Operating System</th>
<th>To run from an executable</th>
<th>To run from command line</th>
</tr>
</thead>
<tbody>
<tr>
<td>Windows* OS</td>
<td>Double-click fga.exe.</td>
<td>1. Open a command prompt application from <code>&lt;advisor-install-dir&gt;/fga/fga</code>.</td>
</tr>
</tbody>
</table>
| Linux* platforms | 1. Add . to your LD_LIBRARY_PATH environment variable.  
                      2. Double-click fga in the `<advisor-install-dir>/fga/fga`. | 2. Run the run_fga.sh from the command line, which automatically sets the LD_LIBRARY_PATH and then starts the executable. |
| macOS*           | 1. Add . to your DYLD_LIBRARY_PATH environment variable.  
                      2. Add ./Frameworks to the DYLD_FRAMEWORK_PATH environment variable.  
                 2. Run run_fga.sh from the command line. |

**Warning**

Moving the executable to a different location may cause the application to fail.

On Windows* OS without the Microsoft Visual Studio* runtime components, you may encounter the following error when launching the Flow Graph Analyzer application:

![fga.exe - System Error](image)

**Solution:** Download the Microsoft Visual Studio* runtime components and install them before running the application.
Flow Graph Collector

The Flow Graph Collector allows you to capture a topology and execution trace information of a running flow graph application. You can load this captured data to the Flow Graph Analyzer for closer inspection of the graph described by the application and the performance data for the graph. See the Scalability Analysis section for the steps to build your application with enabled traces and to capture the trace information. See the Collecting Traces from Applications for instructions on how to use the Flow Graph Analyzer for analyzing the performance of flow graph applications after a trace has been collected.

Flow Graph Analyzer GUI Overview

This section describes the Flow Graph Analyzer tool and the various features it offers to speed up the development of new flow graph applications. There are two main tools represented by corresponding workflows:

- A **designer workflow** enhances productivity during development.
- An **analyzer workflow** supports performance-tuning tasks.

Basic GUI Layout

The Flow Graph Analyzer GUI allows you to create new graphs visually and load previously created or application-generated graphs.

The following figure describes the basic GUI layout and the key elements necessary for constructing graphs visually. These graphs can be saved for later use and, as described in the Generating C++ Stubs section, used to generate C++ framework code.

1. Toolbar supporting basic file and editing operations, visualization and analytics that operate on the graph or performance traces.
2. Canvas for visualizing and drawing flow graphs.
3. Output generated by custom analytics. You can interact with the output.
GUI Layout with Trace Data

When analyzing an existing application’s performance, the graph topology and performance data are captured from a running application and saved for a post-mortem analysis. If performance traces are available when the graph is loaded, they are displayed in a timeline window below the canvas area. You can interact with the trace data in many ways, from cursory inspection of the trace data to detailed inspection of specific tasks and the nodes they map to. The following figure shows the timeline charts created when trace data is available.

1. Selection on the timeline highlights the nodes executing at that point in time.
2. The concurrency histogram shows the parallelism achieved by the graph over time. You can interact with this chart by zooming in to a region of time, for example, during low concurrency. The concurrency histogram remains at the initial zoom level, and the zoomed-in region is displayed below it.
3. The per-thread task view shows the tasks executed by each thread, along with the task durations.
4. Treemap view provides the general health of the graph’s performance, along with the ability to dive to the node level.
Menus

The menus in the menu bar have fixed components such as File, Edit and Help, and dynamic components such as Layouts, Analytics, and Trace Data Collection. The fixed components are always available, and the dynamic components may change depending on the plugins registered with the tool.

- **File** menu: Allows you to create a new graph, load an existing graph, save the current graph on the canvas to a GraphML file, or export it as C++ source files.

  ![File Menu](image.png)

  The menu also keeps a list of recently used files for quick access. Print support is currently unavailable. The Generate Image option enables printing the graph displayed in the canvas as a PNG file.

- **Edit** menu: Allows you to edit the graph displayed on the canvas and supports common edit actions, such as Cut, Copy, Paste, Delete, Group, Ungroup, and Preferences. These actions support the common keyboard shortcuts.

  ![Edit Menu](image.png)

- **Help** menu: Allows you to switch to the What’s This? mode, which provides help information for various GUI elements. In this mode, you can click any GUI element that has supporting help information to view more information about the element and what it helps you accomplish.

  ![Help Menu](image.png)

  You can also get into the What’s This? mode using the keyboard shortcut Shift+F1.
• **Layout** menu: Allows you to visualize a graph on the canvas in different ways. Currently supported layout types are **Hierarchical**, **Radial**, **Force-Directed**, **New Hierarchical**, and **Circular**. For most graphs, the **Hierarchical Layout** is enough and is set as the default layout. If the **Hierarchical Layout** does not work properly for a graph model, you can use the **New Hierarchical Layout**.

**NOTE** The **New Hierarchical Layout** is 3x slower than the default **Hierarchical Layout**.

If you cannot get a visually pleasing layout using the hierarchical layouts, use the **Radial**, **Circular**, or **Force-Directed** layout. The **Circular Layout** and **Force-Directed Layout** use the Boost* Graph library. The cost of running the **Force-Directed Layout** is high compared to other layouts, but it provides better graph layout visuals.

• **Analytics** menu: Allows you to choose an analytical algorithm from available plugins. This menu changes as new plugins are added.

These analytical algorithms are available for Threading Building Blocks (TBB) flow graphs. More algorithms may be added in the future.

- **Compute Critical Path** computes one or more critical paths for a graph and lists them in the Analytics Report tab. You can interact with these critical paths to see which nodes are part of them.
- **Graph Rule-check** performs basic rule checks on a graph and highlights potential performance and correctness problems.
- **Compute Modeling Projection** projects the speedup of a graph with varying numbers of threads. The speedup with the corresponding number of threads is shown in the **Analytics Reports** window, while a chart showing the ideal versus actual speedup is shown in the chart area.

**Toolbars**

The menu items are shown in the toolbar area as shortcuts to frequently used operations. Hover the mouse over an icon on a toolbar to see a tooltip with description.

• **File** toolbar: Provides access to the functionality from the **File** menu.

• **Edit** toolbar: Provides access to the editing operations from the **Edit** menu.

• **Window** toolbar: Provides the show/hide toggle capability for configurable GUI elements. From left to right, these icons represent the **Toolbox**, **Reports**, and **Charts** tabs.
• Hide the Toolbox tab, which contains the Designer Mode, Analysis Mode, and Hierarchical View tabs to increase available screen space for large graphs.
• Hide the Reports tab, which shows information such as node properties or output of analytics algorithms, to get more screen space for visualizing large graphs.
• By default, the Charts tab is hidden if the graph does not have associated execution trace data. When the trace data is available, the Charts tab is displayed.
• Analytics toolbar: Provides a subset of plugins from the Analytics menu. A plugin can be hidden in the toolbar, but it is always registered in the Analytics menu.

The analytics supported for Threading Building Blocks (TBB) flow graphs are, from left to right in the toolbar, Compute Critical Path, Graph Rule-Check, and Compute Modeling Projection. Use these algorithms to design new and tune existing graphs.
• Layout toolbar: Provides a subset of the layouts from the Layout menu.

• Zoom toolbar: Allows you to zoom in or out the canvas area that displays the graph. The reset-zoom button resets the zoom factor, so the entire graph is visible in the canvas area.

**NOTE** You can also zoom in or out using the mouse-wheel when the mouse is in the canvas area.

• Trace Data Collector toolbar: Opens a dialog box to configure a data collection run on a TBB flow graph application. This dialog box also allows you to configure the environment before launching the application.

• What’s This? toolbar: Switches to the What’s This? mode.

**NOTE** You can also switch to this mode using the Shift+F1 keyboard shortcut.

Tabs

• Properties tab: Displays information about a selected graph in the Graph tab, a node in the Node tab, and an edge in the Edge tab.
The **Node** tab displays all properties supported for a given node type and allows you to interactively edit them. Some properties might be tied to the node type and are marked as read-only. See the Designer Workflow section for instructions on editing the properties of a node.

- **Analytics Report** tab: Displays the output generated by any invoked analytics algorithms. Because analytics algorithms generate different outputs, this view changes when depending on an algorithm you run on the graph. The screenshot below shows a sample output for **Compute Critical Path**. The columns in this tab are sortable and enable efficient data manipulation during the performance tuning workflow.

- **Designer Mode** tab: Shows the available node types you can use to construct a graph. This tab might change when new nodes are added to a Threading Building Blocks flow graph interface. For more information about each node type, use the **What's This?** functionality on the node-type buttons.

After you select a node type to insert into the graph, the mouse cursor takes the shape of the selected node type. You may add as many nodes of the same type as you want to by placing the cursor at a location on the canvas and clicking a mouse. To switch to a different node type, select the node type of interest in the tab.

To exit the **Insert Node** mode, press the **ESC** key if the mouse is placed in the canvas area, or select the **Insert Edge** or **Move Node** modes from the **Toolbar** menu.
• **Debug Output** tab: Displays messages output by the tool. Most of the messages are informational, but warning or error messages may appear. **Warning** messages are green, and **Error** messages are yellow.

• **Source View** tab: Displays the source mapping of a selected node if symbol resolution information has been captured during the collection.

**NOTE** Currently, this feature is only supported on Linux® OS. To enable data collection with symbol resolution information, please refer to Building on a Linux® Operating System and Collecting Trace Files with an fgtrun.sh Script.
Main Canvas

The main canvas shows created or loaded graphs.

The following controls are available in this area:

- Zoom in or zoom out of this area using the mouse wheel or the zoom buttons in the toolbar.
- Open a context menu with node-specific options by right-clicking a node.
- Select and move a node by dragging it when in the **Move Node** mode.
- Insert new edges between two nodes when in the **Insert Edge** mode.

Charts

This area displays the task execution trace data available for a graph. You can zoom in or zoom out of all charts in this area to inspect them at various resolutions. Use the mouse wheel or the buttons in the **Zoom** toolbar above the charts to zoom in or zoom out.

The execution traces are displayed in two different forms:

- **Node Concurrency** display: This form includes two charts.
  - The top chart shows the concurrency for the entire length of the application run.
  - The bottom chart shows the details of the zoomed region in the top chart.

Both charts plot the node concurrency over time. The maximum node concurrency is limited to the maximum number of threads in the Threading Building Blocks (TBB) thread pool.
- **Per-Thread Task** display: This chart shows tasks executed by each thread and their duration. To see tasks associated with a particular node, enable the **Show/Hide Selected tasks** button.

You can zoom in or out the data in both views using the specific buttons in the chart toolbar or a mouse wheel. Use the drop-down box in the toolbar to switch between a **Thread View** and a **Node View**.

- In the **Thread View**, the vertical axis is a set of threads that participated in executing the flow graph, and the horizontal axis is time. Tasks with short durations are displayed with a lighter color than those with a longer duration. The lighter color highlights tasks that are small relative to the cost of scheduling the task.

- In the **Node View**, a set of thread timelines is created for each node in the graph. In each set, the vertical axis is a set of threads that participated in executing the flow graph, and the horizontal axis is time.

**NOTE**
In the **Node View**, a node’s set of timelines only displays tasks related to that node, while in the **Thread View**, a single set of timelines shows the tasks related to all nodes.
In some cases, the trace data can contain additional information about the logical core on which a task executes and the data ID it is processing with the help of user-APIs supported by TBB and the Flow Graph Analyzer. When this information is available, you can visualize the Thread View data and color it by core information or by the data being processed.

Flow Graph Analyzer Workflows

To design flow graph applications using the Threading Building Blocks (TBB) library, you need to understand the various node types supported, how to map them to end-user concepts or computational entities, and link them all together to form the flow graph. However, it is difficult to visualize and map such computational blocks when the count of such blocks goes beyond a handful.

To help you solve this visualization problem, the Flow Graph Analyzer tool supports two workflows:

- **Designer workflow** – Enables expressing the relationships between nodes using TBB flow graph node types.
- **Analyzer workflow** – Enables capturing and viewing graph topologies and related performance data captured from executed applications.

**Designer Workflow**

The Flow Graph Analyzer designer workflow allows you to describe your graphs visually, set meaningful properties, and generate the flow graph framework code in C++. Before generating the framework code, you can also perform rule-checks to make sure the described graph does not have any potential issues that could lead to incorrect execution or a poorly performing graph. The generated code can be compiled and run without modification in many cases. Sometimes, the generated code may have to be modified to provide meaningful inputs or outputs.

Specifically, the tool supports the following capabilities necessary for visual design of graphs:

- Choose from a variety of available node types to build a graph.
- Express the explicit relationships with edges.
- Edit properties of these nodes.
- Perform common editing operations.
- Save the described graph and reload it later.

In addition to these basic capabilities, the tool provides the means to:

- Validate each node to ensure that flow graph rules are not broken.
- Perform basic rule checks on a graph to identify potential performance problems.
• Export the graph as a C++ framework code that uses the Threading Building Blocks (TBB) flow graph API.
• Export the graph as a PNG image.

This section walks through the design workflow and the capabilities that support it. The following figure shows the simple flow of the design process.

Adding Nodes, Edges, and Ports

When the tool starts up, you are presented with a blank canvas to which you can add nodes from the list of nodes under Designer Mode pane on the left side of the tool.
To add new nodes:

1. Expand a required node group in the Designer Mode pane.
2. Drag the required nodes to the canvas.
3. Add the dependencies, or edges, between the nodes by clicking an output port of a node and dragging to an input port of another node.

To add new ports to a node or delete ports:

1. Right-click a node to open a context menu.
2. Choose Add an Input/Output Port or Delete an Input/Output Port.

You also can add or remove ports from the Port Information tab in the Reports area:

New ports are added to the end of a port list and deleted from the end of the list.

**Modifying Node Properties**

After you add nodes to a graph and connect them with edges, inspect the nodes to ensure the data flowing through the graph has correct types. Some data types are dictated by the Threading Building Blocks (TBB) flow graph node types themselves or by the logic the graph represents. Because the data flows through nodes and edges are connected to ports, the data types are managed at the port level. The default node data types are int for most ports and continue_msg for nodes that expect this type of data.
To edit node properties:

1. Select a node.
2. In the Port Information pane of the Node tab, change the Data Type of a port by selecting its type and editing the field.

   ![Port Information](image1)

   For certain nodes, such as a join_node, only the input port data types can be modified, and the output data type is automatically generated when you update the input port data types.

   ![Port Information](image2)

3. Select a node on the canvas to see its properties in the Node tab. This property pane displays all properties for a given node type. The properties that are not set for the selected node type are shown in a darker color. In the figure below, you can see that the Description property is not set for the join_node.

   ![Node Properties](image3)

   Some of the properties for the nodes are set automatically and tied to the node type. Such properties are not available for editing and the Node Properties tab enforces these rules. For example, you cannot change the Node Type, but you can edit the Node Weight and the Node Name.
• The **Node Weight** is a placeholder that indicates the computational complexity of a node. The larger the number, the more computationally intensive the node is with respect to the other nodes in the graph. This number is also used by the C++ code generator to create a busy loop in the empty body that is created for each node. See the Generating C++ Stubs section for more details.

• The **Node Name** is a unique name automatically assigned to each new node. You can change it to something meaningful. This name is a variable name of the object generated for the node by the C++ code generator.

**Viewing Edge Properties**

Select an edge on the canvas to see the edge properties. This opens the edge properties in the **Edge** tab, as shown below.

![Edge Properties](image)

**Validating a Graph**

After you add nodes to a graph, connect them with edges, and set their properties, you can validate the graph to identify data type mismatches between source and target nodes and highlight other possible issues that may manifest within the described graph topology.

To start a sequence of rule checks on the graph to test various aspects of the graph construction, click the **Graph Rule-Check Analytics** icon () on the toolbar. The results are reported in the **Analytics Report** tab in the **Reports** tab. The following figure shows sample output for a graph.

![Analytics Report](image)

Click a reported diagnostic to highlight the node that needs to be inspected again. In the case of data mismatches, both the source and target nodes for a given edge are highlighted. Review the **Node** tab in the **Properties** pane to address any changes that are needed.

**Saving a Graph to a File**

When a graph is in a consistent state and all the major issues are addressed, save it to a file:

• Save the graph to a GraphML* file by clicking the **Save** icon on the toolbar. GraphML format is an open-standard file format for representing graphs.

  You can return to the graph later time to modify its topology and data types.

  See the Generating C++ Stubs section for details on how to save the graph to C++ code form.
• Save the graph to a PNG image by clicking the Generate Image icon on the toolbar. The image is saved to the same directory where the corresponding GraphML file is saved.

When you load a .graphml file for the first time, the tool computes many pieces of information such as performance metrics and graph layout positions. This information computation can be expensive, and it is recommended that you save the graph after this first load. This information is saved in a .metaxml file and provides caching benefits that enable the tool to exhibit better performance on subsequent loads of the same .graphml file.

**NOTE** You may be asked if you wish to save the .graphml files after load even if no interaction or modification took place. This is due to the tool running the layout algorithm on the graph to display it in an intuitive manner.

### Generating C++ Stubs

To generate stubs for a working C++ application:

1. Create a graph in the canvas as described in the Adding Nodes, Edges, and Ports section.
2. Save the graph as described in the Saving a Graph to a File section.
3. Click the Generate C++ icon on the toolbar to create C++ files.

### Generate C++ Stubs for a Hello World Sample

For example, below is a three-node graph you can use to create a Hello World sample. This graph consists of a source_node followed by two continue_node objects. The first node is named s0 and the next two nodes are named c0 and c1. All nodes have continue_msg objects as their input and/or output types. The body of each node is defined by its C++ Function Object field, as shown below.

To generate C++ stubs from this Hello World sample graph:

1. Create the sample graph by adding a source_node followed by two continue_node objects and connect them with edges. Modify the node names in the Node Properties: name the source_node as s0 and the next two continue_node objects as c0 and c1.
2. Set the following properties to the nodes:
<table>
<thead>
<tr>
<th>Node Name</th>
<th>Input Port Type</th>
<th>Output Port Type</th>
<th>C++ Function Object</th>
</tr>
</thead>
<tbody>
<tr>
<td>s0</td>
<td>None</td>
<td>continue_msg</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>[](continue_msg &amp;c) -&gt; bool { static bool done = false; if (!done) { done = true; return true; } else { return false; } }</td>
</tr>
<tr>
<td>c0</td>
<td>continue_msg</td>
<td>continue_msg</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>[](const continue_msg &amp;m) -&gt; continue_msg { printf(&quot;Hello&quot;); return m; }</td>
</tr>
<tr>
<td>c1</td>
<td>continue_msg</td>
<td>continue_msg</td>
<td></td>
</tr>
</tbody>
</table>
|          |                |                 | [](const continue_msg &m) -> continue_msg { printf(" World! 
"); return m; } |

3. Click the **Save** icon on the toolbar to save this graph as **HelloWorld.graphml**.
4. Click the **Generate C++** icon on the toolbar to generate the C++ stubs.

The generation of the stub files should be reported as successful:
The result of C++ code generation is one file located in the same directory where the GraphML file is last saved. The file name generated is `HelloWorld_stubs.cpp`. It should contain the following code:

```cpp
//
// Automatically generated by Flow Graph Analyzer:
// C++ Code Generator Plugin version XYZ
//
#define TBB_PREVIEW_FLOW_GRAPH_NODES 1
#include "tbb/flow_graph.h"
#include "tbb/tick_count.h"
#include "tbb/atomic.h"
#include <cstdlib>
using namespace std;
using namespace tbb;
using namespace tbb::flow;

size_t key_from_message(char *k) {
    return reinterpret_cast<size_t>(k);
}

template<typename T>
size_t key_from_message(const T &k) {
    return static_cast<size_t>(k);
}

static void spin_for( double weight = 0.0 ) {
    if ( weight > 0.0 ) {
        tick_count t1, t0 = tick_count::now();
        const double weight_multiple = 1e-6;
        const double end_time = weight_multiple * weight;
        do {
            t1 = tick_count::now();
        } while ( (t1-t0).seconds() < end_time );
    }
}

int build_and_run_HelloWorld_g0() {
    graph HelloWorld_g0;

    source_node< continue_msg > s0( HelloWorld_g0,
        [](continue_msg &c) -> bool {
            static bool done = false;
            if (!done) {
                done = true;
                return true;
            } else {
                return false;
            }
        } );
```
In the code above, note the s0, c0, and c1 nodes reflect the properties described in the previous table.

If you have the paths to the Threading Building Blocks (TBB) library set up in your environment, you can build this application from a command prompt:

- **On a Windows* system,** run the following command from a Microsoft Visual Studio* command prompt:

  ```cmd
  cl /EHsc HelloWorld_stubs.cpp tbb.lib
  ```

- **On a Linux* system,** run the following command:

  ```sh
  g++ -std=c++11 HelloWorld_stubs.cpp -ltbb
  ```

In addition, Flow Graph Analyzer allows you to control execution policies for nodes, such as setting lightweight for computational nodes and asynchronous nodes. If you set lightweight policies for any node, the current code generator generates stubs for TBB 2019.

- Lightweight policy is not supported by the TBB 2018 Update 3 or lower. Hence, do not enable the lightweight policy for your nodes if you use a TBB version that does not support it.
- Lightweight policy is supported as a preview feature with TBB 2018 Update 4 or Update 5. However, the generated code must be compiled with the `TBB_PREVIEW_FLOW_GRAPH_LIGHTWEIGHT` macro and linked against the `tbb_preview` library. See [Reducing Scheduler Overhead using Lightweight Policy](#) to learn more about how to set lightweight policy.

See more samples demonstrating this feature in the `samples/code_generation` directory.
**Preferences**

Use the **Preferences** dialog box to set your preferred global values for certain properties.

Go to the **Edit > Preferences** or click on the toolbar. You should see the **Preferences** window:

To set a preference, click a property value and change it. The Flow Graph Analyzer applies your preferences to the entire session and restores the preferences after shutdown and restart.

To get more information about a preference property, hover your mouse over the property to enable a tooltip that contains more information on the property:

Some of the configurable preferences are:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Node Shape</td>
<td>This is a global preference in the <strong>Style</strong> group of the <strong>GUI</strong> tab. Possible values are <strong>box</strong>, <strong>circle</strong>, <strong>basic</strong>, <strong>uml</strong>, and <strong>custom</strong>. The default is the custom node shape.</td>
</tr>
<tr>
<td>GUI Theme</td>
<td>This is a global preference in the <strong>Theme</strong> group of the <strong>GUI</strong> tab. Possible themes are dark and light. The default is dark.</td>
</tr>
<tr>
<td>Property</td>
<td>Description</td>
</tr>
<tr>
<td>----------</td>
<td>-------------</td>
</tr>
<tr>
<td><strong>NOTE</strong></td>
<td>The Flow Graph Analyzer applies a theme change only after restart.</td>
</tr>
<tr>
<td>Disable inter-graph edges</td>
<td>This is a global preference in the GraphML group of the File tab. The Flow Graph Analyzer 2019 Update 4 or higher supports viewing GraphML* files with inter-graph edges. This is disabled by default (set to true).</td>
</tr>
</tbody>
</table>

![Image](image.png)

**NOTE**
Even though cross-graph edges are strongly discouraged in Threading Building Blocks documentation, some use-cases may require them.

<table>
<thead>
<tr>
<th>Fast Hierarchical (preferences group)</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>This is a global preferences group in the File tab. The new fast hierarchical layout algorithm has been introduced in Flow Graph Analyzer 2019 Update 5. It allows you to set the following preferences:</td>
<td></td>
</tr>
<tr>
<td>• Render graphs with semi-accurate edge placement.</td>
<td></td>
</tr>
<tr>
<td>• Render graphs with different node placement with respective to its children such as Median, Average, or Minimum distance.</td>
<td></td>
</tr>
<tr>
<td>• Choose an algorithm type. There are three different algorithms for this layout that can be applied, and each has its own performance characteristics:</td>
<td></td>
</tr>
<tr>
<td>• Simple algorithm type sacrifices accuracy for speed.</td>
<td></td>
</tr>
<tr>
<td>• Hash-based type honors node placement and hash-based depth-first search (DFS) helps with better visual quality.</td>
<td></td>
</tr>
</tbody>
</table>
### Property Description

**Topological Sort Algorithm**

This is a global preference in the **General** group of the **Layout** tab. Choose between two different DFS algorithms for a topological sort of a graph:

- Custom sort algorithm is optimized for topologies that are frequently encountered in TBB and OpenMP*.
- Boost library implementation of DFS.

### Scalability Analysis

When designing a parallel application, you need to know if the application performance continues to increase when you add more threads or tapers off. However, in a complex application, it is often not obvious what the overall parallelism of the application is.

Use the scalability analysis plugin to:

- Estimate the application parallelism provided by the topology of the graph.
- Estimate the inherent application parallelism provided within nodes that have unlimited concurrency.
- Identify contributing factors to overall parallelism.

The scalability analysis plugin allows you to run a graph with a varying number of threads and provides speedup information of the graph with respect to running the graph serially. You can specify any configuration of the graph to help design and analyze the graph.

### Activating the Graph

To run the analysis, the graph must contain at least one **source_node** for activation. The **source_node** pushes the data items through the graph.

The plugin internally ensures that valid data types flow through the graph, so even if a rule check on the data types fails, the scalability analysis still runs. Therefore, selecting and connecting nodes on the canvas should be enough; there is no further need to edit the input and output data types to ensure they match.

**NOTE**

You may need to ensure input or output types for other plugins, like the code generator, to ensure accurate code generation.
Scalability Analysis Prerequisites

Before running the scalability analysis, set the following:

- A concurrency specification
- A data count value
- Node weights

Setting Concurrency Specification

The concurrency specification is a list of the number of threads on which the graph is run. To set the concurrency:

1. Open the Preferences window.
2. Go to the Scalability > TBB category.
3. Set the list in the Concurrency Specification field in the \(1,a-b:n\) format, where:
   - \(1\) is the serial run. Because the speedup is estimated with respect to a serial run of a graph, by default, all graphs are first run serially before running on multiple threads. The addition of 1 is merely symbolic and informational, as its omission has no effect on the list.
   - \(a\) is the starting number of multiple threads.
   - \(b\) is the ending number of multiple threads.
   - \(n\) is the step size.

The default end value is the number of cores on the system and the default step size is a quarter of this value. The end value is always included in the list, even if it is not a direct multiple of the step size.

As an example, on a 32-core system, the default is \(1,8-32:8\). This results in the list \(1,8,16,24,32\), as shown below.

Other supported formats are \(a,b,c\) or only \(a\) if you want to see the speedup for only one set of threads.
You can also combine formats. The following shows some valid concurrency specifications:

- **Range without a step size:** 16-64 yields 1, 16, 32, 48, 64
- **Range with a step size:** 16-64:8 yields 1, 16, 24, 32, 40, 48, 56, 64
- **Single value:** 16 yields 1, 16
- **List of values:** 16, 32, 64 yields 1, 16, 32, 64
- **Mix of range and list:** 16-64, 40, 48, 56 yields 1, 16, 32, 40, 48, 56, 64

**Setting Data Count**

**Data Count** is the number of data items generated and pushed through a graph.

For graphs where potential parallelism is apparent from topology, pushing a single item through the graph is enough to explore the parallelism of the graph.

However, for cases where the parallelism is not readily apparent, as with a node with an unlimited concurrency, pushing a single item through the node is not enough to explore parallelism. Therefore, to ensure the graph is saturated, you need to set data count:

1. Open the **Preferences** windows.
2. Go to the **TBB** category, **Scalability** tab.
3. Change the default number of data items to the number of cores available on the system in the **Data Count** field, based on the topology of the graph and the type of nodes in the graph.

**Setting Node Weight**

The weight determines the simulated amount of time the node spins or does active work per data item passed to the node. The default unit for the weight is microseconds. For example, per the default unit of microseconds, a weight value of 1000 makes a node spin for half a second.

**NOTE**

- If the specified weight is high relatively to the unit, the computation might run for a longer time.
- If the graph has an associated trace, the unit of the weight is overwritten by the unit specified in the trace.

To edit a node weight:

1. Select a node in the canvas.
2. Click the **Node Weight** field value in the **Node** tab of the **Properties** pane and enter your value.

The weight value matters only for nodes with a body, such as **source_node**, **continue_node**, **function_node**, **multifunction_node**, **async_node**, and **tag_matching join_node**. Other nodes that simply assist in the topology of the graph do not use the weight value. For example, specifying a weight value for a **broadcast_node** has no effect on the node.
Consider the following:

- To prevent long runtimes, the scalability analysis scales all runs with total serial times beyond a certain threshold. The current default threshold is 5 seconds. To modify this value:
  1. Go to the Edit > Preferences.
  2. Open the TBB category, Scalability tab.
  3. Modify a value in the Serial Run Time(s) field.

The total serial runtime also considers the number of data items passed through the graph. For example, a graph with a serial runtime of 10 seconds and 4 data items has a total serial runtime of 40 seconds.

- To ensure node weights are not scaled into regions that make the overhead dominant, the analysis uses original weights and does not implement scaling if no node has a weight of 100 microseconds after scaling.

- If a graph contains nodes with performance that could benefit from the use of the TBB lightweight policy, the analysis activates the lightweight policy for the recommended nodes and lists the possible improvement in the results.

Running the Scalability Analysis

After you set the concurrency specification, data count, and node weights, you can run the scalability analysis. To run the analysis, click the Scalability icon on the toolbar:

Exploring the Parallelism in a Concurrent Node

This example explores the parallelism inherent in a node with unlimited concurrency. The node used is a function_node. A source_node is connected to the function_node, as shown below, and eight items are pushed through to the function_node from the source_node. The function_node has a weight of 1s (weight = 1e6). To ensure there are only timing results from the function_node, the source_node has a comparatively negligible weight of 1e-6s (weight=1). The concurrency specification used is 1, 2, 4, 8.

The results are the following:

- For a serial execution with only 1 thread, the total time is 8s as the same thread evaluates the tasks one after the other.
- With 2 threads and 2 overlapping tasks, the total time is 4s.
- With 4 threads and 8 tasks, the total time is 2s.
• With 8 threads and 8 tasks, all tasks overlap, giving a total time of 1s.

**Showing Non-Parallel Nature of a Serial Node**

You can use the Scalability analysis plugin to identify nodes that have no parallelism or to tell when an unlimited node is running serially.

For this example, use a source_node and a function_node connected to it. Set the following:

1. Select the function_node.
2. In the Node Concurrency field, enter 1 or serial. This makes the node run serially.
3. In the Node Weight field, enter 1.
4. Select the source_node.
5. In the Node Weight field, enter 1e-6.
6. Go to the Edit > Properties > TBB.
7. Set the concurrency specification to 1,2,4,8 and click OK.
8. Run the scalability analysis.

You will get the following results:

As expected, time is 8s, regardless of the number of threads used.

You can get the same results if you set the weight of the source_node to 1s and the weight to the function_node to a relatively negligible value. This is because a source_node executes its body serially.

**Exploring the Parallelism Provided by the Topology of a Graph**

This example explores the parallelism provided by the topology of a graph. To make the results as predictable as possible, use a graph that is explicitly parallel, as shown below:
Because the source_node is serial, there is no parallelism provided from within the node. This ensures all parallelism observed is provided by the topology of the graph. Eight source_nodes are connected to a join_node and then to a queue_node. In this graph, only the source_nodes do useful work. Because the parallelism is solely from the topology of the graph, one item per source_node is enough to through the graph. Each source_node has a weight of 1s(1e6). The results of scalability analysis of the graph are shown below.
The speedup is directly proportional to the number of threads.

**Understanding Analysis Color Codes**

The results from the scalability analysis runs are color-coded:

- *Green* dots are results with the number of threads either equal to or less than the number of cores on the system.
- *Blue* dots are results with the number of threads greater than the number of cores.

For example, the following image shows the result for a concurrency specification of $1, 4, 8, 32, 48, 64$ on a 32-core system.

![Graph showing results](image)

Results for runs with less than 32 threads are coded in green, while those with more than 32 threads are coded in blue.

**Collecting Traces from Applications**

This section explains how to collect traces from an application that uses the Threading Building Blocks (TBB) flow graph interfaces.

**Prerequisites**

You need the following to collect traces from an application:

- An application that uses the TBB library flow graph interface
- Threading Building Blocks 4.3 or higher
- The Flow Graph Collector library

Check the links in the Additional Resources section if you are missing any of these prerequisites.

**Simple Sample Application**

This section uses the sample code below as a running example. Assume this code is contained in a file `example.cpp`. You can also use your own application or sample instead of this simple example.

```cpp
#include "tbb/flow_graph.h"
#include <iostream>
using namespace std;
using namespace tbb::flow;
```
int main() {
    graph g;
    continue_node< continue_msg> hello( g,
        []( const continue_msg &)
            { cout << "Hello";
            });
    continue_node< continue_msg> world( g,
        []( const continue_msg &)
            { cout << " World\n";
            });
    make_edge(hello, world);
    hello.try_put(continue_msg());
    g.wait_for_all();
    return 0;
}

Building an Application for Trace Collection

To build an application enabled for trace collection:

1. Define the required marco for your version of the Threading Building Blocks (TBB):
   - For TBB 2019 or higher, define the TBB_USE_THREADING_TOOLS macro and link against the tbb library. This macro activates the required instrumentations in the flow_graph.h header. The tbb library supports flow graph and algorithm profiling. All features other than set_name extensions are available as non-preview features.
   - For TBB lower than 2019, define the TBB_PREVIEW_FLOW_GRAPH_TRACE macro and link against tbb_preview or tbb_preview_debug library. This macro activates the required instrumentation in the flow_graph.h header. The tbb_preview and tbb_preview_debug libraries offer support for preview features not yet supported in the main library. The instrumentation needed by the Flow Graph Analyzer is supported as a Preview feature in TBB 4.3 and higher versions of the libraries.

2. Compile using TBB 4.3 or higher.

Refer to the OS-specific topics for instructions on how to build an application depending on your operating system.

Building an Application on Windows* OS

Building from a Microsoft Visual Studio* Command Prompt

After you open a Visual Studio* command prompt and set up the proper paths for using the Threading Building Blocks (TBB) library, use one of the following command lines to build a Release executable for a running example:

- With Threading Building Blocks (TBB) 2019 or higher:
  
  cl /EHsc /DTBB_USE_THREADING_TOOLS example.cpp tbb.lib

- With TBB version lower than 2019:
  
  cl /EHsc /DTBB_PREVIEW_FLOW_GRAPH_TRACE example.cpp tbb_preview.lib

These command lines define the required macro and link the application against the appropriate tbb library, based on the TBB version you use.
Building from a Microsoft Visual Studio* IDE

To build a Release configuration of your application within a Microsoft Visual Studio* IDE, you must change your project to define the TBB_PREVIEW_FLOW_GRAPH_TRACE/TBB_USE_THREADING_TOOLS macro and link against the tbb_preview.lib/tbb.lib, as shown below for the Microsoft Visual Studio* 2015 IDE based on the TBB version you use.

1. Open the **Project Properties** dialog box, and select **Configuration Properties > C/C++ > Command Line**. In the **Additional Options** textbox, enter one of the following:
   - For TBB versions lower than 2019: /DTBB_PREVIEW_FLOW_GRAPH_TRACE
   - For TBB 2019 or higher: /DTBB_USE_THREADING_TOOLS

2. Select **Configuration Properties > Linker > Input**.

3. In the **Additional Dependencies** field:
   - For a Release build: Enter tbb_preview.lib for TBB versions lower than 2019 or tbb.lib for TBB version 2019 or higher.
   - For a Debug build: Enter tbb_preview_debug.lib for TBB versions lower than 2019 or tbb_debug.lib for TBB version 2019 and higher.
Building an Application on Linux* OS

Use one of the following command lines to build an executable for the running example:

- With Threading Building Blocks (TBB) version 2019 or higher:
  ```
  icpc -std=c++11 -DTBB_USE_THREADING_TOOLS example.cpp -ltbb
  ```

- With TBB version lower than 2019:
  ```
  icpc -std=c++11 -DTBB_PREVIEW_FLOW_GRAPH_TRACE example.cpp -ltbb_preview
  ```

These command lines define the `TBB_USE_THREADING_TOOLS` or `TBB_PREVIEW_FLOW_GRAPH_TRACE` macro and also link the application against the `tbb.lib` or `tbb_preview.lib` library based on the TBB version you use. `-std=c++11` is present because the running example uses lambda expressions, which are a C++11 feature.

To map nodes to source code, use `-g`, `-DTBB_PREVIEW_FLOW_GRAPH_TRACE`, and `-DTBB_USE_THREADING_TOOLS` flags when building the application:

```
icpc -g -std=c++11 -DTBB_USE_THREADING_TOOLS -DTBB_PREVIEW_FLOW_GRAPH_TRACE example.cpp -ltbb_preview
```

Building an Application on macOS*

Use one of the following command lines to build an executable for the running example:

- With Threading Building Block (TBB) 2019 or higher:
  ```
  clang++ -std=c++11 -DTBB_USE_THREADING_TOOLS example.cpp -ltbb
  ```

- With TBB version lower than 2019:
  ```
  clang++ -std=c++11 -DTBB_PREVIEW_FLOW_GRAPH_TRACE example.cpp -ltbb_preview
  ```

These command lines define the `TBB_USE_THREADING_TOOLS` or `TBB_PREVIEW_FLOW_GRAPH_TRACE` macro and also link the application against the `tbb.lib` or `tbb_preview.lib` library based on the TBB version you use. `-std=c++11` is present because the running example uses lambda expressions, which are a C++11 feature.
Collecting Trace Files

While executing, your application collects trace files only if there is an appropriate collector library at the locations specified by the `INTEL_LIBITTNOTIFY32` or `INTEL_LIBITTNOTIFY64` environment variables. The `INTEL_LIBITTNOTIFY32` path is searched by 32-bit executables and the `INTEL_LIBITTNOTIFY64` path is searched by 64-bit executables.

To collect and convert trace files to use them with the Flow Graph Analyzer, chose one of the following options:

- Collect traces by starting your application from the Flow Graph Analyzer GUI.
- Set up your environment so a trace is collected when the application is started outside of the GUI.

**NOTE**
Both approaches require you to build the application following the steps from Building an Application section for your operating system.

Collecting Traces Inside the Flow Graph Analyzer GUI

To run an existing Threading Building Blocks (TBB) application and collect execution trace information for analysis, you can launch the TBB trace collector feature from the Flow Graph Analyzer GUI as follows, assuming the paths for the Threading Building Blocks (TBB) libraries are set up (see Release Notes and Known Issues for limitations):

1. Run the trace collector using one of the following options:
   - Go to the **Offload Actions > Run and collect traces** menu option.
   - Click the **Run and Collect Traces** icon on the toolbar.

The collector **Preferences** window opens, which lets you specify the application to run:
2. In the collector **Preferences** window:
   - Specify an application to run in the **Application** field.
   - Optional: Change the **Working Directory** value if required. The default is the Flow Graph Analyzer directory.
   - View and set other environment variables, including **INTEL_LIBITTNOTIFY32** and **INTEL_LIBITTNOTIFY64**, using the **Environment Variables** pane of the dialog box.

The environment variables for running trace collection have default settings if you did not change their values in the environment from which the Flow Graph Analyzer is launched. The inherited values are used if the environment variables are set in the environment.

<table>
<thead>
<tr>
<th>OS</th>
<th>Environment Variable</th>
<th>Default Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Windows*</td>
<td>INTEL_LIBITTNOTIFY32</td>
<td>..\fgt\windows\bin \ia32\vc14.1\fgt.dll</td>
<td>32-bit collector library</td>
</tr>
<tr>
<td></td>
<td>INTEL_LIBITTNOTIFY64</td>
<td>..\fgt\windows\bin \intel64\vc14.1\fgt.dll</td>
<td>64-bit collector library</td>
</tr>
<tr>
<td></td>
<td>INTEL_ITTNOTIFY_GROUPS</td>
<td>all;</td>
<td>Trace events from all groups</td>
</tr>
</tbody>
</table>
### OS Environment Variable Default Value Description

<table>
<thead>
<tr>
<th>OS</th>
<th>Environment Variable</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Linux*</td>
<td>INTEL_LIBITNOTIFY</td>
<td>../fgt/linux/lib/ia32/cc4.8_libc2.19_kernel3.13.0/libfgt.so</td>
<td>32-bit collector library</td>
</tr>
<tr>
<td></td>
<td>32</td>
<td>../fgt/linux/lib/intel64/cc4.8_libc2.19_kernel3.13.0/libfgt.so</td>
<td>64-bit collector library</td>
</tr>
<tr>
<td></td>
<td>INTEL_ITTNOTIFY_GR</td>
<td>all;</td>
<td>Trace events from all groups</td>
</tr>
<tr>
<td></td>
<td>OUPS</td>
<td></td>
<td></td>
</tr>
<tr>
<td>macOS*</td>
<td>INTEL_LIBITNOTIFY</td>
<td>../fgt/macos/lib/ia32/osx10.12.6_kernell6.7.0/libfgt.dylib</td>
<td>32-bit collector library</td>
</tr>
<tr>
<td></td>
<td>32</td>
<td>../fgt/macos/lib/intel64/osx10.12.6_kernell6.7.0/libfgt.dylib</td>
<td>64-bit collector library</td>
</tr>
<tr>
<td></td>
<td>INTEL_ITTNOTIFY_GR</td>
<td>all;</td>
<td>Trace events from all groups</td>
</tr>
<tr>
<td></td>
<td>OUPS</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

3. Click the **Accept Run collection** button. If the application is executed correctly, the trace files are converted to a GraphML* format. The output file is stored in the working directory with a name based on the executable name and the time of the trace collection run.

4. To examine the trace file, load the GraphML* file into the Flow Graph Analyzer GUI manually.

### Collecting Traces Outside the Flow Graph Analyzer GUI

To collect traces outside the Flow Graph Analyzer GUI:

- If you launch your application from a Windows* command prompt or a Linux* terminal, the simplest approach is using the `fgtrun` script to run your application.
- If you cannot launch your application from a prompt or terminal or you want to launch it from within an IDE, you must manually perform the steps performed by the `fgtrun` script.

In either case, you must update your **PATH** environment variable to add the paths to essential tools, as described below, and set the **FGT_ROOT** variable.

This section assumes the full path to your Flow Graph Analyzer installation is `<advisor-install-dir>\fga`. The version of your Visual Studio* compiler is `<vc_version>` with possible values of `vc12`, `vc14`, and `vc14.1`.

<table>
<thead>
<tr>
<th>OS</th>
<th>Environment Variable</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Windows*</td>
<td>FGT_ROOT</td>
<td>&lt;advisor-install-dir&gt;\fga\fgt</td>
<td>The path to the Flow Graph Collector installation</td>
</tr>
</tbody>
</table>
Collecting Trace Files with fgtrun Script

Use the fgtrun script to collect the trace information from your application.

Before running the script, you must set the FGT_ROOT variable to <advisor-install-dir>/fga/fgt as described in the Collecting Traces Outside the Flow Graph Analyzer GUI. The fgtrun script sets the paths necessary to execute your application and generate the GraphML* and TraceML* files that can be loaded into the Flow Graph Analyzer for visualization.

The following table lists the directories in which the scripts are located on a given system.

<table>
<thead>
<tr>
<th>Operating System</th>
<th>Version</th>
<th>Location</th>
<th>Example Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>Windows*</td>
<td>fgtrun.bat</td>
<td>%FGT_ROOT%\windows \bin;%FGT_ROOT% \windows\bin \ia32&lt;vc version&gt;; %FGT_ROOT%\windows \bin\intel64&lt;vc version&gt;; %PATH%</td>
<td>fgtrun.bat &lt;app-binary-name&gt; [&lt;binary-args&gt;] [---]</td>
</tr>
<tr>
<td>Linux*</td>
<td>FGT_ROOT</td>
<td>&lt;advisor-install-dir&gt;/fga/fgt</td>
<td>The path to the Flow Graph Collector installation</td>
</tr>
<tr>
<td></td>
<td>PATH</td>
<td>${FGT_ROOT}/linux/bin:${FGT_ROOT}/ linux/bin/ia32/ cc4.8_libc2.19_kern el3.13.0:${FGT_ROOT}/ linux/bin/intel64/ cc4.8_libc2.19_kern el3.13.0:${PATH}</td>
<td>The path must include paths to fgtrun.sh, fgtrun.csh, and fgtxml.</td>
</tr>
<tr>
<td>macOS*</td>
<td>FGT_ROOT</td>
<td>&lt;advisor-install-dir&gt;/fga/fgt</td>
<td>The path to the Flow Graph Collector installation</td>
</tr>
<tr>
<td></td>
<td>PATH</td>
<td>${FGT_ROOT}/macos/bin:${FGT_ROOT}/ macos/bin/ia32/ osx10.12.6_kernel16.7.0:${FGT_ROOT}/ macos/bin/intel64/ osx10.12.6_kernel16.7.0:${PATH}</td>
<td>The path must include paths to fgtrun.sh, fgtrun.csh, and fgtxml.</td>
</tr>
<tr>
<td>Operating System</td>
<td>Version</td>
<td>Location</td>
<td>Example Use</td>
</tr>
<tr>
<td>------------------</td>
<td>---------</td>
<td>----------</td>
<td>-------------</td>
</tr>
<tr>
<td>Linux*</td>
<td>fgtrun.sh</td>
<td>${FGT_ROOT}/linux/bin</td>
<td>fgtrun.sh &lt;app-binary-name&gt; [binary-args] [-ia32/ --intel64] [-omp] [-xml]</td>
</tr>
<tr>
<td>macOS*</td>
<td>fgtrun.sh</td>
<td>${FGT_ROOT}/macos/bin</td>
<td>fgtrun.sh &lt;app-binary-name&gt; [binary-args] [-ia32/ --intel64] [-omp] [-xml]</td>
</tr>
</tbody>
</table>

The `fgtrun` script tries to automatically detect the architecture and C/C++ runtime version (Windows* OS only) of the executable used to collect the traces and requires the presence of helper tools. If the helper tools are not available or fail to identify the required information, `fgtrun` scripts sets default values and runs the collection. Optionally you can override these default values by setting architecture and C/C++ runtime version information (Windows* OS only) using command line arguments when the script is invoked.

The `fgtrun` has the following options:

---

**--omp**

Enable OpenMP* trace collection for applications linked with `-qopenmp`.

The OpenMP* runtime environment must be set correctly before the script is launched.

**NOTE** This OpenMP* trace collection capability is currently not supported on the Windows* OS.

---

**--xml**

Collect traces in XML format.

By default, the collector generates binary traces. If trace collection fails, you can switch to XML trace generation mode to debug the cause of the failure.

---

**--sym**

Get mapping between nodes and source code.

**NOTE** The symbol resolution feature is currently only supported on Linux* OS.

---

### Collecting Trace Files without fgtrun Script

You can choose to collect traces without the `fgtrun` script if you do not want to launch your application from a Visual Studio* command prompt or Linux* terminal. In this case, follow the steps below to collect and convert trace files manually.
These steps do not outline steps required to capture symbol resolution information.

1. Set paths to the collector libraries.
   
   Set up or update the environment variables shown below. For Windows* systems, specify also the proper version of the Microsoft Visual Studio* compiler (vc12, vc14, or vc14.1).

<table>
<thead>
<tr>
<th>Operating System</th>
<th>Environment Variable</th>
<th>Default Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Windows*</td>
<td>INTEL_LIBITTNOTIFY 32</td>
<td>..\fgt\windows\bin\ia32\vc version&gt;\fgt.dll</td>
<td>32-bit collector library</td>
</tr>
<tr>
<td></td>
<td>INTEL_LIBITTNOTIFY 64</td>
<td>..\fgt\windows\bin\intel64\vc version&gt;\fgt.dll</td>
<td>64-bit collector library</td>
</tr>
<tr>
<td></td>
<td>INTEL_ITTNOTIFY_GROUPS</td>
<td>all;</td>
<td>Trace events from all groups</td>
</tr>
<tr>
<td>Linux*</td>
<td>INTEL_LIBITTNOTIFY 32</td>
<td>../fgt/linux/lib/ia32/cc4.8_libc2.19_kernel3.13.0/libfgt.so</td>
<td>32-bit collector library</td>
</tr>
<tr>
<td></td>
<td>INTEL_LIBITTNOTIFY 64</td>
<td>../fgt/linux/lib/intel64/cc4.8_libc2.19_kernel3.13.0/libfgt.so</td>
<td>64-bit collector library</td>
</tr>
<tr>
<td></td>
<td>INTEL_ITTNOTIFY_GROUPS</td>
<td>all;</td>
<td>Trace events from all groups</td>
</tr>
<tr>
<td>macOS*</td>
<td>INTEL_LIBITTNOTIFY 32</td>
<td>../fgt/macos/lib/ia32/osx10.12.6_kernel16.7.0/libfgt.dylib</td>
<td>32-bit collector library</td>
</tr>
<tr>
<td></td>
<td>INTEL_LIBITTNOTIFY 64</td>
<td>../fgt/macos/lib/intel64/osx10.12.6_kernel16.7.0/libfgt.dylib</td>
<td>64-bit collector library</td>
</tr>
<tr>
<td></td>
<td>INTEL_ITTNOTIFY_GROUPS</td>
<td>all;</td>
<td>Trace events from all groups</td>
</tr>
</tbody>
</table>

If you want the Microsoft Visual Studio* IDE to use the environment variables set in a Microsoft Visual Studio command prompt, you can launch the Visual Studio* IDE from the command prompt using the following command:

`devenv /useenv`

2. Run the application.
If your paths are set up correctly, the application generates one or more files that start with \_fgt. There is one file per thread that participates in executing the parallelism in the application. So, for example, if two threads participate in the execution of the flow graph, your run generates two files: \_fgt.0 and \_fgt.1, with an autogenerated folder in the format \_fga_YYYYMMDD_HHMMSS according to its creation (for example, 20191111_1111).

3. Convert the trace files to GraphML\* and TraceML\* format.

Convert the \_fgt binary files to the XML format understood by the Flow Graph Analyzer tool using the \texttt{fgt2xml.exe} converter in the directory containing the folder with the trace files:

\texttt{fgt2xml.exe }<\texttt{desired}\_\texttt{name}>

This converter scans the current directory for all \_fgt files within the most recent folder according to its name and generates two output files: desired\_name.graphml and desired\_name.traceml.

**Nested Parallelism in Flow Graph Analyzer**

The Flow Graph Analyzer supports visualization of applications that contain multiple levels of parallelism, such as nested Threading Building Blocks (TBB) algorithms and OpenMP* parallel regions. This feature requires additional support from the parallel runtime libraries and can be used in combination with a TBB flow graph.

The sample code below is an example of nested parallelism that combines a TBB flow graph, a TBB parallel\_for algorithm, and an OpenMP* parallel region.

```c++
#include "tbb/tbb.h"
#include "tbb/flow_graph.h"
#include <omp.h>
#include <iostream>
using namespace tbb;
using namespace tbb::flow;
int main() {
    graph g;
    const int size = 20;
    continue_node< continue_msg> hello( g,
        []( const continue_msg &)
        { std::cout << "Hello\n";
            tbb::parallel_for(0, size, 1, [=](int k) {
                std::cout << k << "\n";
            });
        });
    continue_node< continue_msg> world( g,
        []( const continue_msg &)
        { std::cout << " World\n";
            #pragma omp parallel for
            for (int i=0; i<20; i++) {
                std::cout << i <<"\n";
            }
        });
    make_edge(hello, world);
    hello.try_put(continue_msg());
    g.wait_for_all();
    return 0;
}
```
Tracing Nested Threading Building Blocks (TBB) Algorithms

Threading Building Blocks (TBB) 2019 enables general tracing of parallel algorithms, which is enabled by default and activated by the Flow Graph Analyzer trace collector.

As a result, Flow Graph Analyzer can display TBB library activity in nested and non-nested algorithms. Therefore, task context switches are captured and can be visualized in the Flow Graph Analyzer GUI. This work is similar to tasks in the timeline and is named according to its algorithm (for example, parallel_for).

NOTE
This information might not be available for user-defined task groups.

Tracing Nested OpenMP* Algorithms

For detailed information on Flow Graph Analyzer support for OpenMP* technology, see Experimental Support for OpenMP* Applications.

Analyzer Workflow

NOTE
This section describes a recommended workflow to identify performance issues in the executed graph. This workflow may change as more analytics plugins are added. However, the fundamental principle should not change, as the goal is to maximize the throughput of the graph in a streaming case, and provide the best scaling performance with respect to the serial run.

The Flow Graph Analyzer provides the following capabilities for analyzing flow graph performance:

- Display the graph for which the execution trace is captured. See the Preferences section for details on how to enable loading .graphml files that contain graphs with cross-graph edges.
- Display the trace information and highlight parallel performance issues.
- Map poorly scaling time regions to nodes executing at that time.
- Compute the critical path of the graph.
- View compute statistics for the computational nodes based on the execution traces.
- View prioritized diagnostics.

Follow the steps in this section to analyze performance.

Finding Time Regions of Low Concurrency and Their Cause

1. Run the trace collector.
2. In the Execution Trace Views tab, inspect the node concurrency histogram for regions in red, which indicate low concurrency.

3. Zoom in to a red region to inspect the data at a higher resolution and provide a better idea as to how concurrency varies over time.
4. Select a point in the chart where the concurrency is low to highlight the relevant node(s). Hover the mouse over the highlighted node to identify the node name.

Because the analysis tool does not have built-in symbol resolution, the determination of the C++ class of the body executed by a node must be explicitly encoded into the application. Explicit encoding affects the Name and/or object_name fields in the Node Properties tab. For example:

```cpp
#include <tbb/flow.hpp>

tbb::flow::graph g;
...
tbb::flow::source_node<int> s_node (g, source_node_body(),
false);
#if TBB_PREVIEW_FLOW_GRAPH_TRACE
s_node.set_name("My Source Node");
#endif
```

This coding enables node annotation with the specified string during trace collection, and the annotation appears when you hover the mouse over the node.

**NOTE**
The set_name functions are only available when the TBB_PREVIEW_FLOW_GRAPH_TRACE macro is defined at compile-time.

**Finding a Critical Path**

The critical path from a node N to a node M is the longest (in time) path from N to M. Its length is a lower-bound on the execution time of a computation that begins at N and ends at M.

The Critical Path Analytic in the Flow Graph Analyzer computes a critical path for each source node/sink node pair. A given graph has as many critical paths as the product of the number of source nodes and the number of sink nodes. Source nodes are the nodes in the graph without any predecessors, or nodes with an in-degree of zero. Sink nodes are the nodes in the graph without any successors, or nodes with an out-degree of zero.

Click the Compute Critical Path icon on the toolbar to calculate the critical paths in the graph. These critical paths are displayed in descending order by cost. Inspect the topmost critical path first because, as the longest critical path, it sets the lower bound on the execution time for the whole graph.

The screenshot below shows a sample critical path report.
Selecting a critical path in the **Analytics Report** window highlights all the nodes on the critical path.

**Finding Tasks with Small Durations**

Tasks executed by a flow graph are spawned as Threading Building Blocks (TBB) tasks, so the task duration must be large enough to amortize the cost of a task spawn.

**Finding Tasks with Small Durations using Statistics**

1. Open the **Statistics** tab in the bottom pane.
2. Open the **Graph** tab to see execution time metrics. The metrics are computed as the mean and standard deviation for each node based on the execution traces.
3. Sort the resulting data by the **Avg Task Duration** column to identify the nodes with the smallest average durations.

Any node with an average duration of a few microseconds requires that you additionally inspect it, because any concurrency gained by its parallel execution may be overwhelmed by its scheduling costs.

The screenshot above shows a sample graph that executed 250 times. Notice the **Count** column, which is the number of times a node executes, is 250 for all functional nodes. Inspect the **Avg Task Duration** column for nodes that execute in a few microseconds.

**NOTE** Times are provided in milliseconds.

**Finding Tasks with Small Durations using Treemap**

Another way to visualize task durations and the average concurrencies of each node is the Treemap view in the **Analysis Mode** tab. If multiple graphs are present in the application run, you see a high-level treemap showing the health of all the graphs in the run.

The Treemap view organizes the nodes in the graph by node durations. The larger the area of the square, the more time the node spent on the CPU. The node color is determined by the average observed concurrency of the graph while the node was running, and the colors use the same scale as in the active thread chart. A red node indicates poor concurrency when the node is executed on the system.

Hover a mouse over the Treemap to see the details about nodes in the graph. Double-click the graph in the Treemap view to keep the node treemap visible.
When a graph has nested subgraphs, the treemap presents this information by embedding the nested subgraph in the node which spawned the subgraph. This is visually represented by increasing the width of the border surrounding each node that contain subgraphs.

- Hover the mouse over a node that has embedded subgraphs to see the hidden subgraph.
- Double-click the node to zoom to the child-node level and see the contents of the embedded subgraph as a treemap.

Click any node in the Treemap view to highlight that node in the graph view on the canvas. If you select the default zoom factor or reset the zoom factor, clicking a node in the Treemap view zooms in and centers on the node in the graph view. The smaller the node size, the smaller the tasks executed by that node.

The Treemap view supports three different layouts for visualizing the treemap: squarified layout, alternating layout, and snake layout. All three layouts use:

- The node CPU time to determine the size of each node in the treemap
- The average concurrency observed in the graph while a node was active to determine the color

To switch between layouts:

1. Open the Preferences window.
2. Go to the GUI > General.
3. Change the value of the Default Treemap Rendering option in the Analysis View group.

**Reducing Scheduler Overhead using Lightweight Policy**

The Flow Graph API allows you to apply lightweight policy for computational nodes such as function node, multifunction node, continue node, and async node. Enabling the lightweight policy helps reduce scheduling overhead. It can limit parallel execution of tasks, so apply this policy on a per-node basis after careful evaluation.

The lightweight policy indicates that the body of a node contains a small amount of work and, if possible, should be executed without the overhead of scheduling a task. By default, the async node has the lightweight policy enabled because it has small computation weight. All other computational nodes do not have the lightweight policy enabled when they are dragged and dropped into the canvas.
Use the lightweight policy in the following cases:

- Node weight is less than 1 microsecond when no trace information is available.
- Node average time is less than 1 microsecond if the graph is loaded into context with trace data.

When validating a graph, the graph rule check automatically identifies nodes that can use the lightweight policy. If the above conditions are not met but the lightweight policy is set, the graph rule check recommends removing the lightweight policy for the corresponding node.

To display recommendations for applying the lightweight policy:

1. Click the graph rule check icon on the toolbar to run the check.
2. Go to the Analytics Report tab to see the results.
3. Based on the results, set or disable the lightweight policy for certain nodes.

- To set or disable the lightweight execution policy for a single node:
  1. Click the node to display the node properties on the right pane.
  2. Set the Execution policy property to none to disable lightweight policy or to lightweight to enable lightweight policy.
- To set or disable the lightweight execution policy for all the nodes listed by graph rule check:
  1. Multi-select the report lines that say Consider enabling lightweight policy for small computational or async node or Consider disabling lightweight policy for small computational or async node to highlight all nodes in the canvas and display the common properties for all the selected nodes.
  2. Set the Execution policy property to none to disable lightweight policy or to lightweight to enable lightweight policy.

For example, to enable the lightweight policy for multiple nodes:

Another important attribute related to node execution policy is a buffer policy:

- If you set the buffer policy to queueing when lightweight policy is enabled, the Flow Graph Analyzer adds queueing_lightweight to the policy parameter of the node declaration during C++ code generation.
- If you set buffer policy to rejecting, the Flow Graph Analyzer adds rejecting_lightweight to the policy parameter of the node declaration during C++ code generation.

For example, by default, the async node is set to queueing_lightweight, and the Flow Graph Analyzer does not add any policy during code generation for async node. Setting buffer policy to rejecting for the async node adds rejecting_lightweight to the policy parameter of async node declaration during code generations.
Identifying Tasks that Operate on Common Input

Threading Building Blocks (TBB) 2018 Update 5 introduced a new preview feature that adds additional meta information to trace data, such as a frame number that a current task is working on. You can use this metadata to track different pipeline stages in execution, for example to identify an unbalanced pipeline. The following demonstrates how to use the user event tracing interface to enable the Flow Graph Analyzer color-by-data feature.

Highlighted in bold is code that enables the Flow Graph Analyzer to add a unique ID (frame ID) to group tasks.

```cpp
#include "tbb/flow_graph.h"
#include "tbb/tbb_profiling.h"
#include <string>
#include <vector>

int main() {
    tbb::flow::graph g;
    const int max_frames = 20;
    std::vector<tbb::profiling::event*> e;
    for(int i=0; i<nbr_of_frames;++i)
        e.push_back(new tbb::profiling::event(std::to_string(i)));
    tbb::flow::source_node<int> source( g,
        [&] (int &v) -> bool {
            static int i = 0;
            if( i < max_frames ) {
                e[i]->emit(); v = i++;
                return true;
            }
            return false;
        }, false);
    tbb::flow::function_node<int> foo( g, tbb::flow::unlimited,
        [](const int &input1) -> int {
            tbb::profiling::event::emit(std::to_string(input1));
            return input1;
        });
    tbb::flow::function_node<int> bar( g, tbb::flow::unlimited,TBB
        [](const int &input1) -> int {
            tbb::profiling::event::emit(std::to_string(input1));
            return input1;
        });
    make_edge(source, foo);
    make_edge(source, bar);
    source.activate();
    g.wait_for_all();
    return 0;
}
```

NOTE
Compilation of the above code differs based on the TBB version you use:

- For TBB version 2019 or higher, compile the code with the TBB_USE_THREADING_TOOLS macro and link against the tbb library.
- For TBB version 2018 Update 5, compile the code with the TBB_PREVIEW_FLOW_GRAPH_TRACE and TBB_USE_THREADING_TOOLS macros and link against the tbb_preview library.

To enable the Flow Graph Analyzer color-by-data feature:
1. Assign IDs to events doing one of the following:
   - Option 1:
     1. Create an event object or a collection of events upfront, where the only argument is a string (data ID) that identifies the event.
     2. Call the emit function of the object to tag a task with a data ID.
   - Option 2: Call a static function inline (inside a task body).
2. In the Execution Trace Views tab in the bottom pane, choose Color By Data from the drop-down list.

Result: The Flow Graph Analyzer groups tasks that share the same data ID and displays them in a common color:

Support for Data Parallel C++ Applications

Flow Graph Analyzer is a feature of Intel® Advisor that allows you to explore, debug, and analyze graph computation problems. Since the DPC++ runtime constructs an asynchronous task graph from submitted work, Flow Graph Analyzer allows you to visualize and interact with the asynchronous task graph, and its execution traces. The tool introduces the following features:

- For a CPU device: Execution trace-based analytics.
- For CPU and GPU devices: Graph-related analytics.

NOTE

The data collection support for DPC++ applications is currently supported only on Linux* OS.

The code sample below illustrates a simple example of a DPC++ application that adds two vectors. The subsequent sections will use it as an example.

```cpp
#include <CL/sycl.hpp>
#include <iostream>
#define VECTOR_SIZE 16384
using namespace cl::sycl;
```
void vec_add(queue &q, const float A[], const float B[], float C[],
const int size) {
  // Create the buffers
  buffer<float, 1> bufA(A, range<1>(VECTOR_SIZE));
  buffer<float, 1> bufB(B, range<1>(VECTOR_SIZE));
  buffer<float, 1> bufC(C, range<1>(VECTOR_SIZE));

  q.submit([&](handler &cgh) {
    auto Acc = bufA.get_access<access::mode::read>(cgh);
    auto Bcc = bufB.get_access<access::mode::read>(cgh);
    auto Ccc = bufC.get_access<access::mode::write>(cgh);
    cgh.parallel_for<class saxpy_kernel>(range<1>(size), [=](id<1> idx) {
      Ccc[idx[0]] = Acc[idx[0]] + Bcc[idx[0]];
    });
  });
}

int main(int argc, char **argv) {
  if (argc < 2) {
    std::cout << "Usage:- " << argv[0] << " [cpu, gpu]\n"
              << "[cpu, gpu]\n"
              << return 1;
  }
  if (argc < 2) {
    std::cout << "Usage:- " << argv[0] << " [cpu, gpu]\n"
              << return 1;
  }
  float A[VECTOR_SIZE], B[VECTOR_SIZE], C[VECTOR_SIZE];

  if (std::string("cpu") == argv[1]) {
    cpu_selector device;
    queue q(device);
    vec_add(q, A, B, C, VECTOR_SIZE);
  } else if (std::string("gpu") == argv[1]) {
    gpu_selector device;
    queue q(device);
    vec_add(q, A, B, C, VECTOR_SIZE);
  }

  return 0;
}

Collect DPC++ Application Traces

**NOTE** DPC++ trace collection is currently supported only on Linux* OS.

The command line collector for Flow Graph Analyzer enables you to capture trace data from Data Parallel C++ (DPC++) applications. To collect DPC++ traces, you will need trace-enabled DPC++ run-times. Use the code from Analyze Data Parallel C++ Application as a sample application. To build it:

1. Copy the code snippet from Analyze Data Parallel C++ Application and save it as va_const.cpp.
2. Run the following command to build it:

   dpcpp -o vac ./va_const.cpp
To collect traces for the DPC++ application using the built sample and create the XML files to view with Flow Graph Analyzer:

1. Set the environment variable `FGT_ROOT` to point to `<fga-install-dir>/fgt`:
   ```
   export FGT_ROOT=<advisor-install-dir>/fga/fgt
   ```

2. Set the back end for the DPC++ run-time to OpenCL™ by setting the following environment variable:
   ```
   export SYCL_BE=PI_OPENCL
   ```

   **NOTE** Current version of Flow Graph Collector does not support Level0.

3. Run the application using the Flow Graph Analyzer collector:
   ```
   $FGT_ROOT/linux/bin/fgtrun.sh ./vac
   ```

   This command will generate two files: `vac.graphml`, which contains the semantic information of what was executed, as in the asynchronous task-graph, and `vac.traceml`, which contains the execution traces of the application. Open the files in the Flow Graph Analyzer on the current system or copy them to another system with Flow Graph Analyzer installed to investigate.

To launch the Flow Graph Analyzer graphical user interface, use the following command:

```
<fga-install-dir>/fga/run_fga.sh &
```

**Examine a DPC++ Application Graph**

The visualization of Data Parallels C++ (DPC++) applications is similar to data from other run-times, such as Threading Building Blocks (TBB) or OpenMP®. The graph in DPC++ represents the asynchronous task graph created from the end-user construct such as buffer accessors, command group handler, and data parallel constructs such as `parallel_for`.

The data from the sample viewed in Flow Graph Analyzer is shown above. As with other runtimes, the graph view is correlated with the execution trace views. The workflows will provide similar information for DPC++.

To better visualize the overlapping tasks in the execution trace view, select the **Stacked View** attribute from the pull-down menu as shown below:
This will change the view to an icicle chart that displays everything in detail, and you can see the calls to the OpenCL™ stack.

Clicking on a task in the timeline views will highlight a node in the graph if that task belongs to the graph. If you want to highlight all the tasks that belong to a graph node, you should enable task highlighting button and select a node on the graph to see the associations.

The screenshot below shows the tasks that belong to the memory transfer node are highlighted in a different color. Using the correlation features, one can debug the execution profile of the application to get a better understanding of the execution. Flow Graph Analyzer also includes features that target specific performance-related issues and the other sections go into detail for each one of these potential performance problems.
Hotspot View

Flow Graph Analyzer supports hot-spot views, but the data is limited to the objects in the graph. The data collection for Flow Graph Analyzer-based collectors is currently limited to time-based traces. To obtain the hotspot view, select the Statistics tab, and click the Graph pane for the data.

View Performance Inefficiencies of Data-parallel Constructs

The Statistics Tab also contains the efficiency information for each parallel construct if they are employed by the algorithm. This data will show up under the Parallel Efficiency tab in the Statistics group.
The data parallel construct efficiency for each instance of a kernel. The column provides information that is useful for understanding the execution, and makes inferences to improve performance.

- The parallel algorithms are nested under the kernel name when the kernel name can be demangled correctly.
- The Efficiency column indicates the efficiency of the algorithm, when associated with the algorithm name. For the participating worker threads, the efficiency column indicates the efficiency of the thread while participating in the execution. This data is typically derived from the total time spent on the parallel construct and the time the thread spent participating in other parallel constructs.
- Task Count column indicates the number of tasks executed by the participating thread.
- Duration indicates the time the participating thread spends executing tasks from the parallel construct.
- CPU time is the Duration column data expressed as a percentage of the wall clock time of the parallel construct.
- Other Time will be 0 if the thread fully participates in the execution of tasks from the parallel construct. However, in runtimes such as Threading Building Block (TBB), the participating threads may steal tasks from other parallel constructs submitted to the device to provide better dynamic load balancing and throughput. In such cases, the Other Time column will indicate the percentage of the total wall clock time the participating thread spends executing tasks from other parallel constructs.
- Fork Imbalance indicates the penalty for waking up threads to participate in the execution of tasks from the parallel construct. For more information, see Startup Penalty.
- Join Imbalance indicates the degree of imbalanced execution of tasks from the parallel constructs by the participating worker threads. For more information, see Data Parallel Efficiency.

**Find Issues Using Static Rule-check Engine**

The Rule-check engine in Flow Graph Analyzer includes DPC++ specific analysis that can be invoked by clicking on the Rule-check button in the toolbar. Run it to check for issues.
NOTE In the sample code, the kernel name demangling has an issue. This is the current limitation of the demangling mechanism available in the runtime. For correct demangling of kernel names, use the open-source version of the Clang++ compiler.

The Rule-Check Engine currently identifies the following types of issues that may be present in an application:

- Use of const reference to a host pointer to initialize a buffer
- Use of host pointer accessor in a loop
- Data parallel construct inefficiency

**Issue: Const Reference to a Host Pointer Used to Initialize a Buffer**

When porting applications from C++ to DPC++, these issues may be implicitly present. The convention for passing arguments to a function is defined in C++. If a function requires read-only parameters, they are sent in as const references and any parameter that has to be used for read-write will be without a const. This causes a secondary issue in DPC++ algorithms if these const references are used by the algorithm to construct buffers. Since the type of access required to use this data is not known at the time of construction of the buffer, the Intel® oneAPI DPC++/C++ Compiler is conservative and creates copies of const references while creating the buffers. If these are large arrays, the cost incurred is not trivial. This issue only affects the CPU device as everything is communicated through shared memory and the copying of data pointer to the host pointer is not necessary.

The va_const.cpp example demonstrates such an issue and indicates buffer copy in the application that needs to be looked at.
To eliminate this copy, you should change the code sample in the following way:

```c++
// Old code:
// The function prototype passed in the read-only buffers as const, which is the
// recommended practice in C++
//
// void vec_add(queue &q, const float A[], const float B[], ...) {
void vec_add(queue &q, float A[], float B[], float C[],
             const int size) {

    ...
}
```

**NOTE** The recommended practice in C++ is to pass in read-only parameters as const values. However, this causes the Intel® oneAPI DPC++/C++ Compiler to be conservative and create a copy. If you are porting C++ code to DPC++, the static rule-check should help you identify such issues in your application.

**Issue: Host Pointer Accessor Used in a Loop**

In many algorithms, it is likely that a lot of operations are performed on the device memory and some operations on the host memory. This is particularly true in simulation code where the host memory is updated using a host accessor. This causes many things to happen within the DPC++ run-time where the locks the buffer the accessor points to and updates the copy of this buffer memory on the host device. This pattern of access could cause a lot of memory copies from the device to the host and back in order to keep the data coherent.

Flow Graph Analyzer reports such issues in the following way:

Click the issue to highlight the loop in the graph that consists of a host pointer accessor.

If the buffer pointed by the host pointer accessor is large, the costs incurred due to this access can be a significant portion of each loop.

**Issue: Data Parallel Construct Inefficiency**

The DPC++ language allows to use data parallel constructs within each command group. This feature of rule-check analysis tries to capture the efficiency of a data parallel construct. The inefficiencies in the data parallel construct are broken down into two parts:

- Startup costs for kicking off the data parallel algorithm on the worker threads.
- Imbalance costs encountered during the execution of the algorithm.
The combination of these parts affects the overall efficiency of the data parallel algorithm. The data and screenshots shown in this section are from the Nbody sample that is available with Intel® oneAPI Toolkits (Beta).

Startup Penalty
The startup costs are primarily related to the worker threads participating in the parallel algorithm starting up slowly. If your application exhibits a lot of inefficiencies due to startup costs, the Linux* kernel maybe biased towards power. You can use the following command to ensure that it is set to performance:

```
echo performance > /sys/devices/system/cpu/cpu0/cpufreq/scaling_governor
```

**NOTE**
You will need sudo privileges to run this command and this setting is reset after system reboot.

Imbalance Penalty
Imbalance penalties are usually due to static partitioning of a data parallel workload that have varying costs per iteration of the loop or when the granularity of the block size is too large when dynamic partitioning is used. Many times, addressing the startup penalties will improve the amount of imbalance in the algorithm, but if they are retained, the following options may be tried to improve performance:

- If the algorithm uses range, the runtime is automatically picking the block size and may be causing the imbalance. You can override this by using nd_range and specifying a block size that would eliminate the imbalance.
- If nd_range is used, this issue may be caused by using a block size that is larger than optimal. Reducing the block size may improve performance or using range and letting the runtime decide may also be an option.

Experimental Support for OpenMP* Applications
You can now trace, visualize, and analyze OpenMP* parallel regions, tasks, and task dependencies in your application with the Flow Graph Analyzer.

The Flow Graph Analyzer support for OpenMP technology is experimental and currently covers two basic scenarios:

- OpenMP parallel regions are nested inside a Threading Building Blocks (TBB) flow graph. For this case, the Flow Graph Analyzer shows the execution of the parallel regions in the per-thread task execution timelines.
  
  The sample code below, omp_nested.cpp, is an example of an OpenMP construct nested inside a TBB flow graph:

```cpp
#include "tbb/tbb.h"
#include "tbb/flow_graph.h"
#include<omp.h>
#include <iostream>
```
using namespace tbb;
using namespace tbb::flow;
int main() {
  graph g;
  const int size = 20;
  continue_node<continue_msg> hello( g,
    []( const continue_msg & ) {
      std::cout << "Hello\n";
      tbb::parallel_for(0, size, 1, [=](int k) {
        std::cout << k << "\n"; });
    });
  continue_node<continue_msg> world( g,
    []( const continue_msg & )
    std::cout << " World\n";
    #pragma omp parallel for for (int i=0; i<20; i++) {std::cout << i <<"\n"; }
  });
  make_edge(hello, world);
  hello.try_put(continue_msg());
g.wait_for_all();
  return 0;
}
• OpenMP tasks that use `depends` clauses. In this, the Flow Graph Analyzer shows task execution in the timelines and provides experimental support that lets you see the dependency relationships between OpenMP tasks as a graph in the graph canvas.

The sample code below, `omp_depend.cpp`, is a hello-world example of OpenMP task dependencies:

```cpp
#include <omp.h>
#include<iostream>

int main() {
  #pragma omp parallel
  {
    std::string s = "";
    #pragma omp single
    {
      #pragma omp task depend( out: i)
      {
        s = "hello";
        printf("%s", s);
      }
      #pragma omp task depend( out: s )
      {
        s = "world";
        printf("%s",s);
      }
    }
    return 0;
  }
}
```

**Collecting Traces for OpenMP* Applications**

**NOTE**
OpenMP* trace collection is currently supported only on Linux* and macOS* operating systems.

To collect OpenMP traces for an application, you need an OMPT-enabled OpenMP* version 5.0 library, such as LLVM-OpenMP.
To build a sample code (for example, the `omp_depend.cpp` described in the Experimental Support for OpenMP* Applications) on a Linux OS with the Intel® C++ Compiler Classic, use the following command:

```bash
icpc -std=c++11 -qopenmp omp_depend.cpp –o example
```

**NOTE** Please use `–g` compiler flag to enable symbol resolution information.

To collect traces for OpenMP applications and create XML files, follow these steps:

1. Set the `FGT_ROOT` variable and update your `PATH` environment variable as shown in the table below.

<table>
<thead>
<tr>
<th>OS</th>
<th>Environment Variable</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Linux*</td>
<td>FGT_ROOT</td>
<td><code>&lt;advisor-install-dir&gt;/fga&lt;version&gt;/fgt</code></td>
<td>The path to the Flow Graph Collector installation</td>
</tr>
<tr>
<td></td>
<td>PATH</td>
<td><code>${FGT_ROOT}/linux/bin:${FGT_ROOT}/linux/bin/ia32/cc4.8_libc2.19_kerne13.0:${FGT_ROOT}/linux/bin/intel64/cc4.8_libc2.19_kerne13.0:${PATH}</code></td>
<td>The path must include paths to fgtrun.sh, fgtrun.csh, and fgt2xml.exe.</td>
</tr>
</tbody>
</table>

2. To enable tracing from OMPT, set the following variables:

<table>
<thead>
<tr>
<th>OS</th>
<th>Environment Variable</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Linux*</td>
<td>OMP_TOOL</td>
<td>Enabled</td>
<td>OMPT tool support enabler.</td>
</tr>
<tr>
<td></td>
<td>OMP_TOOL_LIBRARIES</td>
<td><code>${FGT_ROOT}/lib/intel64/cc4.8_libc2.19_kerne13.0/libfgt.so</code></td>
<td>Path to OpenMP collector library.</td>
</tr>
</tbody>
</table>

3. Run the application.

   If your paths are set up correctly, the application generates one or more files that start with `_fgt`. There is one file per thread that participates in executing the parallelism in the application. So, for example, if two threads participate in the execution of the flow graph, running the application generates two files, `_fgt.0` and `_fgt.1`, in an autogenerated folder in the format `_fga_YYYYMMDD_HHMMSS` according to its creation (for example, `_fga_20191111_1111`).

4. Convert the trace files to GraphML* and TraceML* format.

   Convert the `_fgt` binary files to the XML format understood by the Flow Graph Analyzer using the `fgt2xml.exe` converter in the directory containing the folder with the trace files:

   ```bash
   fgt2xml.exe <desired_name> --omp_experimental
   ```
or

\texttt{fgt2xml.exe --omp_experimental <desired_name>}

or

\texttt{fgt2xml.exe --omp_experimental}

This converter scans the current directory for all _fgt files within the most recent folder according to its name and generates two output files: <desired_name>.graphml and <desired_name>.traceml. If you do not provide a <desired_name>, the converter creates unknown.graphml and unknown.traceml.

The \texttt{omp_experimental} flag enables displaying a subgraph and tasks dependence graph. By default, display support is disabled and you see only information related to OpenMP constructs in the per-thread traces.

\textbf{OpenMP* Constructs in the Per-Thread Task View}

After you run the steps from the Collecting Traces for OpenMP* Applications section, you can see the execution of the OpenMP* tasks in the per-thread timelines. A display example for the \texttt{omp_depend.cpp} sample is shown below. The OpenMP region names are prefixed with \texttt{omp}.

\textbf{OpenMP* Constructs in the Graph Canvas}

To map OpenMP* parallel regions and task constructs to a graph, run the \texttt{fgt2xml} converter with the \texttt{--omp_experimental} flag. In such graph, nodes represent parallel regions and tasks, and edges represent task dependencies.

\textbf{Parallel Regions}

All OpenMP-related parallelism is contained within OpenMP parallel regions. In the Flow Graph Analyzer, a parallel region is mapped to a subgraph node in the graph canvas. Inside the subgraph node are at least two nodes:

- A node that represents the start of the parallel region.
- A node that represents the implicit barrier at the end of the region.

For example, for an empty parallel region like the following, the Flow Graph Analyzer creates a subgraph node, such as \texttt{omp0::n0}, in the graph canvas.

```cpp
#pragma omp parallel
{
}
```
When you double-click the subgraph node, you see the following, where `omp0::n0::n1` is the start of the parallel region and `omp0::n0::n2` is the implicit barrier at the end of the node.

### OpenMP* Tasks

An OpenMP* task is a block of code contained in a parallel region that can be executed simultaneously with other tasks in the same region. In the Flow Graph Analyzer, an OpenMP task is mapped to a generic node. For example, in the code below, there are two tasks: one prints `hello` and the other prints `world`. The order in which these tasks execute is not specified, so they can execute in any order. However, the two tasks always start after the enclosing parallel region begins, and they complete before the enclosing parallel region ends.

```c
#pragma omp parallel
{
    #pragma omp task
    {  printf("hello "); }
    #pragma omp task
    {  printf("world "); }
}
```

When you visualize this program in the Flow Graph Analyzer, it looks like this:

When you double-click this subgraph, you see the following, where `omp0::n0::n1` is the start of the parallel region, `omp0::n0::n4` is the implicit barrier at the end of the region, `omp0::n0::n2` is the "hello" task and `omp0::n0::n3` is the "world" task.
OpenMP* Task Dependencies

In the OpenMP* specification, a partial ordering of tasks can be expressed with depend clauses. The task dependence is fulfilled when the predecessor task completes. There are three dependency types supported by the OpenMP API: in, out, and in-out:

- **in** dependency type: The generated task is a dependent task of all previously generated sibling tasks that reference at least one of the list items in an out or in-out clause.
- **out** and **in-out** dependency types: The generated task is a dependent task of all previously generated sibling tasks that reference at least one of the list items in an in, out, or in-out clause.

In the Flow Graph Analyzer, task dependencies are represented by edges between the nodes that represent OpenMP tasks.

It is important to understand what dependencies are visualized in the Flow Graph Analyzer.

- **The task dependency graph** represents the partial order set by the depend clauses for the OpenMP tasks executed by the application. The nodes in the graphs are OpenMP tasks and the edges represent the partial order.
- **To reduce the complexity of the graph**, the Flow Graph Analyzer **omits some transitive dependencies**. A transitive dependence is a dependency between three tasks, such that if it holds between the first and the second tasks and between the second and the third tasks, it must hold between the first and the third tasks. In the figure below, the node 

  ![Graph Example](image)

  • **Part (a)** of the figure shows an example that only includes dependencies due to a single location x. **Because** a \( \lessdot_x b \) and \( b \lessdot_x d \), the Flow Graph Analyzer **does not show the transitive edge** a \( \lessdot_x d \).

  • **Part (b)** of the figure shows two locations x and y that determine the partial order. **There are two potential dependency edges** from a to d: a \( \lessdot_x d \) and a \( \lessdot_y d \). The Flow Graph Analyzer **includes an edge from a to d** because a is the direct source of y for d, but it excludes a \( \lessdot_x d \).
**NOTE** If there are parallel edges between two nodes and at least one of them can be omitted due to transitivity, they all can be omitted without changing the partial order. The Flow Graph Analyzer includes edges like \( a < y \) \( d \) in the graph topology because including edges to satisfy all required data dependencies is the most natural representation.

For example:

```c
#pragma omp parallel
{
    std::string s = "";
    #pragma omp single
    {
        #pragma omp task depend( out: s)
        {
            s = "hello";
            printf("%s", s);
        }
        #pragma omp task depend( out: s )
        {
            s = "world";
            printf("%s",s);
        }
    }
}
```

This application, when visualized with the Flow Graph Analyzer, has a single top-level subgraph node representing the OpenMP parallel region.

When you double-click this subgraph, you see the following:

The edge between `omp0::n0::n2` and `omp0::n0::n3` represents task dependency due to the variable `s`.

The main components of the Flow Graph Analyzer include the treemap view, the graph-topology canvas, the timeline and concurrency histogram view, and the critical-path report. OpenMP task traces map naturally to these views:

- The treemap view shows the time spent in each OpenMP parallel region, colored according to the average application concurrency during the time it was executing.
- The graph topology canvas shows the partial ordering of the tasks.
- The timeline and concurrency histogram view show the execution of each task on the OpenMP runtime threads and the application concurrency over time.
- The critical report shows the most time-consuming path from each source to each sink in the graph, sorted with the longest critical path at the top.

For more examples, see [https://link.springer.com/chapter/10.1007/978-3-319-98521-3_12](https://link.springer.com/chapter/10.1007/978-3-319-98521-3_12).
OpenMP* Nodes to Source Code Mapping:
In addition to the graphical view of OpenMP* task dependency graphs, the Flow Graph Analyzer also shows nodes mapping to corresponding source code. To get this information, you must build an OpenMP application with the `–g` flag.

For example, source code mapping with subgraph nodes in a parallel region looks as follows:

![Source code mapping example](image)

Sample Trace Files
The Flow Graph Analyzer includes five sample traces you can explore to get familiar with the features of the tool. These traces are in the `<advisor_installation>/fga/samples` directory. Whenever you launch the Flow Graph Analyzer, the File dialog box defaults to the samples directory.

The samples subdirectories contain samples you can load. The samples are described in the table below and explained in more detail in the sections that follow.

<table>
<thead>
<tr>
<th>Location</th>
<th>XML files</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>samples/code_generation</td>
<td>dining_like.graphml</td>
<td>Generates a Dining Philosophers sample.</td>
</tr>
<tr>
<td></td>
<td>feature_like.graphml</td>
<td>Generates a Features Detection sample.</td>
</tr>
<tr>
<td>samples/performance_analysis</td>
<td>feature_detection.graphml</td>
<td>Provides a runtime trace of a feature detection sample.</td>
</tr>
<tr>
<td></td>
<td>feature_detection.traceml</td>
<td></td>
</tr>
<tr>
<td></td>
<td>forward_substitution.graphml</td>
<td>Provides a runtime trace of a forward substitution sample.</td>
</tr>
<tr>
<td></td>
<td>forward_substitution.traceml</td>
<td></td>
</tr>
<tr>
<td></td>
<td>computerVision.graphml</td>
<td>Provides a runtime trace of a computer vision sample.</td>
</tr>
<tr>
<td></td>
<td>computerVision.traceml</td>
<td></td>
</tr>
</tbody>
</table>
NOTE The performance_analysis samples were captured by runtime tracing. Because runtime tracing cannot infer all types and cannot capture the user bodies of nodes, these samples do not contain complete descriptions of applications and cannot be used to regenerate the applications. You can generate a C++ code from these samples, but it will be incomplete and will need modification before compilation and execution. In contrast, the code_generation samples were completely described from within Flow Graph Analyzer. When you generate a C++ code from the code_generation samples, no modifications are necessary before compilation and execution.

code_generation Samples

Dining Philosophers

The dining_like.graphml sample provides a complete description of a Threading Building Blocks (TBB) flow graph application that implements a version of the dining philosophers problem. You can generate a complete TBB flow graph by loading this file in to the Flow Graph Analyzer and then following the instructions provided in the Generating C++ Stubs section.

Feature Detection

The feature_like.graphml sample provides a complete description of a feature detection application based on the example described in the blog posting at https://software.intel.com/content/www/us/en/develop/blogs/a-feature-detection-example-using-the-intel-threading-building-blocks-flow-graph.html. You can generate a complete Threading Building Blocks (TBB) flow graph by loading this file into the Flow Graph Analyzer and then following the instructions provided in the Generating C++ Stubs section.
Forward Substitution with Trace

The forward_substitution.graphml sample shows the topology and behavior of a Threading Building Block (TBB) flow graph application that provides an implementation of forward substitution on a lower-triangular matrix. The trace is for a single execution of the graph, using 4 threads for a 8192x8192 matrix with a block size of 128. The runtime trace of the application is contained in the matching forward_substitution.traceml file. This matching file is loaded automatically by the Flow Graph Analyzer.
Feature Detection with Trace


This trace was collected using 8 threads and 32 buffers provided to the buffer queue. The concurrency varies over time, but is limited to 8 threads at most.
Computer Vision with Trace

The `computer_vision.graphml` sample shows the topology and behavior of a Threading Building Blocks (TBB) flow graph application that represents a classic example of data flow parallelism. It is composed of three different computer vision (CV) algorithms that process the same input data. The data is a video input stream, and you can observe a resulting regular pattern in the timeline chart (the trace contains around 20 frames).

Notice the following:
You can use the critical path calculation functionality (turquoise box) to identify bottlenecks in the data flow. As a result of this feature, all nodes on the critical path are highlighted.

Zoom in the timeline to analyze a single frame execution in detail. The frame execution flow is the following:

1. The source node spawns a task. This is the first stage of the image processing pipeline.
2. A limiter node is used to balance the pipeline. It forwards the frame only if the number of frames that are currently executed is below a user-specified threshold.
3. Three different algorithms are executed in parallel. Concurrency changes during the algorithm stage because less work is available. In the timeline, a high concurrency is colored in green.

For a Threading Building Blocks (TBB) flow graph, an external activity can be encapsulated in a predefined async node. This activity represents offloading work to an Accelerator (for example, FPGA, GPU). The beginning and end of this activity are displayed as green vertical lines in the timeline. You can find a single execution within a single frame for each CV algorithm (represented by the nodes CV serial, CV nested, CV async). CV nested represents a node with a nested TBB parallel for algorithm that consumes most of the CPU time on average.
Additional Resources

- Flow Graph Analyzer is released as a feature of Intel® Advisor. You can find out more information about Intel® Advisor and instructions on how to obtain it and its components at https://software.intel.com/content/www/us/en/develop/tools/advisor.html.

Reference

Data Reference

This reference section describes the contents of data columns in Survey and Refinement Reports.

A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P | Q | R | S | T | U | V | W | XYZ

A
- Access Pattern
- Access Type
- Address Range
- Average

Access Pattern

Description: Summary of access types.

Collected during Memory Access Patterns Analysis and found in Loop Information Pane (Refinement Reports).

Access Type

Description: Memory access type: Read, Write, Read/Write

Collected during Memory Access Patterns Analysis and found in Memory Access Patterns Report.

Address Range

Description: Instruction address range in memory.

Interpretation: A wide range indicates one or more of the following:
- The application uses too much memory.
- Memory usage is not optimal.

Average

Description: Loop trip count average.

Collected during Trip Counts Analysis and Roofline Analysis, and found in Loop Information Pane (Survey Report).
**Prerequisites for collection/display:** Enabled *Trip Counts* on Workflow tab or enabled *Collect information about Loop Trip Counts* on *Trip Counts and FLOP Analysis* tab of *Project Properties Dialog Box*.

### B
### C
- Cache Line Utilization
- Cache Misses
- Call Count
- Compiler Estimated Gain

**Cache Line Utilization**
**Description:** Simulated cache line utilization for data transfer operations.
**Collected** during Memory Access Patterns Analysis and **found** in Loop Information Pane (Refinement Reports).

**Cache Misses**
**Description:** Number of memory load operations served by memory subsystem higher than cache. Calculated for the first instance of the loop (assuming cold CPU cache). Value is a result of virtual cache modeling, which might not match exact counter reported by hardware for this analysis run.
**Collected** during Memory Access Patterns Analysis and **found** in Loop Information Pane (Refinement Reports).

**Call Count**
**Description:** Number of times loop/function was invoked.
**Collected** during Trip Counts Analysis and Roofline Analysis, and **found** in Loop Information Pane (Survey Report) and Advanced View Pane (Survey Report).

**Prerequisites for collection/display:** Enabled *Trip Counts* on Workflow tab or enabled *Collect information about Loop Trip Counts* on *Trip Counts and FLOP Analysis* tab of *Project Properties Dialog Box*.

**Interpretation:** A high number means there is an outer loop in the selected loop call chain with high trip count values. If the loop has a low trip count value, the outer loop could be a better candidate for parallelization (threading/vectorization).

**Compiler Estimated Gain**
**Description:** Theoretical compiler estimate of relative loop performance speedup achieved or achievable due to vectorization.
**Collected** during Survey Analysis and Roofline Analysis, and **found** in Loop Information Pane (Survey Report) and Advanced View Pane (Survey Report).

**Comparison with similar metrics:** *Gain Estimate* is Intel Advisor-calculated estimate of relative loop performance speedup achieved due to vectorization.

### D
- Data Types
- Description
- Dirty Evictions
Data Types

**Description:** Data types provided by binary static analysis.

**Collected** during Survey Analysis and Roofline Analysis, and **found** in Loop Information Pane (Survey Report) and Advanced View Pane (Survey Report).

**Interpretation:** Bold indicates primary data type used for vectorization.

Description

**Description:** Code location classification.

**Collected** during Dependencies Analysis and **found** in Dependencies Report.

Dirty Evictions

**Description:** Number of evicted cache lines with a modified state introducing upstream memory traffic to a higher memory subsystem.

**Collected** during Memory Access Patterns Analysis and **found** in Loop Information Pane (Refinement Reports).

E

- Efficiency
- Elapsed Time

Efficiency

**Description:** Intel Advisor-calculated performance estimated gain compared to maximum achievable gain from vectorization.

**Collected** during Survey Analysis and Roofline Analysis, and **found** in Loop Information Pane (Survey Report).

**Interpretation:** Normally means how effectively vectorization was applied, compared to maximum possible gain (higher is better).

**Calculation/Aggregation:** \((\text{Estimated gain}/\text{Vector length}) \times 100\%\)

**Interpretation:** Hover mouse over data cell for more information.

Elapsed Time

**Description:** Elapsed (wall-clock) application time.

**Collected** during Survey Analysis and Roofline Analysis, and **found** in Filters banner.

F

- First Instance Site Footprint
- Function
- Function Call Sites and Loops

First Instance Site Footprint

**Description:** For each memory access instruction for the first instance of a loop, the Intel Advisor:

- Tracks the minimum and maximum access addresses.
- Displays the maximum range in this metric.

**Collected** during Memory Access Patterns Analysis and **found** in Loop Information Pane (Refinement Reports).
Comparison with similar metrics: This metric is more reliable than the Maximum Per-Instruction Address Range metric.

<table>
<thead>
<tr>
<th>Max. Per-Instruction Addr. Range</th>
<th>First Instance Site Footprint</th>
<th>Simulated Memory Footprint</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of threads analyzed for loop/site</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Number of loop instances analyzed</td>
<td>All instances, but with some memory access instruction filtering</td>
<td>1</td>
</tr>
<tr>
<td>Awareness of overlap between address ranges accessed in loop</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Suitability for code with random memory access</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

**Function**

**Description:** Function name.

**Collected** during Dependencies Analysis and found in Dependencies Report.

**Function Call Sites and Loops**

**Description:** Information about parent function, source file, and line where site/loop begins in Loop Information Pane (Survey Report), and top-down call tree of target functions and loops in Loop Information Pane (Survey Report).

**Collected** during Survey Analysis and Roofline Analysis, and found in Loop Information Pane (Survey Report) and Advanced View Pane (Survey Report).

**Interpretation:**

- - Scalar function.
- - Vectorized function.
- - Scalar loop. Vectorization might be possible.
- - Vectorized loop. Optimization might be possible.
- - Scalar inner loop within vectorized outer loop. Optimization might be possible.

**Gain Estimate**

**Description:** Intel Advisor-calculated estimate of relative loop performance speedup achieved due to vectorization.
Collected during Survey Analysis and Roofline Analysis, and found in Loop Information Pane (Survey Report).

Comparison with similar metrics: Compiler Estimated Gain is the theoretical compiler estimate of relative loop performance speedup achieved or achievable due to vectorization.

H

I

- Instruction Address
- Instruction Sets
- Iteration Duration

Instruction Address
Description: Instruction address in memory.
Collected during Dependencies Analysis and found in Dependencies Report.

Instruction Sets
Description: Instruction Set Architecture (ISA) usage for individual instructions.
Collected during Survey Analysis and Roofline Analysis, and found in Loop Information Pane (Survey Report) and Advanced View Pane (Survey Report).

Iteration Duration
Description: Average loop iteration time.
Collected during Trip Counts Analysis and Roofline Analysis, and found in Loop Information Pane (Survey Report) and Advanced View Pane (Survey Report).

Prerequisites for collection/display: Enabled Trip Counts on Workflow tab or enabled Collect information about Loop Trip Counts on Trip Counts and FLOP Analysis tab of Project Properties Dialog Box.

J

K

L

- Loop Instance Total Time
- Loop-Carried Dependencies

Loop Instance Total Time
Description: Average loop instance total time.
Collected during Trip Counts Analysis and Roofline Analysis, and found in Loop Information Pane (Survey Report).

Prerequisites for collection/display: Enabled Trip Counts on Workflow tab or enabled Collect information about Loop Trip Counts on Trip Counts and FLOP Analysis tab of Project Properties Dialog Box.
Loop-Carried Dependencies

**Description**: Dependencies summary across iterations

**Collected** during Dependencies Analysis and **found** in Loop Information Pane (Refinement Reports).

**Possible values:**
- **RAW** (Read after Write) - Flow dependency
- **WAR** (Write after Read) - Anti dependency
- **WAW** (Write after Write) - Output dependency

**M**
- **Max**
- **Max Site Footprint**
- **Maximum Per-Instruction Address Range**
- **Memory Access Footprint**
- **Memory Loads**
- **Memory Stores**
- **Memory, GB**
- **Min**
- **Module/Modules**
- **Multi-Pumping Factor**

**Max**

**Description**: Loop trip count maximum.

**Collected** during Trip Counts Analysis and Roofline Analysis, and **found** in Loop Information Pane (Survey Report).

**Prerequisites for collection/display**: Enabled Trip Counts on Workflow tab or enabled Collect information about Loop Trip Counts on Trip Counts and FLOP Analysis tab of Project Properties Dialog Box.

**Max Site Footprint**

**Description**: Maximum distance (among all instances of the loop) between the minimum and maximum memory address values.

**Maximum Per-Instruction Address Range**

**Description**: For most memory access instructions for all instances of a loop, the Intel Advisor:
- Tracks the minimum and maximum access addresses.
- Displays the maximum range in this metric.

The value may be imprecise because the Intel Advisor filters some memory access instructions while analyzing all instances of a loop. Unreliable values are displayed in gray.

**Collected** during Memory Access Patterns Analysis and **found** in Loop Information Pane (Refinement Reports) and Memory Access Patterns Report.

**Comparison with similar metrics**: This metric is less reliable than the First Instance Site Footprint metric.
<table>
<thead>
<tr>
<th></th>
<th>Max. Per-Instruction Addr. Range</th>
<th>First Instance Site Footprint</th>
<th>Simulated Memory Footprint</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of threads analyzed for loop/site</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Number of loop instances analyzed</td>
<td>All instances, but with some memory access instruction filtering</td>
<td>1</td>
<td>Depends on loop call count limit:</td>
</tr>
<tr>
<td>Awareness of overlap between address ranges accessed in loop</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Suitability for code with random memory access</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
</tr>
</tbody>
</table>

### Memory Access Footprint

**Description:** Maximum distance (among all instances of the loop) between minimum and maximum memory address values, accessed by the instructions, generated from the current source line.

### Memory Loads

**Description:** Number of memory load operations in first instance of the loop.

**Collected** during Memory Access Patterns Analysis and **found** in Loop Information Pane (Refinement Reports).

### Memory Stores

**Description:** Number of memory store operations in first instance of the loop.

**Collected** during Memory Access Patterns Analysis and **found** in Loop Information Pane (Refinement Reports).

### Memory, GB

**Description:** Number of data transfers, in GB, between the CPU and memory subsystem.

---

**Important**
This is a core metric that is the basis of the arithmetic intensity (AI) calculation.

### Min

**Description:** Loop trip count minimum.

**Collected** during Trip Counts Analysis and Roofline Analysis, and **found** in Loop Information Pane (Survey Report).
Prerequisites for collection/display: Enabled Trip Counts on Workflow tab or enabled Collect information about Loop Trip Counts on Trip Counts and FLOT Analysis tab of Project Properties Dialog Box.

Module/Modules
Description: Executable or library name.
Collected during Survey Analysis, Roofline Analysis, Dependencies Analysis, and Memory Access Patterns Analysis; and found in Loop Information Pane (Survey Report), Advanced View Pane (Survey Report), Dependencies Report, and Memory Access Patterns Report.

Multi-Pumping Factor
Description: The number of times the compiler applied a pumping optimization to extend vector length.
Collected during Survey Analysis and Roofline Analysis, and found in Loop Information Pane (Survey Report) and Advanced View Pane (Survey Report).

N
• Nested Function

Nested Function
Description: Name of the function (invoked from the site) where the stride diagnostic was detected.
Collected during Memory Access Patterns Analysis and found in Memory Access Patterns Report.

O
• Optimization Details

Optimization Details
Description: Compiler optimization details.
Collected during Survey Analysis and Roofline Analysis, and found in Loop Information Pane (Survey Report) and Advanced View Pane (Survey Report).

P
• Performance Issues
• Problem Severity

Performance Issues
Description: Performance issues found.
Collected during Survey Analysis, Roofline Analysis, and Memory Access Patterns Analysis, and found in Loop Information Pane (Survey Report) and Memory Access Patterns Analysis.

Interpretation: Click to display confidence level about issue root cause and recommended fixes.

Problem Severity
Description: Seriousness of a detected problem.
Collected for during Dependencies Analysis and found in Loop Information Pane (Refinement Reports).
Possible values:
• - Error.
• ⚠ - Warning.
• 🔄 - Informational.

Q

R
• RFO Cache Misses

RFO Cache Misses
Description: Number of cache lines loaded to cache due to a modification request (Request for Ownership).
Collected during Memory Access Patterns Analysis and found in Loop Information Pane (Refinement Reports).

S
• Self AI
• Self Elapsed Time
• Self GFLOP
• Self GFLOPS
• Self Giga OP
• Self Giga OPS
• Self GINTOP
• Self GINTOPS
• Self INT AI
• Self Memory (GB)
• Self Memory (GB/s)
• Self Overall AI
• Self Time
• Simulated Memory Footprint
• Site Location
• Site Name
• Source/Source Location/Sources
• State
• Stride
• Strides Distribution

Self AI
Description: Ratio of Self GFLOPS to self L1 transferred bytes.
Collected during Trip Counts Analysis and Roofline Analysis, and found in Loop Information Pane (Survey Report).
Prerequisites for collection/display:
• Enabled FLOP on Workflow tab or enabled Collect information about FLOP, L1 memory traffic, and AVX-512 mask usage on Trip Counts and FLOP Analysis tab of Project Properties Dialog Box.
• Selected Show Floating-Point Operation Columns for column setting.

Self Elapsed Time
Description: Self Time-based wall time from beginning to end of loop/function execution, excluding time for callees.
**Collected** during Survey Analysis and Roofline Analysis, and **found** in Loop Information Pane (Survey Report) and Advanced View Pane (Survey Report).

**Comparison with similar metrics:** Total Elapsed Time is Total Time-based wall time from beginning to end of loop/function execution, including time for callees.

**Interpretation:** Same as **Self Time** for single-threaded applications

**Self GFLOP**

**Description:** Giga floating-point operations, excluding GFLOP for callees.

**Collected** during Trip Counts Analysis and Roofline Analysis, and **found** in Loop Information Pane (Survey Report).

**Prerequisites for collection/display:**
- Enabled **FLOP** on Workflow tab or enabled Collect information about FLOP, L1 memory traffic, and AVX-512 mask usage on Trip Counts and FLOP Analysis tab of Project Properties Dialog Box.
- Selected **Show Floating-Point Operation Columns** for column setting.

**Self GFLOPS**

**Description:** Ratio of **Self GLOP** to **Self Elapsed Time**.

**Collected** during Trip Counts Analysis and Roofline Analysis, and **found** in Loop Information Pane (Survey Report).

**Prerequisites for collection/display:**
- Enabled **FLOP** on Workflow tab or enabled Collect information about FLOP, L1 memory traffic, and AVX-512 mask usage on Trip Counts and FLOP Analysis tab of Project Properties Dialog Box.
- Selected **Show Floating-Point Operation Columns** for column setting.

**Self Giga OP**

**Description:** Giga floating-point operations plus giga integer operations, excluding giga floating-point and integer operations for callees.

**Collected** during Trip Counts Analysis and Roofline Analysis, and **found** in Loop Information Pane (Survey Report).

**Prerequisites for collection/display:**
- Enabled **FLOP** on Workflow tab or enabled Collect information about FLOP, L1 memory traffic, and AVX-512 mask usage on Trip Counts and FLOP Analysis tab of Project Properties Dialog Box.
- Selected **Show Sum of Integer and Floating-Point Operation Columns** for column setting.

**Self Giga OPS**

**Description:** Ratio of **Self GFLOP** plus **Self GINTOP** to **Self Elapsed Time**.

**Collected** during Trip Counts Analysis and Roofline Analysis, and **found** in Loop Information Pane (Survey Report).

**Prerequisites for collection/display:**
- Enabled **FLOP** on Workflow tab or enabled Collect information about FLOP, L1 memory traffic, and AVX-512 mask usage on Trip Counts and FLOP Analysis tab of Project Properties Dialog Box.
- Selected **Show Sum of Integer and Floating-Point Operation Columns** for column setting.

**Self GINTOP**

**Description:** Giga integer operations, excluding giga integer operations for callees.
**Collect**ed during Trip Counts Analysis and Roofline Analysis, and **found** in Loop Information Pane (Survey Report).

**Prerequisites for collection/display:**
- Enabled FLOP on Workflow tab or enabled Collect information about FLOP, L1 memory traffic, and AVX-512 mask usage on Trip Counts and FLOP Analysis tab of Project Properties Dialog Box.
- Selected Show Integer Operation Columns for column setting.

**Self GINTOPS**

**Description:** Ratio of Self GINTOP to Self Elapsed Time.

**Collect**ed during Trip Counts Analysis and Roofline Analysis, and **found** in Loop Information Pane (Survey Report).

**Prerequisites for collection/display:**
- Enabled FLOP on Workflow tab or enabled Collect information about FLOP, L1 memory traffic, and AVX-512 mask usage on Trip Counts and FLOP Analysis tab of Project Properties Dialog Box.
- Selected Show Integer Operation Columns for column setting.

**Self INT AI**

**Description:** Ratio of Self GINTOPS to self L1 transferred bytes.

**Collect**ed during Trip Counts Analysis and Roofline Analysis, and **found** in Loop Information Pane (Survey Report).

**Prerequisites for collection/display:**
- Enabled FLOP on Workflow tab or enabled Collect information about FLOP, L1 memory traffic, and AVX-512 mask usage on Trip Counts and FLOP Analysis tab of Project Properties Dialog Box.
- Selected Show Integer Operation Columns for column setting.

**Self Memory (GB)**

**Description:** Data transfers between CPU and memory subsystem (total traffic, including caches and DRAM) in gigabytes, excluding transfers for callees.

**Collect**ed during Trip Counts Analysis and Roofline Analysis, and **found** in Loop Information Pane (Survey Report).

**Prerequisites for collection/display:** Enabled FLOP on Workflow tab or enabled Collect information about FLOP, L1 memory traffic, and AVX-512 mask usage on Trip Counts and FLOP Analysis tab of Project Properties Dialog Box.

**Self Memory (GB/s)**

**Description:** Data transfers between CPU and memory subsystem (total traffic, including caches and DRAM) in gigabytes per second, excluding transfers for callees.

**Collect**ed during Trip Counts Analysis and Roofline Analysis, and **found** in Loop Information Pane (Survey Report).

**Prerequisites for collection/display:** Enabled FLOP on Workflow tab or enabled Collect information about FLOP, L1 memory traffic, and AVX-512 mask usage on Trip Counts and FLOP Analysis tab of Project Properties Dialog Box.

**Calculation/Aggregation:** Self GBs / Self Elapsed Time

**Self Overall AI**

**Description:** Ratio of Self GFLOPS plus Self GINTOPS to self L1 transferred bytes.
**Collected** during Trip Counts Analysis and Roofline Analysis, and **found** in Loop Information Pane (Survey Report).

**Prerequisites for collection/display:**
- Enabled FLOP on Workflow tab or enabled **Collect information about FLOP, L1 memory traffic, and AVX-512 mask usage** on Trip Counts and FLOP Analysis tab of Project Properties Dialog Box.
- Selected **Show Sum of Integer and Floating-Point Operation Columns** for column setting.

**Self Time**

**Description:** Time actively executing a function/loop, excluding time for callees.

**Collected** during Survey Analysis and Roofline Analysis, and **found** in Loop Information Pane (Survey Report) and Advanced View Pane (Survey Report).

**Comparison with similar metrics:** **Total Time** is time actively executing a function/loop, including time for callees.

**Simulated Memory Footprint**

**Description:** The summarized and overlap-aware memory footprint across all instances of a loop.

**Collected** during Memory Access Patterns Analysis and found in Loop Information Pane (Refinement Reports).

**Prerequisites for collection/display:**

In the GUI Project Properties Dialog Box:
- Enable **Enable CPU cache simulation**.
- In the **Cache simulation mode** drop-down list, choose **Model cache misses and loop footprint**.
- Tweak other **Enable CPU cache simulation** parameters as necessary.

CLI example:

```
advixe-cl -collect map
-mark-up-list=1,2,7,17,26
-enable-cache-simulation
-cachesim-mode=footprint
-project-dir C:\my_advisor_project
-- my_application.exe
```

**Comparison with similar metrics:**

<table>
<thead>
<tr>
<th>Max. Per-Instruction Addr. Range</th>
<th>First Instance Site</th>
<th>Simulated Memory Footprint</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of threads analyzed for loop/site</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Number of loop instances analyzed</td>
<td>All instances, but with some memory access instruction filtering</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Depends on loop call count limit:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• GUI: Project Properties &gt; Analysis Target &gt; Memory Access Patterns Analysis &gt; Advanced &gt; Loop call count limit</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• CLI action option: -loop-call-count limit</td>
</tr>
</tbody>
</table>

348
<table>
<thead>
<tr>
<th>Max. Per-Instruction Addr. Range</th>
<th>First Instance Site Footprint</th>
<th>Simulated Memory Footprint</th>
</tr>
</thead>
<tbody>
<tr>
<td>Awareness of overlap between address ranges accessed in loop</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Suitability for code with random memory access</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

**Calculation/Aggregation**: Number of unique cache lines accessed during cache simulation * Cache line size.

For performance reasons, not all accesses and cache lines are simulated. Instead the Intel Advisor tracks a subset and then scales up to the whole cache size to determine the final footprint value.

**Site Location**

**Description**: Information about parent function, source file, and line where site/loop begins.

**Collected** during Dependencies Analysis and Memory Access Patterns Analysis, and **found** in Loop Information Pane (Refinement Reports).

**Site Name**

**Description**: Site name if using source annotations; sequence ID if marking loops for deeper analysis in Survey Report.

**Collected** during Dependencies Analysis and Memory Access Patterns Analysis, and **found** in Loop Information Pane (Refinement Reports), Dependencies Report, and Memory Access Patterns Report.

**Source/Source Location/Sources**

**Description**: Source file name(s) and line number(s).

**Collected** during Survey Analysis, Roofline Analysis, Dependencies Analysis and Memory Access Patterns Analysis; and **found** in Loop Information Pane (Survey Report), Advanced View Pane (Survey Report), Dependencies Report, and Memory Access Report.

**State**

**Description**: State of most severe problem in problem set.

**Collected** during Dependencies Analysis and **found** in Dependencies Report.

**Possible values**:

- **Regression** - Not investigated. Set by the Intel Advisor.
  Issue requires more investigation because it was marked as **Fixed** in baseline result but still appears.
- **New** - Not investigated. Set by the Intel Advisor or user.
  Issue did not appear in the baseline result, or there is no older result from which the Intel Advisor can propagate state information.
- **Not Fixed** - Not investigated. Set by user.
  Issue appeared in the baseline result and still requires investigation.
- **Confirmed** - Investigated. Set by user.
  Issue requires fixing but has not yet been fixed.
**Stride**

**Description:** Distance, in elements, between memory accesses in two consequent iterations.

**Collected** during Memory Access Patterns Analysis and found in Memory Access Patterns Report.

**Strides Distribution**

**Description:** Stride ratio in following format: Unit%/Constant%/Variable%

**Collected** during Memory Access Patterns Analysis and found in Loop Information Pane (Refinement Reports).

**T**

- **Total Elapsed Time**
- **Total Time**
- **Traits**
- **Transformations**
- **Type**

**Total Elapsed Time**

**Description:** Total Time-based wall time from beginning to end of loop/function execution, including time for callees

**Collected** during Survey Analysis and Roofline Analysis, and **found** in Loop Information Pane (Survey Report) and Advanced View Pane (Survey Report).

**Comparison with similar metrics:** Self Elapsed Time is Self Time-based wall time from beginning to end of loop/function execution, excluding time for callees.

**Interpretation:** Same as **Total Time** for single-threaded applications.

**Total Time**

**Description:** Time actively executing a function/loop, including time for callees.

**Collected** during Survey Analysis and Roofline Analysis, and **found** in Loop Information Pane (Survey Report) and Advanced View Pane (Survey Report).

**Comparison with similar metrics:** Self Time is time actively executing a function/loop, not including time for callees.

**Traits**

**Description:** Scalar and vectorization characteristics that may impact performance.

**Collected** during Survey Analysis and Roofline Analysis, and **found** in Loop Information Pane (Survey Report) and Advanced View Pane (Survey Report).

**Possible values:**

- **Fixed** - Investigated. Set by user.
  - Issue requires fixing and has been fixed.
- **Not a problem** - Investigated. Set by user.
  - Issue does not require fixing.
- **Deferred** - Investigated. Set by user.
  - You are postponing further investigation on an issue that may or may not require fixing.
<table>
<thead>
<tr>
<th>Trait</th>
<th>Detected ASM Instructions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Divisions</td>
<td><em>DIV</em></td>
</tr>
<tr>
<td>Square Roots</td>
<td><em>SQRT</em></td>
</tr>
<tr>
<td>Type Conversions</td>
<td><em>CVT</em></td>
</tr>
<tr>
<td>NT-stores</td>
<td><em>MOVNT</em></td>
</tr>
<tr>
<td>Gathers</td>
<td><em>GATHER</em></td>
</tr>
<tr>
<td>Scatters</td>
<td><em>SCATTER</em></td>
</tr>
<tr>
<td>Shuffles</td>
<td><em>SHUF</em></td>
</tr>
<tr>
<td>Permutations</td>
<td><em>PERM</em></td>
</tr>
<tr>
<td>Blends</td>
<td><em>BLEND</em></td>
</tr>
<tr>
<td>Packs</td>
<td><em>PACK</em></td>
</tr>
<tr>
<td>Unpacks</td>
<td><em>UNPCK</em></td>
</tr>
<tr>
<td>Inserts</td>
<td><em>INSERT</em></td>
</tr>
<tr>
<td>Extracts</td>
<td><em>EXTRACT</em></td>
</tr>
<tr>
<td>Masked Stores</td>
<td><em>MASKMOV</em></td>
</tr>
<tr>
<td>Shifts</td>
<td><em>PROR</em>, <em>PROL</em>, <em>PSLL</em>, <em>PSRA</em>, <em>PSRL</em></td>
</tr>
<tr>
<td>FMA</td>
<td><em>FMADD</em>, <em>FMSUB</em>, <em>FNADD</em>, <em>FNMSUB</em></td>
</tr>
<tr>
<td>Mask Manipulations</td>
<td><em>KADD</em>, <em>KTEST</em>, <em>KAND</em>, <em>KOR</em>, <em>KXOR</em>, <em>KXNOR</em>, <em>KNOT</em>, <em>KUNPCK</em>, <em>KMOV</em>, <em>KSHIFT</em></td>
</tr>
<tr>
<td>Conflict Detections</td>
<td><em>VPCONFLICT</em></td>
</tr>
<tr>
<td>Exponent extractions</td>
<td><em>VGETEXP</em></td>
</tr>
<tr>
<td>Mantissa extractions</td>
<td><em>VGETMANT</em></td>
</tr>
<tr>
<td>Expands</td>
<td><em>EXPAND</em></td>
</tr>
<tr>
<td>Compresses</td>
<td><em>COMPRESS</em></td>
</tr>
<tr>
<td>VNNI</td>
<td><em>VNNI</em></td>
</tr>
</tbody>
</table>

**Transformations**

**Description**: Loop transformations applied by compiler.

**Collected** during Survey Analysis and Roofline Analysis, and **found** in Loop Information Pane (Survey Report) and Advanced View Pane (Survey Report).

**Type**

**Collected** during Survey Analysis, Roofline Analysis, Dependencies Analysis, and Memory Access Patterns Analysis; and **found** in Loop Information Pane (Survey Report), Advanced View Pane (Survey Report), Dependencies Report, and Memory Access Patterns Report.
Possible Survey Report values:

- **Peeled/Remainder** - For loops that have child loops. Appears only when scalar peeled loop and/or remainder loop executed.
- **Threaded** - For loops that have child loops. Appears when some parallel framework (OpenMP* or automatically by Intel compiler) is used in the loop.
- **Vectorized (<loop part(s)>)** - For vectorized parent and child loops. Appears when a parent loop has any of the following parts executed: peeled, body, remainder. Also appears for child loops that have one of the following parts executed: peeled, body, remainder.
- **Peeled** - For small, (usually) compiler-generated loops created to align the memory accesses inside the loop body and maximize its efficiency.
- **Body** - For vectorized loops (compiler-generated from a source loop). Most loop iterations should execute in body, as body normally processes more data than peeled or remainder loops. Vector length in the body is usually larger than in peeled and/or remainder loops, which means body is the most efficient place for performance.
- **Remainder** - For (usually) compiler-generated loops created to clean up any remaining iterations that do not fit within the scope of the loop body.
- **[Not Executed]** - Mark that appears next to any other loop metric when a loop was not executed.
- **Scalar** - Appears when non-vectorized loops executed.
- **Completely Unrolled** - Appears when the loop body was copied several times (equal to trip counts value) by the compiler.
- **Inside vectorized** - Appears when the inner loop was vectorized in addition to the outer loop.
- **Inlined Function** - Appears when the function body was inlined into the loop/function body.
- **Vector Function** - Appears when a SIMD-enabled version of the function executed. (See Intel compiler documentation for details).
- **Function** - Appears when a scalar version of the function executed.

Possible Memory Access Patterns Report values:

- **Uniform stride 0** - Instruction accesses the same memory from iteration to iteration. Represents the ideal situation and does not require any improvements.
- **Unit stride (stride 1)** - Instruction accesses memory that consistently changes by one element from iteration to iteration. Represents the ideal situation and does not require any improvements.
- **Constant stride (stride N)** - Instruction accesses memory that consistently changes by N elements (N>1) from iteration to iteration. Code uses more memory than is ideal and requires more cache lines. Consider studying recommendations on AOS/SOA optimization.
- **Irregular stride** - Instruction accesses memory addresses that change by an unpredictable number of elements from iteration to iteration. Might limit vectorization or even make vectorization impossible.
- **Gather (irregular) stride** - Detected for v(p)gather* instructions on AVX2 Instruction Set Architecture (ISA).
  The compiler vectorized code with an irregular memory access pattern. Consider improving the code to use a more constant memory access pattern.

Possible Dependencies Report values - See Problem and Message Types.

- **Unroll Factor**
**Unroll Factor**

*Description:* Loop unroll factor applied by the compiler.

*Collected* during Survey Analysis and Roofline Analysis, and *found* in Loop Information Pane (Survey Report) and Advanced View Pane (Survey Report).

**V**

- Variable References
- Vector ISA
- Vector Widths
- Vectorization Details
- VL (Vector Length)

**Variable References**

*Description:* Name of the variable for which the dependency or memory access stride is detected.

*Collected* during Dependencies Analysis and Memory Access Patterns Analysis, and *found* in Dependencies Report and Memory Access Patterns Report.

**Vector ISA**

*Description:* The highest vector Instruction Set Architecture used for individual instructions.

*Collected* during Survey Analysis and Roofline Analysis, and *found* in Loop Information Pane (Survey Report) and Advanced View Pane (Survey Report).

*Comparison with similar metrics:* An ISA higher than the ISA of your current hardware appears when you add corresponding codepaths with x, Qx / ax, Qax compiler options. To see the ISA of non-executed codepaths, enable the Analyze non-executed codepaths option in Project Properties.

**Vector Widths**

*Description:* Vector register width in bits.

*Collected* during Survey Analysis and Roofline Analysis, and *found* in Loop Information Pane (Survey Report) and Advanced View Pane (Survey Report).

*Possible values:* Combination of values, including 32, 64, 128, 256, 512, delimited by a slash or semi-colon (/ or ;).

**Vectorization Details**

*Description:* Compiler notes on vectorization.

*Collected* during Survey Analysis and Roofline Analysis, and *found* in Loop Information Pane (Survey Report) and Advanced View Pane (Survey Report).

**VL (Vector Length)**

*Description:* The number of elements processed in a single iteration of vector loops, or the number of elements processed in individual vector instructions.

*Collected* during Survey Analysis and Roofline Analysis, and *found* in Loop Information Pane (Survey Report) and Advanced View Pane (Survey Report).

*Calculation/Aggregation:* Estimated by binary static analysis or the Intel compiler.

**W**

- Why No Vectorization?
**Why No Vectorization?**

**Description:** The reason the compiler did not vectorize the loop.

**Collected** during Survey Analysis and Roofline Analysis, and **found** in Loop Information Pane (Survey Report) and Advanced View Pane (Survey Report).

**Interpretation:** Click to display the issue root cause and recommended fixes.

**X, Y, Z**

---

**Command Line Interface Reference**

*This reference section describes the CLI actions and options used in the command syntax:* $ advixe-cl

$ advixe-cl <--action> [--action-options] [--global-options] [target [target options]]

The main advantage of using the Intel® Advisor command line interface, advixe-cl, instead of the GUI is you can collect data as part of an automated or background task, and then view the result in a command line interface (CLI) report or in the GUI at your convenience.

**NOTE**

Set command line environment variables before using the command line interface. For instructions, see Setting and Using Intel Advisor Environment Variables.

---

**advixe-cl Command Syntax**

The advixe-cl command syntax is:

$ advixe-cl <--action> [--action-options] [--global-options] [target [target options]]

where:

- **advixe-cl**
  - The name of the Intel Advisor command line tool.

- **<--action>**
  - The action to perform, such as collect or report. Each command has exactly one action. For example, you cannot use both the collect and report actions in the same command.

- **[--action-options]**
  - Action options modify behavior specific to the action. You can have multiple action options per command. Using an action option that does not apply to the action results in a usage error.

- **[--global-options]**
  - Global options modify behavior in the same manner for all actions. You can have multiple global options per action.

- **target**
  - The target (application executable) to analyze.

- **[target-options]**
  - Options that apply to the target.

**Action option/Global option rules:**

- If opposing action options are used on the same command line the last specified action option applies.
- An action option that is redundant or has no meaning in the context of the specified action is ignored.
- Attempted use of an inappropriate action option that might lead to unexpected behavior returns a usage error.
Syntax Alternatives

An action option or global option can be preceded by one or two dashes. This chapter uses one dash before
the short form of an action option/global option, and two dashes before the long form of an action option/
global option. For example: The following are equivalent:

$ advixe-cl --help
$ advixe-cl -help

An option-value pair can be separated by an equal sign (=) or by a space. This chapter uses an equal sign.
For example: The following are equivalent:

$ advixe-cl --report=survey
$ advixe-cl --report survey

The target executable must be preceded by two dashes and a space. For example:

$ advixe-cl --collect=survey -- myApplication

Some action options accept multiple arguments. Most of the time, you can pass these arguments in a
comma-separated string (with no spaces), or by repeating the action option. For example: The following are
equivalent.

$ advixe-cl --collect=survey --project-dir=./advi --exclude-files=./src/foo,./src/bar
-- myApplication
$ advixe-cl --collect=survey --project-dir=./advi --exclude-files=./src/foo --exclude-
files=./src/bar -- myApplication

Directories

Project Directory

By default, the project directory is your current working directory. Use the project-dir action option to
write a result to a different directory. For example:

Survey the application for hotspots and write the result to the ./advi project directory.

$ advixe-cl --collect=survey --project-dir=./advi --search-dir all:=./src --
myApplication

Generate a Survey report from the Survey result and write it to the ./advi project directory.

$ advixe-cl --report=survey --project-dir=./advi --format=text --report-output=./out/
survey.txt

Search Directory

Use the search-dir action option to specify the directories containing the source, symbol, and binary files
that support analysis.

You can specify multiple search directories. For example:

$ advixe-cl --collect=survey --project-dir=./advi --search-dir src:=./src1,./src2 --
myApplication

Tip
Always specify your search directories when using collect action.

User Data Directory
Use the `user-data-dir` action option to write result files to a directory other than `project-dir`, such as a remote directory or simply another directory when there is not enough space in `project-dir`.

For example: Collect Suitability data and write the result to a remote directory.

```bash
$ advixe-cl --collect=suitability --project-dir=./advi --user-data-dir=./remote_dir --search-dir src=../src -- myApplication
```

**advixe-cl Command Action Reference**

The `advixe-cl` command currently supports the actions shown below.

### collect

*Run the specified type of analysis and collect data.*

#### GUI Equivalent

**Workflow**

**Toolbar**

*File > New > Start [Name] Analysis*

#### Syntax

```bash
-c=<string> [--action-options] [--global-options] [[-- <target> [<target options>]]
--collect=<string> [--action-options] [--global-options] [[-- <target> [<target options>]]
```

#### Arguments

*<string>* is the type of analysis:

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>dependencies</td>
<td>Collect dependencies data to predict and eliminate data sharing problems.</td>
</tr>
<tr>
<td>map</td>
<td>Collect memory access patterns data.</td>
</tr>
<tr>
<td>roofline</td>
<td>Run the Survey analysis immediately followed by the Trip Counts &amp; FLOP analysis to visualize actual performance against hardware-imposed performance ceilings.</td>
</tr>
<tr>
<td>suitability</td>
<td>Collect suitability data by executing annotated code to analyze the proposed threading parallelism opportunities and estimate where performance gains are most likely.</td>
</tr>
<tr>
<td>survey</td>
<td>Survey the target (your executable application) and collect data about code that may benefit from (more) parallelism.</td>
</tr>
<tr>
<td>tripcounts</td>
<td>Collect the following data and add it to the Survey report: loop iteration, floating-point and integer operation, and memory traffic statistics, and more.</td>
</tr>
</tbody>
</table>

#### Default

No default argument

#### Modifiers

*app-working-dir, auto-finalize, benchmarks-sync, cache-config, cache-sources, cachesim-associativity, cachesim-cacheline-size, cachesim-mode, cachesim-sets, data-limit, data-transfer-page-size (Beta), delete-tripcounts, duration, enable-cache-simulation, enable-data-transfer-analysis (Beta), exclude-files, executable-of-interest, filter-by-scope, filter-reductions, flop, gpu-carm (Beta), gpu-sampling-interval (Beta),*

**Important**
The enable-data-transfer-analysis, data-transfer-page-size, track-heap-objects, profile-intel-perf-libs, track-stack-accesses, profile-gpu, gpu-carm, gpu-sampling-interval options are available only in the Intel® Advisor Beta, which is part of Intel® oneAPI Base Toolkit. For more information, see https://software.intel.com/content/www/us/en/develop/tools/oneapi/components/advisor.html.

**Example**
Survey the application to find candidates for code that may benefit from (more) parallelism.

```
$ advixe-cl --collect=survey --project-dir=./advi --search-dir src:r=./src -- ./bin/myApplication
```

Collect memory access patterns data on the specified loops.

```
$ advixe-cl --collect=map --mark-up-list=5,10,12 --project-dir=./advi --search-dir src:r=./src -- ./bin/myApplication
```

Collect survey data on four nodes of an MPI cluster into the shared ./advi project directory.

```
$ mpirun -n 4 advixe-cl --project-dir=./advi --collect=survey -- <PATH>/mpi-sample/1_mpi_sample_serial
```

Collect dependencies data for all innermost loops that account for over 2% of the total CPU time.

```
$ advixe-cl --collect=dependencies --project-dir=./advi --loops="loop-height=0,total-time>2" -- ./bin/myApplication
```

See Also
Command Line Interface Reference  This reference section describes the CLI actions and options used in the command syntax: $ advixe-cl <--action> [--action-options] [--global-options] [--] <target> [target options].
advixe-cl Command Option Reference

**GUI Equivalent**
Control the Intel Advisor while running analyses.

**Workflow**

**Syntax**

```
--command=<string> [--action-options] [--global-options] [--] <target> [target options]
```

**Arguments**

<string> is one of the following:
<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>cancel</td>
<td>Interrupt data collection without saving results.</td>
</tr>
<tr>
<td>detach</td>
<td>Similar to pause, except you can attach to an already running process.</td>
</tr>
<tr>
<td>pause</td>
<td>Pause data collection while the target application continues running.</td>
</tr>
<tr>
<td>resume</td>
<td>Resume paused data collection.</td>
</tr>
<tr>
<td>stop</td>
<td>Stop data collection.</td>
</tr>
<tr>
<td>status</td>
<td>Print collection status.</td>
</tr>
</tbody>
</table>

**Default**
No default argument

**Modifiers**
quiet, result-dir, verbose

**Usage**
Usage can decrease collection overhead.

**Example**
Pause the analysis run that is currently collecting data into result directory `r000hs`.

```bash
$ advixe-cl --command=pause -r=<PATH>/r000hs
```

**See Also**
Minimizing Analysis Overhead

Command Line Interface Reference
advixe-cl Command Option Reference

**create-project**
*Create an empty project, if it does not already exist.*

**GUI Equivalent**
File > New > Project...

**Syntax**
```
--create-project [--action-options] [--global-options] [[--] <target> [target options]]
```

**Modifiers**
project-dir, quiet, search-dir, verbose

**Usage**
Use the `--project-dir` action option to:
- Specify a project name.
- Create a project somewhere other than the current working directory.
**Example**

Create a new advi project in the current working directory.

```
$ advixe-cl --create-project --project-dir=./advi -- ./bin/myApplication
```

**See Also**

**Command Line Interface Reference**  This reference section describes the CLI actions and options used in the command syntax: $ advixe-cl <--action> [--action-options] [--global-options] [[--] target [target options]].

**advixe-cl Command Option Reference**

**help**

*Explain command line actions with corresponding options.*

**Syntax**

-h

--help

-h <action>

--help <action>

**Arguments**

<action> is one of the following: collect, command, create-project, import-dir, mark-up-loops, report, snapshot, version, workflow

**Description**

Explain command line actions with corresponding options.

**Examples**

Display overall help.

```
$ advixe-cl --help
```

Display help for the collect action.

```
$ advixe-cl -h collect
```

**See Also**

**Command Line Interface Reference**  This reference section describes the CLI actions and options used in the command syntax: $ advixe-cl <--action> [--action-options] [--global-options] [[--] target [target options]].

**advixe-cl Command Option Reference**

**import-dir**

*Import and finalize data collected on an MPI cluster.*

**Syntax**


**Arguments**

<PATH> is the full path to a directory where previously collected data resides.
**Default**

No default argument

**Modifiers**

mpi-rank, project-dir, quiet, search-dir, verbose

**Usage**

For best results, specify the location of the source application files using the search-dir action option. Use the mpi-rank action option to specify the process data to import.

For MPI workloads:

1. Copy the data from the rank node to the home node.
2. Import the data.
3. View the data.

**Example**

Import data collected on rank 2 of the MPI cluster to the new project.

```
$ advixe-cl --import-dir=./advi --project-dir=./new_advi --mpi-rank-2 --search-dir src:r=./src
```

**See Also**

Analyze MPI Workloads

Command Line Interface Reference  This reference section describes the CLI actions and options used in the command syntax: $ advixe-cl <--action> [--action-options] [--global-options] [--] target [target options].

advixe-cl Command Option Reference

**mark-up-loops**

After running a Survey analysis and identifying loops of interest, select loops (by file and line number or criteria) for deeper analysis.

**GUI Equivalent**

**Survey**

**Syntax**

```
```

**Modifiers**

append, loops, mpi-rank, project-dir, quiet, remove, select, verbose

**Usage**

Do not confuse the mark-up-loops action with the mark-up-list action option. The mark-up-loops action coupled with the select action option enables a GUI checkbox; therefore loop selection persists beyond the duration of the mark-up-loops action and applies to downstream analyses, such as Dependencies and Memory Access Patterns analyses. The collect action coupled with the mark-up-list action option simulates enabling a GUI checkbox; therefore loop selection persists only for the duration of the collect action.
Example

Select loops for downstream analysis based on file and line number.

```bash
$ advixe-cl --mark-up-loops --select=foo.cpp:34,bar.cpp:192 --project-dir=./myAdvisorProj -- ./bin/myTargetApplication
```

Select loops for downstream analysis based on criteria.

```bash
$ advixe-cl --mark-up-loops --loops="scalar,has-issue" --project-dir=./myAdvisorProj -- ./bin/myTargetApplication
```

See Also

mark-up-list After running a Survey analysis and identifying loops of interest, select loops (by file and line number or ID) for deeper analysis. 

Minimizing Analysis Overhead

Command Line Interface Reference
advixe-cl Command Option Reference

report

Generate a report from data collected during a previous analysis.

GUI Equivalent

File > Open > Results
File > Recent Results

Syntax

```
-R=<string> [--action-options] [--global-options] [--] target [target options]
```

Arguments

<string> is the list of available reports:

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>annotations</td>
<td>Report the annotations in the source code.</td>
</tr>
<tr>
<td>custom</td>
<td>Generate a custom report.</td>
</tr>
<tr>
<td>dependencies</td>
<td>Report results of a Dependencies analysis.</td>
</tr>
<tr>
<td>joined</td>
<td>Combine results for several analyses into a single report.</td>
</tr>
<tr>
<td>map</td>
<td>Report results of a Memory Access Patterns analysis.</td>
</tr>
<tr>
<td>roofline</td>
<td>Report results of a Roofline analysis.</td>
</tr>
<tr>
<td>roofs</td>
<td>Report roof values.</td>
</tr>
<tr>
<td>suitability</td>
<td>Report results of a Suitability analysis.</td>
</tr>
<tr>
<td>summary</td>
<td>Report the analysis summary.</td>
</tr>
</tbody>
</table>
### Table

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>survey</td>
<td>Report results of a Survey analysis.</td>
</tr>
<tr>
<td>threads</td>
<td>Report on threads.</td>
</tr>
<tr>
<td>top-down</td>
<td>Report results of a Survey analysis in top-down view.</td>
</tr>
<tr>
<td>tripcounts</td>
<td>Add trip counts data to a Survey report.</td>
</tr>
</tbody>
</table>

### Default

No default argument

### Modifiers

`bottom-up`, `csv-delimiter`, `data-type`, `display-callstack`, `dynamic`, `enable-task-chunking`, `filter`, `gpu` (Beta), `format`, `limit`, `memory-level`, `memory-operation-type`, `mix`, `mpi-rank`, `option-file`, `project-dir`, `quiet`, `recalculate-time`, `reduce-lock-contention`, `reduce-lock-overhead`, `reduce-site-overhead`, `reduce-task-overhead`, `refinalize-survey`, `report-output`, `report-template`, `search-dir`, `show-all-columns`, `show-all-rows`, `show-functions`, `show-loops`, `show-not-executed`, `sort-asc`, `sort-desc`, `target-system`, `threading-model`, `top-down`, `verbose`, `with-stack`

#### Important

The `gpu` option is available only in the Intel® Advisor Beta, which is part of Intel® oneAPI Base Toolkit. For more information, see https://software.intel.com/content/www/us/en/develop/tools/oneapi/components/advisor.html.

### Usage

Suitability reports are the most configurable.

Generate a report from data collected during a previous analysis.

#### Example

Generate a Suitability report.

```
$ advixe-cl --report=suitability --project-dir=./advi --search-dir src=r=./src --format=text --report-output=./out/suitability.txt
```

Generate a Dependencies report for data collected on rank 3 of MPI cluster:

```
$ advixe-cl --report=dependencies --project-dir=./advi --mpi-rank=3 --search-dir src=r=./src
```

### See Also

- `collect` Run the specified type of analysis and collect data.
- Command Line Interface Reference
- `advixe-cl Command Option Reference`

### GUI Equivalent

**File**  >  **Create Data Snapshot**
Syntax

Default
Save the current analysis result.

Modifiers
cache-binaries, cache-sources, mpi-rank, pack, project-dir, quiet, search-dir, verbose

Usage
Intel Advisor stores only the most recent analysis result. Visually comparing one or more snapshots to each other or to the most recent analysis result can be an effective way to judge performance improvement progress.

Example
Create a new snapshot in the project directory. Name it snapshotXXX (default name).

advixe-cl --snapshot --project-dir=./advi --no-pack

Create a new snapshot in the project directory. Name it new_snapshot.

advixe-cl --snapshot --project-dir=./advi --no-pack -- new_snapshot

Create a new snapshot. Pack it into an archive. Put it in the current directory. Name it snapshotXXX.advixeexpz (default name).

advixe-cl --snapshot --project-dir=./advi --pack

Create a new snapshot. Include sources and binaries. Pack it into an archive. Name it /tmp/new_snapshot.advixeexpz.

advixe-cl --snapshot --project-dir=./advi --pack --cache-sources --cache-binaries -- /tmp/new_snapshot

See Also
Command Line Interface Reference  This reference section describes the CLI actions and options used in the command syntax: $ advixe-cl <--action> [--action-options] [--global-options] [[--] target [target options]].
advice-cl Command Option Reference

version
Display product version information.

Syntax
-v

--version

Example
Write product version information to stdout.

$ advixe-cl --version

See Also
Command Line Interface Reference
advixe-cl Command Option Reference
**workflow**

*Explain typical Intel Advisor user scenarios, with corresponding command lines.*

**Syntax**

```
--workflow
```

**Usage**

Explain typical Intel Advisor user scenarios, with corresponding command lines. For example:

**Add SIMD Parallelism:**

1. Find hotspots.

   ```
   advixe-cl --collect=survey --project-dir=./advi --search-dir src:r=./src -- ./bin/myApplication
   advixe-cl --report=survey --project-dir=./advi --search-dir src:r=./src --format=text --report-output=./out/survey.txt
   ```

2. Determine the number of loop iterations.

   ```
   advixe-cl --collect=tripcounts --project-dir=./advi --search-dir src:r=./src -- ./bin/myApplication
   advixe-cl --report=tripcounts --project-dir=./advi --search-dir src:r=./src --format=text --report-output=./out/tripcounts.txt
   ```

3. Check for possible dependencies.

   ```
   advixe-cl --collect=dependencies --project-dir=./advi --search-dir src:r=./src -- ./bin/myApplication
   advixe-cl --report=dependencies --project-dir=./advi --search-dir src:r=./src --format=text --report-output=./out/dependencies.txt
   ```

4. Check memory access patterns.

   ```
   advixe-cl --collect=map --project-dir=./advi --search-dir src:r=./src -- ./bin/myApplication
   advixe-cl --report=map --project-dir=./advi --search-dir src:r=./src --format=text --report-output=./out/map.txt
   ```

5. Update the application to enable automatic compiler vectorization, or explicitly mark the loops you need to vectorize. Rebuild the application and test.

**Add Threading Parallelism**

1. Find hotspots. This step is similar to the first step in the SIMD-parallel workflow (above).

2. Determine the number of loop iterations. This step is similar to the second step in the SIMD parallel workflow (above).

3. Add annotations to the application source code and rebuild the application.

4. Collect suitability data. Note: Annotations must be present in the source code for this collection to be successful.

   ```
   advixe-cl --collect=suitability --project-dir=./advi --search-dir src:r=./src -- ./bin/myApplication
   advixe-cl --report=suitability --project-dir=./advi --search-dir src:r=./src --format=text --report-output=./out/suitability.txt
   ```

5. Check for the possible dependencies.

   ```
   advixe-cl --collect=dependencies --project-dir=./advi --search-dir src:r=./src -- ./bin/myApplication
   advixe-cl --report=dependencies --project-dir=./advi --search-dir src:r=./src --format=text --report-output=./out/dependencies.txt
   ```
6. Display a list of currently used annotations.

```bash
advixe-cl --report=annotations --project-dir=./advi --search-dir src:r=./src --format=text --report-output=./out/annotations.txt
```

7. Update the application using the chosen parallel coding constructs. Rebuild the application and test.

**Tip**

Use an option file for efficiency. Enter one option on each line. No spaces are allowed in the option entry; use a new line. The option file must be in UTF-8 format.

```bash
advixe-cl --report=annotations --option-file=./advi/option.txt
```

with an option.txt file that looks like this:

```bash
--project-dir
./advi
--search-dir
src:r=./src
--format=text
--report-output
./out/annotations.txt
```

**See Also**

Command Line Interface Reference  This reference section describes the CLI actions and options used in the command syntax: $ advixe-cl <<--action> [--action-options] [--global-options] [[--] target [target options]].

advixe-cl Command Option Reference

The advixe-cl command currently supports the options shown below.

**append**

Add loops (by file and line number) to the loops selected for deeper analysis.

**GUI Equivalent**

Survey > 🌐

**Syntax**

`--append=<string>`

**Arguments**

`<string>` is a comma-separated list of files/line numbers in the following format: file1:line1.

**Default**

No default argument

**Actions Modified**

`mark-up-loops`
Usage
Do not confuse the `mark-up-loops` action with the `mark-up-list` action option. The `mark-up-loops` action coupled with the `select` action option enables a GUI checkbox; therefore loop selection persists beyond the duration of the `mark-up-loops` action and applies to downstream analyses, such as Dependencies and Memory Access Patterns analyses. The `collect` action coupled with the `mark-up-list` action option simulates enabling a GUI checkbox; therefore loop selection persists only for the duration of the `collect` action.

Example
1. Run a Survey analysis.
2. Select a loop for deeper analysis.
3. Add `bar.cpp:192` to the selection list.
4. Run a Dependencies analysis on both loops.

```
$ advixe-cl --collect=survey --project-dir=./myAdvisorProj -- ./bin/myTargetApplication
$ advixe-cl --mark-up-loops --select=foo.cpp:34 --project-dir=./myAdvisorProj -- ./bin/myTargetApplication
$ advixe-cl --mark-up-loops --append=bar.cpp:192 --project-dir=./myAdvisorProj -- ./bin/myTargetApplication
$ advixe-cl --collect=dependencies --project-dir=./myAdvisorProj -- ./bin/myTargetApplication
```

See Also
Command Line Interface Reference
advixe-cl Command Action Reference

app-working-dir
Specify the directory where the target application runs during analysis, if it is different from the current working directory.

GUI Equivalent
Project Properties > Analysis Target > [Analysis Type] > Working directory

Syntax
`--app-working-dir=<PATH>`

Arguments

`<PATH>` is a string containing the PATH/name.

Default
Default is the current working directory.

Actions Modified
`collect`

Usage
If your data files are saved in a separate location from the target application, use the `app-working-dir` option to specify the target application working directory.
**Example**

Run a Survey analysis on `myApplication`. Use `work-dir` to launch `myApplication`.

```bash
> advixe-cl --collect=survey --app-working-dir=./work_dir --project-dir=./advi -- myApplication
```

**See Also**

Command Line Interface Reference
advixe-cl Command Action Reference

---

### auto-finalize

*Finalize Survey and Trip Counts & FLOP analysis data after collection is complete.*

**Syntax**

```bash
--auto-finalize
--no-auto-finalize
```

**Default**

On (auto-finalize)

**Actions Modified**

- `collect=survey`
- `collect=tripcounts`

**Usage**

Use the `--no-auto-finalize` option in situations where you want to suppress finalization, such as when you want to view collected data on a different machine. If finalization is suppressed during data collection and analysis, it occurs automatically when you open the result in the GUI or generate a report from the result. Disabling temporarily decreases overhead.

**Example**

Run a Survey analysis. Search recursively for source files in the `./src` search directory. Suppress finalization and write the unfinalized results to the `./advi` project directory instead of the default working directory.

```bash
$ advixe-cl --collect=survey --no-auto-finalize --project-dir=./advi --search-dir src:=./src -- ./bin/myApplication
```

**See Also**

Minimizing Analysis Overhead

Command Line Interface Reference
advixe-cl Command Action Reference

---

### benchmarks-sync

*Run benchmarks on only one concurrently executing Intel Advisor instance to avoid concurrency issues with regard to platform limits.*

**Syntax**

```bash
--benchmarks-sync
--no-benchmarks-sync
```
Default
On (benchmarks-sync)

Actions Modified
collect

Usage
Analyze a multi-process application running more than one process on the same host machine.

Example
Run a Trip Counts & FLOP analysis for the MPI application myApplication. Disable benchmark synchronization to get platform limits corresponding to concurrent runs of multiple processes on the same host machine.

$ mpirun -n 4 --gtool="advixe-cl --collect=tripcounts --flop --no-benchmarks-sync --project-dir=./advi:0-3" myApplication

See Also
Command Line Interface Reference
advixe-cl Command Action Reference

bottom-up
Generate a Survey report in bottom-up view.

GUI Equivalent
Survey > Loop Information

Syntax
--bottom-up
--no-bottom-up

Default
On (bottom-up)

Actions Modified
report = survey

Example
1. Run a Survey analysis.
2. Generate a Survey report. Show a bottom-up list of target loops/functions.

$ advixe-cl --collect=survey --project-dir=./advi -- myAplication
$ advixe-cl --report=survey --bottom-up --project-dir=./advi

See Also
Command Line Interface Reference
advixe-cl Command Action Reference

cache-binaries
Enable binary visibility in a read-only snapshot you can view any time.
GUI Equivalent

File > Create Data Snapshot > Cache Binaries

Syntax
--cache-binaries
--no-cache-binaries

Default
Off (no-cache-binaries)

Actions Modified
snapshot

Example
1. Run a Survey analysis.
2. Create a snapshot. Include binaries.

   $ advixe-cl --collect=survey --project-dir=./advi -- myApplication
   $ advixe-cl --snapshot --cache-binaries --project-dir=./advi

See Also
Command Line Interface Reference
advixe-cl Command Action Reference

cache-config
Set the cache hierarchy to collect modeling data for CPU cache behavior during Trip Counts & FLOP analysis.

GUI Equivalent

Project Properties > Analysis Target > Trip Counts and FLOP Analysis > Cache simulator configuration

Syntax
--cache-config=<string>

Arguments
<string> follows this template:
[num_of_level1_caches]:[num_of_ways_level1_connected]:[level1_cache_size]:[level1_cacheline_size]/
[num_of_level2_caches]:[num_of_ways_level2_connected]:[level2_cache_size]:[level2_cacheline_size]/
[num_of_level3_caches]:[num_of_ways_level3_connected]:[level3_cache_size]:[level3_cacheline_size]
For example: 4:8w:32k:64l/4:4w:256k:64l/1:16w:6m:64l

Actions Modified
collect=tripcounts --enable-cache-simulation
collect=roofline --enable-cache-simulation

Usage
When no specific configuration is set, the Intel Advisor uses system cache hierarchy for modeling.
NOTE
Cache simulation modeling applies to the following analyses:

- Memory Access Patterns analysis - This basic simulation functionality models accurate memory footprints, miss information, and cache line utilization for a downstream Memory Access Patterns report.
- Trip Counts and FLOP / Roofline - This enhanced simulation functionality models multiple levels of cache for a downstream Memory-Level Roofline chart or Roofline interactive HTML report.

This option is applicable only to Trip Counts and FLOP and Roofline analyses.

Example

1. Run a Survey analysis.

2. Run a Trip Counts & FLOP analysis. Model cache behavior for the specified configuration.

   $ advixe-cl --collect=survey --project-dir=./advi -- myApplication

   $ advixe-cl --collect=tripcounts --flop --enable-cache-simulation --cache-config=4:8w:32k:64l/4:4w:256k:64l/1:16w:6m:64l --project-dir=./advi -- myApplication

   Run Roofline analysis for all memory levels (Memory-Level Roofline) for the specified cache configuration.

   $ advixe-cl --collect=roofline --enable-cache-simulation --cache-config=4:8w:32k:64l/4:4w:256k:64l/1:16w:6m:64l --project-dir=./advi -- myApplication

See Also

cachesim-associativity  Set the cache associativity for modeling CPU cache behavior during Memory Access Patterns analysis.

cachesim-cacheline_size  Set the cache line size (in bytes) for modeling CPU cache behavior during Memory Access Patterns analysis.

cachesim-mode  Set the focus for modeling CPU cache behavior during Memory Access Patterns analysis.

cachesim-sets  Set the cache set size (in bytes) for modeling CPU cache behavior during Memory Access Patterns analysis.

enable-cache-simulation  Model CPU cache behavior on your target application.

Command Line Interface Reference
advixe-cl Command Action Reference

cache-sources
Enable source code visibility in a read-only snapshot you can view any time (with --snapshot action).
Enable keeping source code cache within a project (with --collect action).

GUI Equivalent

To add source to a snapshot: **File > Create Data Snapshot > Cache Sources**

To keep cached source during data collection: **Project Properties > Analysis Target > Survey Hotspots Analysis > Source caching**

Syntax

--cache-sources
--no-cache-sources
Default
Off (no-cache-sources)

Actions Modified
snapshot
collect

Usage
When used with collect action, report is supplied with source code folded like a snapshot. Once set, this option triggers a flag in project configuration that prevents deleting cache with each analysis run unless you manually disable it.

Example
Create a read-only snapshot. Include performance data, sources, and binaries. Save the snapshot to the tmp directory. Name the snapshot myAdvisorProjSnapshot.advixeexpz.

```bash
$ advixe-cl --snapshot --project-dir=./myAdvisorProj --pack --cache-sources --cache-binaries -- ./tmp/myAdvisorProjSnapshot
```

Run a Survey analysis and keep source cache.

```bash
$ advixe-cl --collect=survey --cache-sources --project-dir=./advi -- ./bin/myApplication
```

See Also
Command Line Interface Reference
advixe-cl Command Action Reference
cachesim-associativity
Set the cache associativity for modeling CPU cache behavior during Memory Access Patterns analysis.

GUI Equivalent
Project Properties > Analysis Target > Memory Access Patterns Analysis > Advanced > Cache associativity

Syntax
```bash
--cachesim-associativity=<integer>
```

Arguments

<integer> is the number of cache locations where one memory entry can be placed: 1 | 2 | 4 | 8 | 16

Default
8

Actions Modified
collect=map --enable cache-simulation

Usage
1 stands for a direct mapped cache, where a memory entry can occupy only one cache line.
NOTE
Cache simulation modeling applies to the following analyses:

- Memory Access Patterns analysis - This basic simulation functionality models accurate memory footprints, miss information, and cache line utilization for a downstream Memory Access Patterns report.
- Trip Counts and FLOP / Roofline - This enhanced simulation functionality models multiple levels of cache for a downstream Memory-Level Roofline chart or Roofline interactive HTML report.

This option is applicable only to Memory Access Patterns analysis.

Example
Run a Memory Access Patterns analysis. Model four-way associative cache with default cache line and cache set size.

```
$ advixe-cl --collect=map --enable-cache-simulation --cachesim-associativity=4 --cachesim-mode=utilization --project-dir=./myAdvisorProj -- ./myApp
```

See Also

- `cache-config` Set the cache hierarchy to collect modeling data for CPU cache behavior during Trip Counts & FLOP analysis.
- `cachesim-cacheline-size` Set the cache line size (in bytes) for modeling CPU cache behavior during Memory Access Patterns analysis.
- `cachesim-mode` Set the focus for modeling CPU cache behavior during Memory Access Patterns analysis.
- `cachesim-sets` Set the cache set size (in bytes) for modeling CPU cache behavior during Memory Access Patterns analysis.
- `enable-cache-simulation` Model CPU cache behavior on your target application.

Command Line Interface Reference

This reference section describes the CLI actions and options used in the command syntax: $ advixe-cl <!--action> [--action-options] [--global-options] [[--] target [target options]].

advixe-cl Command Action Reference

- `cachesim-cacheline-size`
  Set the cache line size (in bytes) for modeling CPU cache behavior during Memory Access Patterns analysis.

GUI Equivalent

Project Properties > Analysis Target > Memory Access Patterns > Advanced > Cache line size

Syntax

```
--cachesim-cacheline-size=<integer>
```

Arguments

`<integer>` is in bytes: 4 | 8 | 16 | 32 | 64 | 128 | 256 | 512 | 1024 | 2048 | 4096 | 8192 | 16384 | 32768 | 65536

Default

64
Actions Modified
`collect=map --enable cache-simulation`

Usage

**NOTE**
Cache simulation modeling applies to the following analyses:

- **Memory Access Patterns analysis** - This basic simulation functionality models accurate memory footprints, miss information, and cache line utilization for a downstream Memory Access Patterns report.
- **Trip Counts and FLOP / Roofline** - This enhanced simulation functionality models multiple levels of cache for a downstream Memory-Level Roofline chart or Roofline interactive HTML report.

This option is applicable only to Memory Access Patterns analysis.

Example
Run a Memory Access Patterns analysis. Model four-way associative cache with 64-byte cache line size and default cache set size.

```shell
$ advixe-cl --collect=map --enable-cache-simulation --cachesim-cacheline-size=64 --cachesim-associativity=4 --cachesim-mode=utilization --project-dir=./myAdvisorProj -- ./myApp
```

See Also
- `cache-config` Set the cache hierarchy to collect modeling data for CPU cache behavior during Trip Counts & FLOP analysis.
- `cachesim-associativity` Set the cache associativity for modeling CPU cache behavior during Memory Access Patterns analysis.
- `cachesim-mode` Set the focus for modeling CPU cache behavior during Memory Access Patterns analysis.
- `cachesim-sets` Set the cache set size (in bytes) for modeling CPU cache behavior during Memory Access Patterns analysis.
- `enable-cache-simulation` Model CPU cache behavior on your target application.

Command Line Interface Reference
- `advixe-cl Command Action Reference`

**cachesim-mode**
Set the focus for modeling CPU cache behavior during Memory Access Patterns analysis.

GUI Equivalent
**Project Properties > Analysis Target > Memory Access Patterns Analysis > Advanced > Cache simulation mode**

Syntax
`--cachesim-mode=<string>`

Arguments
`<string>` is one of the following:
<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>cache-misses</td>
<td>Model cache misses only.</td>
</tr>
<tr>
<td>footprint</td>
<td>Model cache misses and memory footprint of a loop. Calculation: Cache line size x Number of unique cache lines accessed during simulation.</td>
</tr>
<tr>
<td>utilization</td>
<td>Model cache misses and cache lines utilization.</td>
</tr>
</tbody>
</table>

**Default**

utilization

**Actions Modified**

```
collect=map --enable cache-simulation
```

**Usage**

For memory footprint simulation, the Intel Advisor tracks only a subset of accesses and cache lines, and scales it up to the total size of cache to calculate the final footprint value.

**NOTE**

Cache simulation modeling applies to the following analyses:

- Memory Access Patterns analysis - This basic simulation functionality models accurate memory footprints, miss information, and cache line utilization for a downstream Memory Access Patterns report.
- Trip Counts and FLOP / Roofline - This enhanced simulation functionality models multiple levels of cache for a downstream Memory-Level Roofline chart or Roofline interactive HTML report.

This option is applicable only to Memory Access Patterns analysis.

**Tip**

Usage can increase analysis overhead.

**Example**

Run a Memory Access Patterns analysis. Model cache misses for a default cache configuration.

```
$ advixe-cl collect=map --enable-cache-simulation --cachesim-mode=cache-misses --project-dir=./myAdvisorProj -- ./myApp
```

Run a Memory Access Patterns analysis. Model cache miss and memory footprint data for 1024-byte cache set size, default cache associativity and cache line size.

```
$ advixe-cl collect=map --enable-cache-simulation --cachesim-sets=1024 --cachesim-mode=footprint --project-dir=./myAdvisorProj -- ./myApp
```

**See Also**

- `cache-config` Set the cache hierarchy to collect modeling data for CPU cache behavior during Trip Counts & FLOP analysis.
- `cachesim-associativity` Set the cache associativity for modeling CPU cache behavior during Memory Access Patterns analysis.
cachesim-cacheline_size Set the cache line size (in bytes) for modeling CPU cache behavior during Memory Access Patterns analysis.
cachesim-sets Set the cache set size (in bytes) for modeling CPU cache behavior during Memory Access Patterns analysis.
enable-cache-simulation Model CPU cache behavior on your target application.

Command Line Interface Reference
advixe-cl Command Action Reference

cachesim-sets
Set the cache set size (in bytes) for modeling CPU cache behavior during Memory Access Patterns analysis.

GUI Equivalent
Project Properties > Analysis Target > Memory Access Patterns Analysis > Advanced > Cache sets

Syntax
--cachesim-sets=<integer>

Arguments
<integer> is in bytes: 256 | 512 | 1024 | 2048 | 4096 | 8192

Default
4096

Actions Modified
collect=map --enable cache-simulation

Usage

NOTE
Cache simulation modeling applies to the following analyses:

- Memory Access Patterns analysis - This basic simulation functionality models accurate memory footprints, miss information, and cache line utilization for a downstream Memory Access Patterns report.
- Trip Counts and FLOP / Roofline - This enhanced simulation functionality models multiple levels of cache for a downstream Memory-Level Roofline chart or Roofline interactive HTML report.

This option is applicable only to Memory Access Patterns analysis.

Example

Run a Memory Access Patterns analysis. Model cache misses for 2048-byte cache set size, default cache associativity and cache line size.

```bash
$ advixe-cl collect=map --enable-cache-simulation --cachesim-sets=2048 --cachesim-mode=cache-misses --project-dir=./myAdvisorProj -- ./myApp
```

See Also
cache-config Set the cache hierarchy to collect modeling data for CPU cache behavior during Trip Counts & FLOP analysis.
cachesim-associativity Set the cache associativity for modeling CPU cache behavior during Memory Access Patterns analysis.
cachesim-cacheline_size  Set the cache line size (in bytes) for modeling CPU cache behavior during Memory Access Patterns analysis.
cachesim-mode Set the focus for modeling CPU cache behavior during Memory Access Patterns analysis.
enable-cache-simulation  Model CPU cache behavior on your target application.

Command Line Interface Reference
advixe-cl Command Action Reference

csv-delimiter
Set the delimiter for a report in CSV format.

Syntax

```
--csv-delimiter=<string>
```

Arguments

<string> is one of the following: comma | semicolon | tab

Default
comma

Actions Modified

report=[report type] --format=csv

Example

Generate a Dependencies report. Output in CSV format with tab delimiters.

```
$ advixe-cl --report=dependencies --project-dir=./advi --format=csv --csv-delimiter=tab --report-output=./out/advisor-Dependencies.csv
```

See Also

Command Line Interface Reference
advixe-cl Command Action Reference

data-limit
Limit the maximum amount (in MB) of raw data collected during Survey analysis.

GUI Equivalent

Project Properties  >  Analysis Target  >  Survey Hotspots Analysis  >  Advanced  >  Collection data limit

Syntax

```
--data-limit=<integer>
```

Arguments

<integer> is the maximum collection size, in MB.

Default
500

Actions Modified

collect=survey
Usage
This option is useful if you have storage space limitations. A smaller value can also decrease collection overhead.

Example
Run a Survey analysis. Stop data collection when a 250-MB limit is reached or upon normal completion.

```
$ advixe-cl --collect=survey --project-dir=./myAdvisorProj --data-limit=250 -- ./bin/myTargetApplication
```

See Also
Minimizing Analysis Overhead
Command Line Interface Reference
advixe-cl Command Action Reference
data-transfer-page-size (Beta)
Specify memory page size to set the traffic measurement granularity for the data transfer simulator.

Syntax
```
--data-transfer-page-size=<integer>
```

Arguments

\(<\text{integer}>\) is a power-of-two value in range of 4 to 8192: 4 | 8 | 16 | 32 | 64 | 128 | 256 | 512 | 1024 | 2048 | 4096 | 8192

Default
4096

Actions Modified
collect=tripcounts --enable-data-transfer-analysis

Usage

**Important**
This option is available only in the Intel® Advisor Beta, which is part of Intel® oneAPI Base Toolkit. For more information, see https://software.intel.com/content/www/us/en/develop/tools/oneapi/components/advisor.html.

Example
Run a Trip Counts and FLOP analysis. Enable data transfer simulation with 512-bites memory page size.

```
$ advixe-cl --collect=tripcounts --project-dir=./advi --flop --enable-data-transfer-analysis --data-transfer-page-size=512 -- myApplication
```

See Also
eable-data-transfer-analysis Model data transfer between host memory and device memory.
Command Line Interface Reference
advixe-cl Command Action Reference
data-type
Show only floating-point data, only integer data, or data for the sum of both data types in a Roofline interactive HTML report.

GUI Equivalent
Roofline > Default: FLOAT > Operations

Syntax
--data-type=<string>

Arguments
<string> is one of the following: float | int | mixed

Default
float

Actions Modified
report=roofline

Example
Generate a Roofline interactive HTML report. Include floating-point data; exclude integer data.

$ advixe-cl --report=roofline --project-dir=./myAdvisorProj --data-type=float

See Also
Command Line Interface Reference
advixe-cl Command Action Reference
delete-tripcounts
Remove previously collected trip counts data when re-running a Survey analysis with changed binaries.

GUI Equivalent
Warning message

Syntax
--delete-tripcounts
--no-delete-tripcounts

Default
On (delete-tripcounts)

Actions Modified
collect=survey

Usage
Enable to eliminate the risk of including out-of-date data in a new Survey analysis if you:

• Change binaries.
• Have previously collected trip counts/FLOP data.

In other cases, this option is ignored.
Example
Run a Survey analysis. Remove previously collected trip counts data during analysis.

$ advixe-cl --collect=survey --project-dir=./advi --search-dir src:=./src bin:=./bin --delete-tripcounts

See Also
Command Line Interface Reference
advixe-cl Command Action Reference
display-callstack
Show a callstack for each loop/function call in a report.

Syntax
--display-callstack

Actions Modified
report

Example
Generate a Suitability report. Include callstack data.

$ advixe-cl --report=suitability --project-dir=./advi --display-callstack

See Also
Command Line Interface Reference
advixe-cl Command Action Reference
duration
Specify the maximum amount of time (in seconds) an analysis runs.

Syntax
-d=<string>
--duration=<string>

Arguments
<string> is maximum number of seconds or unlimited.

Default
unlimited

Actions Modified
collect

Usage
The target application also stops executing when the analysis stops running.
Example

Stop a Survey analysis after 60 seconds.

```
$ advixe-cl --collect=survey --project-dir=./advi --duration=60
```

See Also

Command Line Interface Reference
advixe-cl Command Action Reference

dynamic

Show (in a Survey report) how many instructions of a given type actually executed during Trip Counts & FLOP analysis.

GUI Equivalent

Code Analytics

Syntax

--dynamic

--no-dynamic

Default

On (dynamic)

Actions Modified

report=survey

Usage

Dynamic instruction mix is counted for the entire execution of the application; static instruction mix is counted per iteration. The static-instruction-mix, dynamic, and mix options work together in the following manner:

- Collect static instruction mix data: `--collect=survey --static-instruction-mix`
  
  (In the GUI: Static instruction mix data is calculated on demand.)
- Collect dynamic instruction mix data (and static instruction mix data, from which dynamic mix data is calculated): `--collect=tripcounts --flop`
- Show static instruction mix data in a Survey report: `--report=survey --mix --no-dynamic`
- Show dynamic mix instruction data in a Survey report: `--report=survey --mix --dynamic`
- A Survey report cannot show both static and dynamic mix instruction data.

  (In the GUI: **Code Analytics** can show both static and dynamic instruction mix data.)

Example

1. Run a Survey analysis.
2. Run a Trip Counts & FLOP analysis. Collect dynamic instruction mix data (and static instruction mix data, from which dynamic mix data is calculated).
3. Generate a Survey report. Show dynamic instruction mix data. (**dynamic** is on, by default).

```
$ advixe-cl --collect=survey --project-dir=./advi -- ./bin/myTargetApplication
$ advixe-cl --collect=tripcounts --flop --project-dir=./advi
$ advixe-cl --report=survey --mix --project-dir=./advi
```

1. Run a Survey analysis. Collect static instruction mix data.
2. Generate a Survey report. Show static instruction mix data.

   $ advixe-cl --collect=survey --static-instruction-mix --project-dir=./advi -- ./bin/myTargetApplication
   $ advixe-cl --report=survey --mix --no-dynamic --project-dir=./advi

See Also

mix  Show dynamic or static instruction mix data in a Survey report.
static-instruction-mix  Statically calculate the number of specific instructions present in the binary during Survey analysis.
Command Line Interface Reference
advixe-cl Command Action Reference

enable-cache-simulation
Model CPU cache behavior on your target application.

GUI Equivalent

For basic modeling functionality: Project Properties > Analysis Target > Memory Access Patterns Analysis > Advanced > Enable CPU cache simulation

For enhanced modeling functionality:

Project Properties > Analysis Target > Trip Counts and FLOP Analysis > Enable CPU cache simulation or
Workflow > Run Roofline > For All Memory Levels

Syntax

   --enable-cache-simulation
   --no-enable-cache-simulation

Default

Off (no-enable-cache-simulation)

Actions Modified

collect=map
collect=tripcounts
collect=roofline

Usage

Enabling can increase collection overhead.

NOTE
Cache simulation modeling applies to the following analyses:

- Memory Access Patterns analysis - This basic simulation functionality models accurate memory footprints, miss information, and cache line utilization for a downstream Memory Access Patterns report.
- Trip Counts and FLOP / Roofline - This enhanced simulation functionality models multiple levels of cache for a downstream Memory-Level Roofline chart or Roofline interactive HTML report.
**Example**

Run a Memory Access Patterns analysis. Enable cache simulation with basic functionality and default cache parameters to collect cache modeling data for a downstream Memory Access Patterns report.

```bash
$ advixe-cl --collect=map --project-dir=./advi --enable-cache-simulation -- myApplication
```

Run a Roofline analysis for all memory levels.

```bash
$ advixe-cl --collect=roofline --enable-cache-simulation --project-dir=./advi -- myApplication
```

**See Also**

- **cache-config** Set the cache hierarchy to collect modeling data for CPU cache behavior during Trip Counts & FLOP analysis.
- **cachesim-associativity** Set the cache associativity for modeling CPU cache behavior during Memory Access Patterns analysis.
- **cachesim-cacheline_size** Set the cache line size (in bytes) for modeling CPU cache behavior during Memory Access Patterns analysis.
- **cachesim-mode** Set the focus for modeling CPU cache behavior during Memory Access Patterns analysis.
- **cachesim-sets** Set the cache set size (in bytes) for modeling CPU cache behavior during Memory Access Patterns analysis.

**Command Line Interface Reference**

- advixe-cl Command Action Reference

**enable-data-transfer-analysis (Beta)**

*Model data transfer between host memory and device memory.*

**Syntax**

```bash
--enable-data-transfer-analysis
--no-enable-data-transfer-analysis
```

**Default**

Off (no-enable-data-transfer-analysis)

**Actions Modified**

- `collect=tripcounts`

**Usage**

**Important**

This option is available only in the Intel® Advisor Beta, which is part of Intel® oneAPI Base Toolkit. For more information, see [https://software.intel.com/content/www/us/en/develop/tools/oneapi/components/advisor.html](https://software.intel.com/content/www/us/en/develop/tools/oneapi/components/advisor.html).

**Example**

Run a Trip Counts and FLOP analysis. Enable data transfer simulation.

```bash
$ advixe-cl --collect=tripcounts --project-dir=./advi --flop --enable-data-transfer-analysis -- myApplication
```
See Also

data-transfer-page-size (Beta) Specify memory page size to set the traffic measurement granularity for the data transfer simulator.

track-heap-objects (Beta) Attribute heap-allocated objects to the analyzed loops that accessed the objects.

track-stack-accesses (Beta) Track accesses to stack memory.

Command Line Interface Reference

enable-task-chunking

Examine specified annotated sites for opportunities to perform task-chunking modeling in a Suitability report.

GUI Equivalent

Suitability > Enable Task Chunking

Syntax

--enable-task-chunking=<string>

Arguments

<string> is a comma-separated list of annotated sites (no spaces).

Default

No default argument

Actions Modified

report=suitability

Example

Generate a Suitability report. Include task-chunking analysis for the annotated sites myAnnotatedSiteJ and myAnnotatedSiteX.

$ advixe-cl --report=suitability --project-dir=./adv --enable-task-chunking=myAnnotatedSiteJ,myAnnotatedSiteX

See Also

Enabling Task Chunking

Command Line Interface Reference

advixe-cl Command Action Reference

exclude-files

Exclude the specified files or directories from annotation scanning during analysis.

GUI Equivalent

Project Properties > Source Search > Exclude the following files

Syntax

--exclude-files=<PATH>

Arguments

<Path> is a comma-separated list of file paths or directories to exclude during data collection (no spaces).
**Default**
No default argument

**Actions Modified**
collect

**Example**
Run a Suitability analysis. Exclude all files in the two specified source directories from annotation scanning during analysis.

```bash
$ advixe-cl --collect=suitability --project-dir=./advi --exclude-files=./src1,./src2 --
myApplication
```

**See Also**
Command Line Interface Reference
advixe-cl Command Action Reference

**executable-of-interest**
*Specify an application for analysis that is not the starting application.*

**GUI Equivalent**
**Project Properties > Analysis Target > [Analysis Type] > Child Application**

**Syntax**
```bash
--executable-of-interest=<PATH>
```

**Arguments**
<br>\(<PATH>\) is a string specifying the PATH/name of the executable to be analyzed.

**Actions Modified**
collect

**Usage**
Specify an executable child process to analyze, instead of the script or application that spawns the child process.

**Example**
Run a Survey analysis launched from `myScript`. Specify `myApplication` as the executable of interest.

```bash
$ advixe-cl --collect=survey --project-dir=./advi --executable-of-interest=myApplication.exe --
myScript
```

**See Also**
Command Line Interface Reference
advixe-cl Command Action Reference

**filter**
*Filter data by the specified column name and value in a Survey and Trips Counts & FLOP report.*

**GUI Equivalent**
**Filters**
Syntax
--filter=<string>

Arguments
<string> is in the following format: "[column name]"="[value]".

Default
No default argument

Actions Modified
report = survey
report = tripcounts

Example
Generate a Survey report. Show the top five self-time hotspots that were not vectorized because of a not inner loop msg id.

```
$ advixe-cl --report=survey --limit=5 --filter="Vectorization Message(s)"="loop was not vectorized: not inner loop"
```

See Also
Command Line Interface Reference
advixe-cl Command Action Reference

filter-by-scope
Enable filtering detected stack variables by scope (warning vs. error) in a Dependencies analysis.

GUI Equivalent
Project Properties > Analysis Target > Dependencies Analysis > Advanced > Filter stack variables by scope

Syntax
--filter-by-scope
--no-filter-by-scope

Default
On (filter-by-scope)

Actions Modified
collect=dependencies

Usage
Variables initiated inside the specified loop(s) are considered potential dependencies (warning). Variables initiated outside the specified loop(s) are considered dependencies (error).

Disabling can decrease collection overhead.
**Example**

Run a Dependencies analysis. Analyze innermost scalar loops. Filter detected stack variables by scope.

```
advixe-cl --collect=dependencies --loops="scalar,loop-height=0" --filter-by-scope --project-dir=./myAdvisorProj -- ./bin/myTargetApplication
```

**See Also**

Command Line Interface Reference
advixe-cl Command Action Reference

**filter-reductions**

Mark all potential reductions by specific diagnostic during Dependencies analysis.

**GUI Equivalent**

*Project Properties > Analysis Target > Dependencies Analysis > Advanced > Filter reduction variables*

**Syntax**

```
--filter-reductions
--no-filter-reductions
```

**Default**

Off (no-filter-reductions)

**Actions Modified**

```
collect=dependencies
```

**Usage**

Enabling can increase collection overhead.

**Example**

Run a Dependencies analysis. Analyze innermost scalar loops. Mark all potential reductions by specific diagnostic.

```
advixe-cl -collect dependencies --loops=scalar,loop-height=0" --filter-reductions --project-dir=./myAdvisorProj -- ./bin/myTargetApplication
```

**See Also**

Command Line Interface Reference
advixe-cl Command Action Reference

**flop**

Collect data about floating-point and integer operations, memory traffic, and mask utilization metrics for AVX-512 platforms during Trip Counts & FLOP analysis.

**GUI Equivalent**

*Workflow and Toolbar*

**Syntax**

```
--flop
```
--no-flop

**Default**

Off (no-flop)

**Actions Modified**

collect=tripcounts

**Usage**

Enabling can increase analysis overhead.

**Example**

Run the Trip Counts & FLOP analysis. Collect both Trip Counts and FLOP data. (Default for the trip-counts option is on, so it is not explicitly stated in the command line.)

```
$ advixe-cl -collect=tripcounts --flop --project-dir=./myAdvisorProj -- ./bin/myTargetApplication
```

**See Also**

Command Line Interface Reference
advixe-cl Command Action Reference

**format**

*Set a report output format.*

**Syntax**

```
--format=<string>
```

**Arguments**

<string> is one of the following:

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>csv</td>
<td>Tabular (.csv) file format</td>
</tr>
<tr>
<td>text</td>
<td>.txt file format</td>
</tr>
<tr>
<td>xml</td>
<td>.xml file format</td>
</tr>
</tbody>
</table>

**Default**

text

**Actions Modified**

report

**Usage**

By default, the advixe-cl writes a report to standard output in text format; however, it provides a number of options for generating a report:

- Use the report-output option to write a report to a file.
- Use the csv-delimiter option to set a delimiter other than a comma.
Example

Generate a Dependencies report. Output in XML format. Save it as `advisor-Dependencies.xml`.

```bash
$ advixe-cl --report=dependencies --project-dir=./advi --format=xml --report-output=./out/advisor-Dependencies.xml
```

Generate a Dependencies report. Output in CSV format with tab delimiters. Save it as `advisor-Dependencies.csv`.

```bash
$ advixe-cl --report=dependencies --project-dir=./advi --format=csv --csv-delimiter=tab --report-output=./out/advisor-Dependencies.csv
```

See Also

csv-delimiter  Set the delimiter for a report in CSV format.
report-output  Redirect report output from stdout to another location.

Command Line Interface Reference
advixe-cl Command Action Reference

gpu (Beta)
Create a Roofline interactive HTML report for data collected on GPUs.

Syntax

```bash
--gpu
```

Actions Modified

```
report=roofline
```

Usage

**Important**
This option is available only in the Intel® Advisor Beta, which is part of Intel® oneAPI Base Toolkit. For more information, see https://software.intel.com/content/www/us/en/develop/tools/oneapi/components/advisor.html.

**Prerequisites:** Collect Roofline data with `--profile-gpu` option enabled.

The default Roofline interactive HTML report is created for data collected on CPUs. To create a report for data collected on GPUs, use this option.

**Example**

Generate a Roofline interactive HTML report for data collected on GPUs.

```bash
$ advixe-cl --report=roofline --project-dir=./advi --report-output=./out/roofline.html --gpu
```

See Also

Command Line Interface Reference
advixe-cl Command Action Reference

gpu-carm (Beta)
Collect memory traffic generated by OpenCL™ and Intel® Media SDK programs executed on Intel® Processor Graphics.
Syntax
--gpu-carm
--no-gpu-carm

Default
On (gpu-carm)

Actions Modified
collect=tripcounts --profile-gpu
collect=roofline --profile-gpu

Usage

Important
• This option is available only in the Intel® Advisor Beta, which is part of Intel® oneAPI Base Toolkit. For more information, see https://software.intel.com/content/www/us/en/develop/tools/oneapi/components/advisor.html.
• GPU profiling is applicable only to Intel® Processor Graphics.

This option may affect the performance of your application on the CPU side.

Example
1. Run a Survey analysis with GPU profiling enabled.
   $ advixe-cl --collect=survey --project-dir=./advi --profile-gpu -- myApplication
2. Run a Trip Count and FLOP analysis, enable GPU profiling and explicitly enable CARM traffic metrics collection.
   $ advixe-cl --collect=tripcounts --flop --profile-gpu --gpu-carm --project-dir=./advi -- myApplication

See Also
Command Line Interface Reference
advixe-cl Command Action Reference

gpu-sampling-interval (Beta)
Specify time interval, in milliseconds, between GPU samples during Survey analysis.

Syntax
--gpu-sampling-interval=<double>

Arguments
<double> is the number from 0.1 to 10 milliseconds between GPU samples.

Default
1

Actions Modified
collect=survey --profile-gpu
```bash
collect=roofline --profile-gpu
```

**Usage**

**Important**

- This option is available only in the Intel® Advisor Beta, which is part of Intel® oneAPI Base Toolkit. For more information, see https://software.intel.com/content/www/us/en/develop/tools/oneapi/components/advisor.html.
- GPU profiling is applicable only to Intel® Processor Graphics.

This option may affect the performance of your application on the CPU side. Increasing the wait time between samples can decrease collection overhead.

**Example**

Run a Survey analysis. Enable GPU profiling and increase the GPU sampling interval to 3 ms.

```bash
$ advixe-cl --collect=survey --project-dir=./advi --profile-gpu --gpu-sampling-interval=3 -- myApplication
```

**See Also**

Command Line Interface Reference
advixe-cl Command Action Reference

`ignore-app-mismatch`

*Ignore mismatched target or application parameter errors before starting analysis.*

**GUI Equivalent**

**Survey** > **Your Project Properties changed since the last analysis run**

**error**

(Click **Continue** or **Cancel**)

**Syntax**

`--ignore-app-mismatch`

`--no-ignore-app-mismatch`

**Default**

*Off (no-ignore-app-mismatch)*

**Actions Modified**

`collect`

**Usage**

If you continue collection, trip counts data might become unreliable. To synchronize trip counts and survey data:

1. Run a Survey analysis for the updated parameters using `delete-tripcounts`.
2. Run a Trip Counts analysis.

**Example**

Run a Survey analysis. Ignore mismatched target or application parameter errors before starting analysis.

```bash
$ advixe-cl --collect=survey --ignore-app-mismatch --project-dir=./advi -- ./bin/myApplication
```
See Also
Command Line Interface Reference
advixe-cl Command Action Reference

ignore-checksums
Ignore mismatched module checksums before starting analysis.

GUI Equivalent
Survey > Target binaries changed since the last analysis run error
(Click Continue or Cancel)

Syntax
--ignore-checksums
--no-ignore-checksums

Default
Off (no-ignore-checksums)

Actions Modified
collect

Usage
If you continue collection, trip counts data might become unreliable. To synchronize trip counts and survey data:

1. Run a Survey analysis for the updated binaries using delete-tripcounts.
2. Run a Trip Counts analysis.

Example
Run a Survey analysis. Ignore mismatched module checksums before starting analysis.

```
$ advixe-cl --collect=survey --ignore-checksums --project-dir=./advi -- ./bin/myApplication
```

See Also
Command Line Interface Reference
advixe-cl Command Action Reference

instance-of-interest
Analyze the Nth child process during Memory Access Patterns and Dependencies analysis.

GUI Equivalent
Project Properties > [Analysis Type] > Advanced > Instance of interest

Syntax
--instance-of-interest=<integer>
Arguments

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Analyze all processes.</td>
</tr>
<tr>
<td>1</td>
<td>Analyze first process of the specified name in the application process tree.</td>
</tr>
<tr>
<td>&lt;integer&gt;</td>
<td>Analyze subsequent process of the specified name in the application process tree.</td>
</tr>
</tbody>
</table>

Default

0

Actions Modified

collect=map
collect=dependencies

Example

Run a Memory Access Patterns analysis. Analyze the first process of the specified name in the application process tree.

$ advixe-cl --collect=map --instance-of-interest=1 --project-dir=./advi -- ./bin/myApplication

See Also

Command Line Interface Reference
advixe-cl Command Action Reference

integrated

Model traffic on all levels of the memory hierarchy for a Roofline report.

GUI Equivalent

Workflow > Run Roofline > For All Memory Levels

Syntax

--integrated
--no-integrated

Default

Off (no-integrated)

Actions Modified

collect=roofline

Example

Run a Roofline analysis. Model traffic on all levels of the memory hierarchy.

$ advixe-cl --collect=roofline --integrated --project-dir=./advi -- ./bin/myApplication

See Also

Command Line Interface Reference
advixe-cl Command Action Reference
**interval**
Set the length of time (in milliseconds) to wait before collecting each sample during Survey analysis.

**GUI Equivalent**
Project Properties > Analysis Target > Survey Hotspots Analysis > Advanced > Sampling interval

**Syntax**
```
--interval=<integer>
```

**Arguments**

<integer> is the number of milliseconds between sampling (sampling interval).

**Default**
10

**Actions Modified**
collect=survey

**Usage**
Increasing the wait time between each analysis collection sample can decrease collection overhead.

**Example**
Run a Survey analysis. Increase the sampling interval to 20 milliseconds.

```
$ advixe-cl --collect=survey --project-dir=./myAdvisorProj --interval=20 -- ./bin/myTargetApplication
```

**See Also**
Minimizing Analysis Overhead
Command Line Interface Reference
advixe-cl Command Action Reference

---

**limit**
Set the maximum number of top items to show in a report.

**GUI Equivalent**
Survey > Customize View > Top

**Syntax**
```
--limit=<integer>
```

**Arguments**

<integer> is the maximum number of top items.

**Default**
Off (unlimited lines in output report)

**Actions Modified**
report
Example
Generate a Survey report. Show the top five self-time hotspots that were not vectorized because of a *not inner loop* msg id.

```
$ advixe-cl --report=survey --limit=5 --filter="Vectorization Message(s)"="loop was not vectorized: not inner loop"
```

See Also
Command Line Interface Reference
advixe-cl Command Action Reference

**loop-call-count-limit**
Set the maximum number of instances to analyze for all marked loops.

GUI Equivalent

Project Properties > Analysis Target > [Analysis Type] > Advanced > Loop Call Count Limit

Syntax
```
--loop-call-count-limit=<integer>
```

Arguments

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Analyze all instances of all marked loops.</td>
</tr>
<tr>
<td>&lt;integer&gt;</td>
<td>Analyze up to <em>n</em> number of instances for all marked loops.</td>
</tr>
</tbody>
</table>

Default

0 (analyze all instances of all marked loops)

**Actions Modified**

- `collect=dependencies`
- `collect=map`

Usage

Assumes similar runtime properties, such as the same memory access patterns, over different call instances. If this is not true, using this option may produce skewed results.

A smaller, non-zero value can minimize collection overhead.

Example
Run a Memory Access Patterns analysis. Limit analysis to the first ten invocations of marked loops.

```
$ advixe-cl --collect=map --loop-call-count-limit=10 --project-dir=./new-advi -- ./bin/myApplication
```

See Also
Minimizing Analysis Overhead

Command Line Interface Reference
advixe-cl Command Action Reference
loops
Select loops (by criteria instead of human input) for deeper analysis.

GUI Equivalent
Workflow > Batch mode > Check Memory Access Patterns
Workflow > Batch mode > Check Dependencies

Syntax
--loops=<string>

Arguments
<string> is a double-quote-enclosed, comma-separated list of criteria (no spaces):

<table>
<thead>
<tr>
<th>Arguments for Dependencies Analysis</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>scalar</td>
<td>Include only scalar serial loops.</td>
</tr>
<tr>
<td>total-time&gt;n</td>
<td>Include only loops above n% of total CPU time.</td>
</tr>
<tr>
<td>has-source</td>
<td>Exclude only loops without source location.</td>
</tr>
<tr>
<td>has-issue</td>
<td>Include only loops with Vector Dependence Prevents Vectorization issue.</td>
</tr>
<tr>
<td>loop-height=n</td>
<td>Include only loops by specific hierarchical position. 0 = Innermost loops. For example: Use the following criteria to include specific loops in the following scenario:</td>
</tr>
<tr>
<td></td>
<td>• loop-height=0 selects only innermost loop 3</td>
</tr>
<tr>
<td></td>
<td>• loop-height=1 selects only loop 2 and loop 3</td>
</tr>
<tr>
<td></td>
<td>• loop-height=2 selects all three loops</td>
</tr>
<tr>
<td>top=n</td>
<td>Include only loops with the largest self-time.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Arguments for Memory Access Patterns Analysis</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>total-time&gt;n</td>
<td>Include only loops above n% of total CPU time.</td>
</tr>
<tr>
<td>has-source</td>
<td>Exclude only loops without source location.</td>
</tr>
<tr>
<td>has-issue</td>
<td>Include only loops with Possible Inefficient Memory Access Pattern issue.</td>
</tr>
<tr>
<td>loop-height=n</td>
<td>Include only loops by specific hierarchical position. 0 = Innermost loops. For example: Use the following criteria to include specific loops in the following scenario:</td>
</tr>
</tbody>
</table>
Arguments for Memory Access Patterns Analysis

<table>
<thead>
<tr>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>• loop-height=0 selects only innermost loop 3</td>
</tr>
<tr>
<td>• loop-height=1 selects only loop 2 and loop 3</td>
</tr>
<tr>
<td>• loop-height=2 selects all three loops</td>
</tr>
</tbody>
</table>

```
main
| -> loop 1
| -> loop 2
| -> loop 3
```

| top=n | Include only loops with the largest self-time. |

**Default**

For Memory Access Patterns analysis: "loop-height=0,total-time>0.1"

For Dependencies analysis: "scalar,loop-height=0,total-time>0.1"

**Actions Modified**

- `collect=map`
- `collect=dependencies`
- `mark-up-loops`

**Usage**

All criteria must be met to select loops for analysis.

This option is particularly useful when you are using automated scripts.

You can accomplish the same objective using the `mark-up-loops` action followed by a `collect` action for a Dependencies or Memory Access Patterns analysis.

Usage can decrease collection overhead.

**Example**

Run a Dependencies analysis. Analyze only innermost loops with sources.

```
$ advixe-cl --collect=dependencies --loops="loop-height=0,has-source" --project-dir=./advi --search-dir src:=./src -- ./bin/myApplication
```

**See Also**

- `mark-up-loops` After running a Survey analysis and identifying loops of interest, select loops (by file and line number or criteria) for deeper analysis.

**Minimizing Analysis Overhead**

**Command Line Interface Reference**

- `advixe-cl` Command Action Reference

**mark-up**

`Enable/disable user selection as a way to control loops/functions identified for deeper analysis.`
Syntax
--mark-up
--no-mark-up

Default
On (mark-up)

Actions Modified
collect=dependencies
collect=map

Usage
Intel Advisor offers two ways to identify loops/functions for deeper analysis:

• **Source annotations** - Via code that pinpoints the start and end of loops and iterations
  This is the only identification method for the Suitability analysis.

• **User selection** - In the CLI via `mark-up-loops` and `mark-up-list`, and in the GUI via `Survey >`
  This is the primary identification method for Vectorization Advisor analyses.

  User selection via `mark-up-loops` and `Survey >` persists for downstream analyses. User selection via `mark-up-list` persists only for the duration of a `collect` action.

Use `mark-up` to use both ways to identify loops/functions for deeper analysis. Use `no-mark-up` to use only source annotations.

**NOTE**
There is no `clear selection` option in the Intel Advisor CLI. `no-mark-up` is the closest equivalent if user-selected loops/functions persist but you want to use only source annotations to identify loops/functions for deeper analysis.

Example
Run a Memory Access Patterns analysis. Analyze only loops/functions identified by source annotations.

```
$ advixe-cl --collect=map --no-mark-up --project-dir=./advi -- ./bin/myApplication
```

See Also
Command Line Interface Reference
advixe-cl Command Action Reference

mark-up-list
After running a Survey analysis and identifying loops of interest, select loops (by file and line number or ID) for deeper analysis.

GUI Equivalent

**Survey >**

**Syntax**

--mark-up-list=<string>
Arguments

<string> is a comma-separated list (no spaces) of loop IDs, file/line numbers in the format file1:line1, or both.

Default

The existing selection in Survey > , which is persistent.

If there is no GUI selection:

- For Trip Counts & FLOP and Roofline analyses: all loops
- loops default
  - For Memory Access Patterns analysis: "loop-height=0,total-time>0.1"
  - For Dependencies analysis: "scalar,loop-height=0,total-time>0.1"

Actions Modified

collect=tripcounts
collect=map
collect=dependencies
collect=roofline

Usage

Do not confuse the mark-up-loops action with the mark-up-list action option. The mark-up-loops action coupled with the select action option enables a GUI checkbox; therefore loop selection persists beyond the duration of the mark-up-loops action and applies to downstream analyses, such as Dependencies and Memory Access Patterns analyses. The collect action coupled with the mark-up-list action option simulates enabling a GUI checkbox; therefore loop selection persists only for the duration of the collect action.

Example

1. Run a Survey analysis.
2. Run a Trip Counts & FLOP analysis on Survey analysis loops 1 and 3; and source location my_source.cpp:132.

   $ advixe-cl --collect=survey --project-dir=./myAdvisorProj -- ./bin/myTargetApplication
   $ advixe-cl --collect=tripcounts --project-dir=./advi --mark-up-list=1,my_source.cpp:132,3 -- search-dir src:=./src -- myApplication

See Also

loops  Select loops (by criteria instead of human input) for deeper analysis.
mark-up-loops  After running a Survey analysis and identifying loops of interest, select loops (by file and line number or criteria) for deeper analysis.

Minimizing Analysis Overhead

Command Line Interface Reference
advixe-cl Command Action Reference

memory-level
Model specific memory level(s) in a Roofline interactive HTML report, including L1, L2, L3, and DRAM.
GUI Equivalent
Roofline > Default: FLOAT CARM (L1+NTS) > Memory Level

Syntax
--memory-level=<string>

Arguments
<string> is an underscore-separated list of memory levels (no spaces).

Default
L1 (subject to change)

Actions Modified
report=roofline

Usage
Enable pinpointing memory bandwidth bottlenecks by specific memory layer.

Example
Generate a Roofline interactive HTML report. Show data for L2 and L3 memory.

$ advixe-cl --report=roofline --memory-level=L2_L3 --project-dir=./myAdvisorProj

See Also
integrated Model traffic on all levels of the memory hierarchy for a Roofline report.

Command Line Interface Reference
advixe-cl Command Action Reference

memory-operation-type
Model only load memory operations, store memory operations, or both, in a Roofline interactive HTML report.

GUI Equivalent
Roofline > Default: FLOAT > Memory Operation Type

Syntax
--memory-operation-type=<string>

Arguments
<string> is one of the following: load | store | all

Default
all

Actions Modified
report=roofline
Example
Generate a Roofline interactive HTML report. Show only load memory operations.

$ advixe-cl --report=roofline --memory-operation-type=load --project-dir=./advixe

See Also
Command Line Interface Reference
advixe-cl Command Action Reference

mix
Show dynamic or static instruction mix data in a Survey report.

GUI Equivalent
Code Analytics

Syntax
--mix
--no-mix

Default
Off (no-mix)

Actions Modified
report = survey

Usage
Dynamic instruction mix is counted for the entire execution of the application; static instruction mix is counted per iteration. The static-instruction-mix, dynamic, and mix options work together in the following manner:

- Collect static instruction mix data: --collect=survey --static-instruction-mix
  (In the GUI: Static instruction mix data is calculated on demand.)
- Collect dynamic instruction mix data (and static instruction mix data, from which dynamic mix data is calculated): --collect=tripcounts --flop
- Show static instruction mix data in a Survey report: --report=survey --mix --no-dynamic
- Show dynamic mix instruction data in a Survey report: --report=survey --mix --dynamic
- A Survey report cannot show both static and dynamic mix instruction data.
  (In the GUI: Code Analytics can show both static and dynamic instruction mix data.)

Example
1. Run a Survey analysis.
2. Run a Trip Counts & FLOP analysis. Collect dynamic instruction mix data (and static instruction mix data, from which dynamic mix data is calculated).
3. Generate a Survey report. Show dynamic instruction mix data. (dynamic is on, by default).

$ advixe-cl --collect=survey --project-dir=./advixe -- ./bin/myTargetApplication
$ advixe-cl --collect=tripcounts --flop --project-dir=./advixe
$ advixe-cl --report=survey --mix --project-dir=./advixe

1. Run a Survey analysis. Collect static instruction mix data.
2. Generate a Survey report. Show static instruction mix data.

$ advixe-cl --collect=survey --static-instruction-mix --project-dir=./advi -- ./bin/myTargetApplication
$ advixe-cl --report=survey --mix --no-dynamic --project-dir=./advi

See Also

dynamic  Show (in a Survey report) how many instructions of a given type actually executed during Trip Counts & FLOP analysis.
static-instruction-mix  Statically calculate the number of specific instructions present in the binary during Survey analysis.

Command Line Interface Reference
advixe-cl Command Action Reference

mkl-user-mode
Collect Intel® Math Kernel Library (Intel® MKL) loops and functions data during the Survey analysis

GUI Equivalent

Project Properties > Analysis Target > Survey Analysis > Advanced > Analyze MKL loops and functions

Syntax

--mkl-user-mode
--no-mkl-user-mode

Default

On (mkl-user-mode)

Actions Modified

collect=survey

Usage

Disabling can decrease finalization overhead.

Example

Run a Survey analysis. Disable collecting Intel MKL loops and functions data.

$ advixe-cl --collect=survey --project-dir=./myAdvisorProj --no-mkl-user-mode -- ./bin/myTargetApplication

See Also

Minimizing Analysis Overhead

Command Line Interface Reference
advixe-cl Command Action Reference

module-filter
Specify application (or child application) module(s) to include in or exclude from analysis.

GUI Equivalent

Modules
Syntax

--module-filter=<string>

Arguments

<string> is a comma-separated list of module names (no spaces).

Default

None - so when coupled with module-filter-mode default (exclude), the Intel Advisor analyzes all modules.

Actions Modified

collect=[analysis type] --module-filter-mode

Usage

Usage can decrease collection and finalization overhead.

Example

Run a Survey analysis. Exclude modules foo1.so and foo2.so.

```
$ advixe-cl --collect=survey --project-dir=./myAdvisorProj --module-filter-mode=exclude --module-filter=foo1.so,foo2.so -- ./bin/myTargetApplication
```

See Also

module-filter-mode  Limit, by inclusion or exclusion, application (or child application) module(s) for analysis.

Minimizing Analysis Overhead

Command Line Interface Reference
advixe-cl Command Action Reference

GUI Equivalent

Modules

Syntax

--module-filter-mode=<string>

Arguments

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>include</td>
<td>Include the modules specified in module-filter.</td>
</tr>
<tr>
<td>exclude</td>
<td>Exclude the modules specified in module-filter.</td>
</tr>
</tbody>
</table>

Default

Exclude - so when coupled with module-filter default (empty), the Intel Advisor analyzes all modules.
**Actions Modified**

`collect=[analysis type] --module-filter`

**Usage**

Usage can decrease collection and finalization overhead.

**Example**

Run a Survey analysis. Exclude modules `foo1.so` and `foo2.so`.

```
$ advixe-cl --collect=survey --project-dir=./myAdvisorProj --module-filter-mode=exclude --module-filter=foo1.so,foo2.so -- ./bin/myTargetApplication
```

**See Also**

module-filter Specify application (or child application) module(s) to include in or exclude from analysis.

Minimizing Analysis Overhead

Command Line Interface Reference

advixe-cl Command Action Reference

**mpi-rank**

*Specify MPI process data to import.*

**Syntax**

```
--mpi-rank=<integer>
```

**Arguments**

`<integer>` is the rank of the process with data to import.

**Default**

If an MPI rank is not specified, and there is more than one result directory in the project because the result partition is shared, a rank is chosen at random. Recommendation: Specify a rank.

**Actions Modified**

import-dir, mark-up-loops, report

**Usage**

When you collect analysis data on a cluster, the data is stored in unique subdirectories under the project directory, named `rank.#`. Use this option to specify the process with data to import for viewing. You can import data from only one process at a time.

**Example**

Import MPI analysis data from the `rank.3` cluster. Read source files from the specified search directory. Write the result to the `new-advi` project directory.

```
$ advixe-cl --import-dir=./advi --mpi-rank=3 --project-dir=./new-advi --search-dir src:=./src
```

**See Also**

MPI Workloads

Command Line Interface Reference

advixe-cl Command Action Reference
**mrte-mode**
Set the Microsoft* runtime environment mode for analysis.

**GUI Equivalent**
Project Properties  >  Analysis Target  >  [Analysis Types]  >  Managed code profiling mode

**Syntax**
```
--mrte-mode=<string>
```

**Arguments**

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>auto</td>
<td>Automatically detect the type of target executable and switch to that mode.</td>
</tr>
<tr>
<td>native</td>
<td>Collect data for native code and do not attribute data to managed code.</td>
</tr>
<tr>
<td>mixed</td>
<td>Collect data for both native and managed code, and attribute data to managed code as appropriate. Consider using this option when analyzing a native executable that makes calls to the managed code.</td>
</tr>
<tr>
<td>managed</td>
<td>Collect data for both native and managed code, resolve samples attributed to native code, attribute data to managed source only. The call stack in the analysis result displays data for managed code only.</td>
</tr>
</tbody>
</table>

**Default**
auto

**Actions Modified**
collect

**Usage**
Applies to Windows* OS only.

**Example**
Run a Suitability analysis. Set a native runtime environment mode.
```
$ advixe-cl --collect=suitability --mrte-mode=native -- C:\test\sample.exe
```

**See Also**
Command Line Interface Reference
advixe-cl Command Action Reference

**option-file**
Specify a text file containing command line arguments.

**Syntax**
```
--option-file=<PATH>
```

**Arguments**

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;PATH&gt;</td>
<td>is the PATH/name of a text file containing command line arguments.</td>
</tr>
</tbody>
</table>
**Actions Modified**

`collect, report`

**Usage**

Put commonly used options in a UTF-8 text file to shorten the command line and create a reusable invocation syntax. Enter one option on each line. No spaces are allowed in the option entry; use a new line instead.

Arguments specified in an option file are processed before any arguments specified on the command line; therefore, options specified on the command line can override options in an option file.

**Example**

Create a reusable option file you can use whenever you want to generate a report that specifies the project directory, directory to search for source files, and PATH/name of the output text file.

```bash
--project-dir=./advi
--search-dir src=./src
--format=text
--report-output=./out/annotations.txt
```

Generate a Suitability report. Use an option file named `my_suitability_analysis.txt`.

```bash
$ advixe-cl --report=suitability --option-file=./my_suitability_analysis.txt
```

**See Also**

Command Line Interface Reference
advixe-cl Command Action Reference

**pack**

*Pack a snapshot into an archive.*

**GUI Equivalent**

File > Create Data Snapshot > Pack into archive

**Syntax**

```bash
--pack
--no-pack
```

**Default**

On (pack)

**Actions Modified**

`snapshot`

**Usage**

By default, the archive is saved to the current directory with the default `snapshotXXX.advixeexpz` name.

**Example**

Create a new snapshot in the project directory. Do not pack it into an archive.

```bash
$ advixe-cl --snapshot --project-dir=./advi --no-pack
```
Create a new snapshot. Pack it into an archive. Save it in the current directory with the default name.

$ advixe-cl --snapshot --project-dir=./advi --pack

Create a new snapshot. Pack it into an archive named new_snapshot.advixeexpz.

$ advixe-cl --snapshot --project-dir=./advi --pack -- /new_snapshot

See Also
Command Line Interface Reference
advixe-cl Command Action Reference

profile-gpu (Beta)
Analyze OpenCL™ and Intel® Media SDK programs running on Intel® Processor Graphics.

Syntax
--profile-gpu
--no-profile-gpu

Default
Off (no-profile-gpu)

Actions Modified
collect=survey
collect=tripcounts
collect=roofline

Usage

---

Important
- This option is available only in the Intel® Advisor Beta, which is part of Intel® oneAPI Base Toolkit. For more information, see https://software.intel.com/content/www/us/en/develop/tools/oneapi/components/advisor.html.
- GPU profiling is applicable only to Intel® Processor Graphics.

---

This option may affect the performance of your application on the CPU side.

Example
Run a Survey analysis. Enable GPU profiling to analyze OpenCL™ and Intel® Media SDK programs running on Intel® Processor Graphics.

$ advixe-cl --collect=survey --project-dir=./advi --profile-gpu -- myApplication

See Also
Command Line Interface Reference
advixe-cl Command Action Reference

profile-jit
Collect metrics about Just-In-Time (JIT) generated code regions during the Trip Counts and FLOP analysis.
GUI Equivalent

Project Properties > Analysis Target > Trip Counts and FLOP Analysis > Advanced > Capture metrics for dynamic loops and functions

Syntax
--profile-jit
--no-profile-jit

Default
On (profile-jit)

Actions Modified
collect=tripcounts

Usage
Enabling can increase collection overhead.

Example
1. Run a Survey analysis.
2. Run a Trip Counts and FLOP analysis. Explicitly enable collecting metrics for JIT generated code.

   $ advixe-cl --collect=survey --project-dir=./advi -- myApplication
   $ advixe-cl --collect=tripcounts --flop --profile-jit --project-dir=./advi -- myApplication

See Also
Command Line Interface Reference
advixe-cl Command Action Reference

profile-intel-perf-libs (Beta)
Show Intel® performance libraries loops and functions in Intel® Advisor Beta reports.

Syntax
--profile-intel-perf-libs
--no-profile-intel-perf-libs

Default
On (profile-intel-perf-libs)

Actions Modified
collect

Usage

Important
This option is available only in the Intel® Advisor Beta, which is part of Intel® oneAPI Base Toolkit. For more information, see https://software.intel.com/content/www/us/en/develop/tools/oneapi/components/advisor.html.
Example
Run a Survey analysis and disable showing Intel® performance libraries in the report.

$ advixe-cl --collect=survey --project-dir=./advi --no-profile-intel-perf-libs -- myApplication

See Also
Command Line Interface Reference
advixe-cl Command Action Reference

profile-python
Collect Python loop and function data during Survey analysis.

GUI Equivalent
Project Properties > Analysis Target > Survey Analysis > Advanced > Analyze Python loops and functions

Syntax
--profile-python
--no-profile-python

Default
Off (no-profile-python)

Actions Modified
collect=survey

Usage
Enabling can increase collection overhead.

Example
Run a Survey analysis. Explicitly disable collecting Python loop and function data.

$ advixe-cl --collect=survey --project-dir=./myAdvisorProj --no-profile-python -- ./bin/myTargetApplication

See Also
Minimizing Analysis Overhead

Command Line Interface Reference
advixe-cl Command Action Reference

project-dir
Specify the top-level directory where a result is saved if you want to save the collection somewhere other than the current working directory.

Syntax
--project-dir=<PATH>
Arguments

<PATH> is the PATH/name of a directory.

Default

Current working directory

Actions Modified

collect, create-project, import-dir, mark-up-loops, report, snapshot

Usage

Recommendation: Specify the project directory when you:

- Generate a report from a collection.
- Import MPI collections.

Examples

Run a Survey analysis on myApplication. Search the src directory for all source, binary, and symbol files. Write the result to advi.

```
$ advixe-cl --collect=survey --project-dir=./advi --search-dir all:=./src -- myApplication
```

Generate a Survey report from the result in the advi directory. Output the report in text format as survey.txt.

```
$ advixe-cl --report=survey --project-dir=./advi --format=text --report-output=./out/survey.txt
```

See Also

Command Line Interface Reference
advixe-cl Command Action Reference

quiet

Minimize status messages during command execution.

Syntax

-q

--quiet

Default

Off

Actions Modified

collect, command, import-dir, mark-up-loops, report, snapshot

Example

Generate a Suitability report. Output to stderr. Show warnings, errors, and fatal errors.

```
$ advixe-cl --report=suitability --project-dir=./advi --quiet
```

See Also

verbose Maximize status messages during command execution.

Command Line Interface Reference
advixe-cl Command Action Reference
**recalculate-time**
Recalculate total time after filtering a report.

**GUI Equivalent**

**Filters**

**Syntax**

--recalculate-time
--no-recalculate-time

**Default**
On (recalculate-time)

**Actions Modified**

report=survey --filter
report=tripcounts --filter

**Example**

Generate a Survey report. Show data only for scalar loops. Do not recalculate total time.

```bash
$ advixe-cl --report=survey --project-dir=./adv --filter="Type"="Scalar" --no-recalculate-time
```

Generate a Survey report. Show data only for loops/functions from my_module1. Explicitly recalculate total time.

```bash
$ advixe-cl --report=survey --project-dir=./adv --filter="Module"="my_module1" --recalculate-time
```

**See Also**

Command Line Interface Reference
advixe-cl Command Action Reference

**record-mem-allocations**
Enable heap allocation tracking to identify heap-allocated variables for which access strides are detected during Memory Access Patterns analysis.

**GUI Equivalent**

Project Properties > Analysis Target > Memory Access Patterns Analysis > Advanced > Report heap allocated variables

**Syntax**

--record-mem-allocations
--no-record-mem-allocations

**Default**
On (record-mem-allocations)

**Actions Modified**

collect=map

**Usage**

Disabling can decrease collection overhead.
Example
Run a Memory Access Patterns analysis. Disable heap allocation tracking.

```bash
$ advixe-cl --collect=map --project-dir=./advi --no-record-mem-allocations -- ./bin/myApplication
```

See Also
Minimizing Analysis Overhead

Command Line Interface Reference
advixe-cl Command Action Reference

**record-stack-frame**
Capture stack frame pointers to identify stack variables for which access strides are detected during Memory Access Patterns analysis.

GUI Equivalent
Project Properties > Analysis Target > Memory Access Patterns Analysis > Advanced > Report stack variables

Syntax
--record-stack-frame
--no-record-stack-frame

Default
On (record-stack-frame)

Actions Modified
collect=map

Usage
Disabling can decrease collection overhead.

Example
Run a Memory Access Patterns analysis. Disable capturing stack frame pointers.

```bash
$ advixe-cl --collect=map --project-dir=./advi --no-record-stack-frame -- ./bin/myApplication
```

See Also
Minimizing Analysis Overhead

Command Line Interface Reference
advixe-cl Command Action Reference

**reduce-lock-contention**
Examine specified annotated sites for opportunities to reduce lock contention or find deadlocks in a Suitability report.

GUI Equivalent
Suitability > Lock Contention
Syntax
--reduce-lock-contention=<string>

Arguments
<string> is a comma-separated list of annotated sites (no spaces).

Default
No default argument

Actions Modified
report=suitability

Usage
Lock contention is the time one thread spends waiting for a lock to be released while another thread holds that lock (as opposed to lock overhead, which is the time spent creating, destroying, acquiring, and releasing locks). You can reduce lock contention by using different locks for unrelated data when you convert to a parallel framework.

Usage of this option simulates parallel execution with the assumption that lock contention is zero for a specified site.

Example
Generate a Suitability report. Examine the annotated sites myAnnotatedSiteJ and myAnnotatedSiteX for opportunities to reduce lock contention. Write the report to stdout.

$ advixe-cl --report=suitability --project-dir=./advi --reduce-lock-contention=myAnnotatedSiteJ,myAnnotatedSiteX

See Also
Reducing Lock Contention
Command Line Interface Reference
advixe-cl Command Action Reference

GUI Equivalent
Suitability > Lock Overhead

Syntax
--reduce-lock-overhead=<string>

Arguments
<string> is a comma-separated list of annotated sites (no spaces).

Default
No default argument

Actions Modified
report=suitability
Usage
Lock overhead is the time spent creating, destroying, acquiring, and releasing locks (as opposed to lock contention, which is the time spent waiting for a lock held by another task). Think of lock overhead as the cost of lock operations, assuming the lock is always available.

Usage of this option simulates parallel execution with the assumption that lock overhead is zero for a specified site.

Example
Generate a Suitability report. Examine the annotated sites myAnnotatedSiteJ and myAnnotatedSiteX for opportunities to reduce lock overhead. Write the report to stdout.

```bash
$ advixe-cl --report=suitability --project-dir=./advi --reduce-lock-overhead=myAnnotatedSiteJ,myAnnotatedSiteX
```

See Also
Reducing Lock Overhead
Command Line Interface Reference
advixe-cl Command Action Reference

reduce-site-overhead
Examine specified annotated sites for opportunities to reduce site overhead in a Suitability report.

GUI Equivalent
Suitability > Site Overhead

Syntax
```
--reduce-site-overhead=<string>
```

Arguments

`<string>` is a comma-separated list of sites (no spaces).

Default
No default argument

Actions Modified
```
report=suitability
```

Usage
Site overhead is the time spent starting up (and shutting down) parallel execution. It includes creating threads, scheduling those threads onto cores, and waiting for the threads to begin executing.

Usage of this option simulates parallel execution with the assumption that site overhead is zero for a specified site.

Example
Generate a Suitability report. Examine the annotated sites myAnnotatedSiteJ and myAnnotatedSiteX for opportunities to reduce site overhead. Write the report to stdout.

```bash
$ advixe-cl --report=suitability --project-dir=./advi --reduce-site-overhead=myAnnotatedSiteJ,myAnnotatedSiteX
```
See Also
Reducing Site Overhead
Command Line Interface Reference
advixe-cl Command Action Reference

reduce-task-overhead
Examine specified annotated sites for opportunities to reduce task overhead in a Suitability report.

GUI Equivalent
Suitability > Task Overhead

Syntax
--reduce-task-overhead=<string>

Arguments
<string> is a comma-separated list of sites (no spaces).

Default
No default argument

Actions Modified
report=suitability

Usage
Task overhead is the time spent creating a task, assigning it to a thread, and stopping or pausing the thread when the task is complete.

Usage of this option simulates parallel execution with the assumption that task overhead is zero for a specified site.

Example
Generate a Suitability report. Examine the annotated sites myAnnotatedSiteJ and myAnnotatedSiteX for opportunities to reduce task overhead. Write the report to stdout.

    $ advixe-cl --report=suitability --project-dir=./advi --reduce-task-overhead=myAnnotatedSiteJ,myAnnotatedSiteX

See Also
Reducing Task Overhead
Command Line Interface Reference
advixe-cl Command Action Reference

refinalize-survey
Refinalize a survey result collected with a previous Intel Advisor version or if you need to correct or update source and binary search paths.

GUI Equivalent
Workflow > Re-finalize Survey

Syntax
--refinalize-survey
--no-refinalize-survey

**Default**

Off (no-refinalize-survey)

**Actions Modified**

*collect*=survey
*report*=survey

**Usage**

Typical usage scenarios:

- Source files were moved after compilation.
- You open remotely collected results on a viewing machine before setting the search paths appropriately.

**Example**

Refinalize the result of the Survey analysis. Search recursively for source files in the .*/src search directory and for binary files in the .*/bin search directory. Write the refinalized results to the .*/advi project directory instead of the default working directory.

```
$ advixe-cl --collect=survey --project-dir=./advi --search-dir src:=./src bin:=./bin -- ./bin/myApplication --refinalize-survey
```

**See Also**

Command Line Interface Reference
advixe-cl Command Action Reference

remove

*Remove loops (by file and line number) from the loops selected for deeper analysis.*

**GUI Equivalent**

Survey > 📂

**Syntax**

--remove=<string>

**Arguments**

*<string>* is a comma-separated list of files/line numbers in the following format: file1:line1

**Default**

No default string

**Actions Modified**

mark-up-loops

**Usage**

Removing loops that are not of interest can decrease collection overhead.

Do not confuse the *mark-up-loops* action with the *mark-up-list* action option. The *mark-up-loops* action coupled with the *select* action option enables a GUI 📂 checkbox; therefore loop selection persists beyond the duration of the *mark-up-loops* action and applies to downstream analyses, such as Dependencies and
Memory Access Patterns analyses. The collect action coupled with the mark-up-list action option simulates enabling a GUI checkbox; therefore loop selection persists only for the duration of the collect action.

**Example**

1. Select two loops for deeper analysis.
2. Remove one loop from the selection list.

```bash
$ advixe-cl --mark-up-loops --select=foo.cpp:34,bar.cpp:192 --project-dir=./myAdvisorProj
  -- ./bin/myTargetApplication
$ advixe-cl --mark-up-loops --remove=bar.cpp:192 --project-dir=./myAdvisorProj
  -- ./bin/myTargetApplication
```

**See Also**

Command Line Interface Reference  
advixe-cl Command Action Reference

---

### report-output

*Redirect report output from stdout to another location.*

**Syntax**

```
--report-output=<PATH>
```

**Arguments**

<PATH> is the directory PATH/filename.

**Default**

stdout

**Actions Modified**

report

**Example**

Generate a Suitability report. Output the report in text format. Save it as suitability.txt in the out output directory.

```bash
$ advixe-cl --report=suitability --project-dir=./advi
  --format=text --report-output=./out/suitability.txt
```

**See Also**

Command Line Interface Reference  
advixe-cl Command Action Reference

---

### report-template

*Specify the PATH/name of a custom report template file.*

**Syntax**

```
--report-template=<PATH>
```

**Arguments**

<PATH> is the directory PATH/name.
**Default**

No default argument

**Actions Modified**

report

**Example**

Generate a custom report. Use the ./template/suitability.template file.

```
$ advixe-cl --report=custom --project-dir=./advi --report-template=./template/suitability.template
```

**See Also**

Command Line Interface Reference
advixe-cl Command Action Reference

**result-dir**

*Specify a directory to identify the running analysis.*

**Syntax**

```
--r=<PATH>
--result-dir=<PATH>
```

**Arguments**

<PATH> is the directory PATH.

**Default**

No default argument

**Actions Modified**

command

**Example**

Pause the analysis running in the r000hs directory.

```
$ advixe-cl --command=pause -r=./r000hs
```

**See Also**

Command Line Interface Reference
advixe-cl Command Action Reference

**resume-after**

*Resume collection after the specified number of milliseconds.*

**Syntax**

```
--resume-after=<integer>
```

**GUI Equivalent**

Project Properties > Analysis Target > [Analysis Type] > Advanced > Automatically resume collection after (sec)
Arguments

<integer> is the number of milliseconds to wait before resuming data collection.

Default

Off

Actions Modified

collect

Usage

Skip uninteresting parts of your target application, such as the initialization phase, and analyze only interesting parts.

Collection automatically starts in the paused state.

Usage can decrease collection overhead.

NOTE

The value in the corresponding GUI property is in seconds, not milliseconds.

Example

Run a Survey analysis. Launch the application with collection paused. Start collection after 30 milliseconds.

$ advixe-cl --collect=survey --project-dir=./myAdvisorProj --resume-after=30 -- ./bin/myTargetApplication

See Also

Minimizing Analysis Overhead
Command Line Interface Reference
advixe-cl Command Action Reference

return-app-exitcode

Return the target exit code instead of the command line interface exit code.

Syntax

--return-app-exitcode
--no-return-app-exitcode

Default

Off (no-return-app-exitcode)

Actions Modified

collect

Example

$ advixe-cl --collect=survey --project-dir=./myAdvisorProj --return-app-exitcode -- ./bin/myTargetApplication

See Also

Command Line Interface Reference
advixe-cl Command Action Reference

**search-dir**

*Specify the location(s) for finding target support files.*

**GUI Equivalent**

*Project Properties > Binary/Symbol Search*

*Project Properties > Source Search*

**Syntax**

```
--search-dir <keyword>=<PATH>
```

**Arguments**

Combine keywords with arguments in the following syntax: `<all | bin | src | sym[:<p | r>]>=<PATH>`

*<PATH>* is the PATH/name of the search directory, and can include environment paths and absolute paths.

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>all</td>
<td>Search all types of directories.</td>
</tr>
<tr>
<td>bin</td>
<td>Search binary directories.</td>
</tr>
<tr>
<td>src</td>
<td>Search source directories. This is used for most collect actions.</td>
</tr>
<tr>
<td>sym</td>
<td>Search symbol directories.</td>
</tr>
<tr>
<td>:r</td>
<td>Perform a recursive search of all subdirectories.</td>
</tr>
<tr>
<td>:p</td>
<td>Specify the highest priority search (directories to search prior to others, including environment paths and absolute paths).</td>
</tr>
</tbody>
</table>

**Actions Modified**

*collect, create-project, import-dir, report*

**Usage**

Use `-search-dir src:=<PATH>` when performing collect actions.

To exclude files from analysis, use the `exclude-files` option.

**Example**

Run a Suitability analysis on myApplication. Search for source files in the specified search directory. Write the result to the specified project directory.

```
$ advixe-cl --collect=suitability --project-dir=./advi --search-dir src:=./src1 -- myApplication
```

The following two commands are equivalent. Each runs a Suitability analysis on myApplication and searches for source files in the two specified search directories.

```
$ advixe-cl --collect=suitability --project-dir=./advi --search-dir src:=./src1 --search-dir src:=./src2 -- myApplication
```

```
$ advixe-cl --collect=suitability --project-dir=./advi --search-dir src:=./src1,./src2 -- myApplication
```

**See Also**

*exclude-files*  Exclude the specified files or directories from annotation scanning during analysis.

*Binary/Symbol Search Tab*

*Source Search Tab*
Command Line Interface Reference
advixe-cl Command Action Reference

select
Select loops (by file and line number) for deeper analysis.

GUI Equivalent
Survey > 🗼

Syntax
--select=<string>

Arguments
<string> is a comma-separated list of files/line numbers in the following format: file1:line1.

Default
No default argument

Actions Modified
mark-up-loops

Usage
Selecting loops of interest can decrease collection overhead.

Do not confuse the mark-up-loops action with the mark-up-list action option. The mark-up-loops action coupled with the select action option enables a GUI 🗼 checkbox; therefore loop selection persists beyond the duration of the mark-up-loops action and applies to downstream analyses, such as Dependencies and Memory Access Patterns analyses. The collect action coupled with the mark-up-list action option simulates enabling a GUI 🗼 checkbox; therefore loop selection persists only for the duration of the collect action.

Example
1. Run a Survey analysis to identify loops of interest.
2. Select those loops for deeper analysis.

$ advixe-cl --collect=survey --project-dir=./myAdvisorProj -- ./bin/myTargetApplication
$ advixe-cl --mark-up-loops --select=foo.cpp:34,bar.cpp:192 --project-dir=./myAdvisorProj -- ./bin/myTargetApplication

See Also
Minimizing Analysis Overhead

Command Line Interface Reference
advixe-cl Command Action Reference

show-all-columns
Show data for all available columns in a Survey report.

GUI Equivalent
Survey > Customize View > Settings > Configure Columns
Syntax
--show-all-columns
--no-show-all-columns

Default
Off (no-show-all-columns)

Actions Modified
report = survey
report = top-down

Example
Generate a Survey report. Output in CSV format. Show data for all available columns.

$ advixe-cl --report=survey --project-dir=./advi --show-all-columns --format=csv

See Also
Command Line Interface Reference
advixe-cl Command Action Reference

show-all-rows
Show data for all available rows, including data for child loops, in a Survey report.

GUI Equivalent
Survey > Function Calls Sites and Loops > +

Syntax
--show-all-rows
--no-show-all-rows

Default
On (show-all-rows)

Actions Modified
report = survey

Example
Generate a default Survey report. Output in CSV format. Show data for present child loops.

$ advixe-cl --report=survey --project-dir=./advi --format=csv --report-output=./out/survey.csv

Generate a default Survey report. Output in CSV format. Do not show data for present child loops.

$ advixe-cl --report=survey --project-dir=./advi --format=csv --no-show-all-rows --report-output=./out/survey.csv

See Also
Command Line Interface Reference
advixe-cl Command Action Reference
show-functions
Show only functions in a report.

GUI Equivalent
Filters
Syntax
--show-functions
--no-show-functions

Default
Off (no-show-functions)

Actions Modified
report

Usage
The show-loops option, which shows only loops in generated reports, is switched on by default. The show-functions option, which shows only functions in generated reports, is switched off by default.

Example
Generate a Survey report showing both loops and functions.

$ advixe-cl --report=survey --show-functions --project-dir=./advi --format=text --report-output=./out/survey.txt

Generate a Survey report showing functions only.

$ advixe-cl --report=survey --no-show-loops --show-functions --project-dir=./advi --format=text --report-output=./out/survey.txt

Generate a default Survey report showing loops only.

$ advixe-cl --report=survey --project-dir=./advi --format=text --report-output=./out/survey.txt

Generate an empty Survey report - you disabled showing information about loops, and showing information about functions is off by default.

$ advixe-cl --report=survey --no-show-loops --project-dir=./advi --format=text --report-output=./out/survey.txt

See Also
show-loops  Show only loops in a report.

Command Line Interface Reference
advixe-cl Command Action Reference
advixe-cl Command Action Reference

show-loops
Show only loops in a report.

GUI Equivalent
Filters
Syntax
--show-loops
--no-show-loops

**Default**
On (show-loops)

**Actions Modified**

report

**Usage**

The **show-loops** option, which shows only loops in generated reports, is switched on by default. The **show-functions** option, which shows only functions in generated reports, is switched off by default.

**Example**

Generate a Survey report showing **both loops and functions**.

```
$ advixe-cl --report=survey --show-functions --project-dir=./advi --format=text --report-output=./out/survey.txt
```

Generate a Survey report showing **functions only**.

```
$ advixe-cl --report=survey --no-show-loops --show-functions --project-dir=./advi --format=text --report-output=./out/survey.txt
```

Generate a default Survey report showing **loops only**.

```
$ advixe-cl --report=survey --project-dir=./advi --format=text --report-output=./out/survey.txt
```

Generate an **empty** Survey report - you disabled showing information about loops, and showing information about functions is off by default.

```
$ advixe-cl --report=survey --no-show-loops --project-dir=./advi --format=text --report-output=./out/survey.txt
```

**See Also**

show-functions Show only functions in a report.

Command Line Interface Reference
advixe-cl Command Action Reference

show-not-executed

*Show not-executed child loops in a Survey report.*

**GUI Equivalent**

Survey > **Show all loops (with zero and non-zero time)**

**Syntax**

--show-not-executed

--no-show-not-executed

**Default**

Off (no-show-not-executed)

**Actions Modified**

report = survey

report = top-down
**Example**

Generate a top-down Survey report. Output in text format. Show data for not-executed child loops.

```bash
$ advixe-cl --report=top-down --project-dir=./advi --show-not-executed --format=text --report-output=./out/topdown.txt
```

**See Also**

Command Line Interface Reference  
advice-cl Command Action Reference

**sort-asc**

Sort data in ascending order (by specified column name) in a report.

**GUI Equivalent**

Sort **Survey** by column

**Syntax**

```bash
--sort-asc=<string>
```

**Arguments**

<string> is column name.

**Actions Modified**

report

**Usage**

Data in string format is sorted lexicographically.

**Example**

Generate a Survey report. Sort data in ascending order by **Total Time**.

```bash
$ advixe-cl --report=survey --sort-asc="Total Time"
```

**See Also**

Command Line Interface Reference  
advice-cl Command Action Reference

**sort-desc**

Sort data in descending order (by specified column name) in a report.

**GUI Equivalent**

Sort **Survey** by column

**Syntax**

```bash
--sort-desc=<string>
```

**Arguments**

<string> is column name.
**Actions Modified**
report

**Usage**
Data in string format is sorted lexicographically.

**Example**
Generate a Survey report. Sort data in descending order by **Total Time**.

```
$ advixe-cl --report=survey --sort-desc="Total Time"
```

**See Also**
Command Line Interface Reference
advixe-cl Command Action Reference

**spill-analysis**
*Register flow analysis to calculate the number of consecutive load/store operations in registers and related memory traffic in bytes during Survey analysis.*

**GUI Equivalent**
**Project Properties > Analysis Target > Survey Analysis > Advanced > Enable register spill/fill analysis**

**Syntax**

```
--spill-analysis
--no-spill-analysis
```

**Default**
Off (no-spill-analysis)

**Actions Modified**
collect=survey

**Usage**
Enabling can increase finalization overhead.

**Example**
Run a Survey analysis. Enable spill/fill analysis.

```
$ advixe-cl --collect=survey --project-dir=./myAdvisorProj --spill-analysis -- ./bin/myTargetApplication
```

**See Also**
Minimizing Analysis Overhead
Command Line Interface Reference
advixe-cl Command Action Reference
stacks
Perform advanced collection of callstack data during
Roofline and Trip Counts & FLOP analysis.

GUI Equivalent
Project Properties > Analysis Target > Trip Counts & FLOP Analysis > Advanced > Collect stacks
Workflow > Enable Roofline with Callstacks

Syntax
--stacks
--no-stacks

Default
Off (no-stacks)

Actions Modified
collect=roofline
collect=tripcounts

Usage
Enabling can increase collection overhead.

Example
Run a Roofline analysis. Collect advanced callstack data.

```bash
$ advixe-cl --collect=roofline --project-dir=./myAdvisorProj --stacks -- ./bin/myTargetApplication
```

See Also
Minimizing Analysis Overhead
Command Line Interface Reference
advixe-cl Command Action Reference

stack-stitching
Restructure the call flow during Survey analysis to
attach stacks to a point introducing a parallel
workload.

GUI Equivalent
Project Properties > Analysis Target > Survey Hotspots Analysis > Advanced > Stitch stacks

Syntax
--stack-stitching
--no-stack-stitching

Default
On (stack-stitching)

Actions Modified
collect=survey
Usage
The option restores a logical call tree for Intel® Threading Building Blocks (Intel® TBB) or OpenMP* applications by catching notifications from the runtime and attaching stacks to a point introducing a parallel workload.

Disable when Survey analysis runtime overhead exceeds 1.1x
Disabling can decrease collection overhead and significantly decrease finalization overhead depending on workload.

Example
Run a Survey analysis. Disable stack stitching.

```bash
$ advixe-cl --collect=survey --project-dir=./myAdvisorProj --no-stack-stitching -- ./bin/myTargetApplication
```

See Also
Minimizing Analysis Overhead

Command Line Interface Reference
advixe-cl Command Action Reference

stackwalk-mode
Choose between online and offline modes to analyze stacks during Survey analysis.

GUI Equivalent
Project Properties > Analysis Target > Survey Hotspots Analysis > Advanced > Stack unwinding mode

Syntax
```
--stackwalk-mode=<string>
```

Arguments

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>online</td>
<td>Analyze stacks during collection.</td>
</tr>
<tr>
<td>offline</td>
<td>Analyze stacks after collection.</td>
</tr>
</tbody>
</table>

Default
offline

Actions Modified
```
collect=survey
```

Usage
Set to offline when Survey analysis overhead exceeds 1.1x and/or a large quantity of data is allocated on the stack, which is a common case for Fortran applications or applications with a large number of small, parallel, OpenMP* regions.

Usage can decrease collection overhead.
Example
Run a Survey analysis. Analyze stacks during collection.

```
advixe-cl --collect=survey --project-dir=./myAdvisorProj --stackwalk-mode=online -- ./bin/myTargetApplication
```

See Also
Minimizing Analysis Overhead

Command Line Interface Reference
advixe-cl Command Action Reference

start-paused
Start executing the target application for analysis purposes, but delay data collection.

GUI Equivalent

**Workflow ➤**

To resume data collection: **Workflow ➤**

Syntax
--start-paused

Default
Off

Actions Modified
collect

Usage
Skip *uninteresting* parts of your target application, such as the initialization phase, and analyze only *interesting* parts.

You can use different techniques to resume collection, such as __itt_resume__. Usage can decrease analysis overhead.

Example
Launch the sample application with Suitability data collection paused.

```
$ advixe-cl --collect=suitability --start-paused -- C:\test\sample.exe
```

See Also
Minimizing Analysis Overhead
Command Line Interface Reference
advixe-cl Command Action Reference

static-instruction-mix
*Statically calculate the number of specific instructions present in the binary during Survey analysis.*
GUI Equivalent

Project Properties > Analysis Target > Survey Analysis > Advanced > Enable static instruction mix analysis

Syntax

`--static-instruction-mix`

`--no-static-instruction-mix`

Default

Off (no-static-instruction-mix)

Actions Modified

`collect=survey`

Usage

Dynamic instruction mix is counted for the entire execution of the application; static instruction mix is counted per iteration. The `static-instruction-mix`, `dynamic`, and `mix` options work together in the following manner:

- Collect static instruction mix data: `--collect=survey --static-instruction-mix`
  
  (In the GUI: Static instruction mix data is calculated on demand.)

- Collect dynamic instruction mix data (and static instruction mix data, from which dynamic mix data is calculated): `--collect=tripcounts --flop`

- Show static instruction mix data in a Survey report: `--report=survey --mix --no-dynamic`

- Show dynamic mix instruction data in a Survey report: `--report=survey --mix --dynamic`

- A Survey report cannot show both static and dynamic mix instruction data.

  (In the GUI: Code Analytics can show both static and dynamic instruction mix data.)

Enabling `static-instruction-mix`:

- Is necessary in scenarios involving the Python* API.
- Can increase finalization overhead.

Example

1. Run a Survey analysis. Collect static instruction mix data.
2. Generate a Survey report. Show static instruction mix data.

   `$ advixe-cl --collect=survey --static-instruction-mix --project-dir=./advi -- ./bin/myTargetApplication`

   `$ advixe-cl --report=survey --mix --no-dynamic --project-dir=./advi`

See Also

dynamic  Show (in a Survey report) how many instructions of a given type actually executed during Trip Counts & FLOP analysis.

mix  Show dynamic or static instruction mix data in a Survey report.

Minimizing Analysis Overhead

Command Line Interface Reference

advixe-cl Command Action Reference

strategy

Specify processes and/or children for instrumentation during Survey analysis.
Syntax

--strategy=<string>

Arguments

<string> is a comma-separated list (no spaces) in the format [process1 | child1:]profiling mode.

Available profiling modes include the following:

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>trace:trace</td>
<td>Instrument process and all children.</td>
</tr>
<tr>
<td>trace:notrace</td>
<td>Instrument process but not children.</td>
</tr>
<tr>
<td>notrace:trace</td>
<td>Instrument children but not process.</td>
</tr>
<tr>
<td>notrace:notrace</td>
<td>Do not instrument process or children.</td>
</tr>
</tbody>
</table>

Default

Instrument all processes and children (strategy=trace:trace)

Actions Modified

collect=survey

Example

Process_A starts several processes:

```
Root >
   Process_A >
      Child_of_A_1
      Child_of_A_2
   Process_B >
      Child_of_B
```

Run a Survey analysis. Instrument Child_of_A_2 and all children of Process_B.

```
```

See Also

Command Line Interface Reference
advixe-cl Command Action Reference

support-multi-isa-binaries

Collect a variety of data during Survey analysis for loops that reside in non-executed code paths.

GUI Equivalent

Project Properties > Analysis Target > Survey Hotspots Analysis > Advanced > Analyze loops that reside in non-executed code paths

Syntax

--support-multi-isa-binaries
--no-support-multi-isa-binaries

**Default**

Off (no-support-multi-isa-binaries)

**Actions Modified**

collect=survey for binaries compiled using the ax (Linux* OS)/Qax (Windows* OS) option with an Intel compiler

**Usage**

Disabling can decrease finalization overhead.

**Example**

Run a Survey analysis. Explicitly disable collecting data for loops that reside in non-executed code paths.

```
$ advixe-cl --collect=survey --project-dir=./myAdvisorProj --no-support-multi-user-binaries -- ./bin/myTargetApplication
```

**See Also**

Minimizing Analysis Overhead

Command Line Interface Reference
advixe-cl Command Action Reference

target-pid

*Attach Survey or Trip Counts & FLOP collection to a running process specified by the process ID.*

**GUI Equivalent**

Project Properties > Analysis Target > [Analysis Type] > Attach to Process > PID

**Syntax**

```
--target-pid=<integer>
```

**Arguments**

*<integer>* is a process ID.

**Default**

No default argument

**Actions Modified**

collect=survey - with call stacks disabled (default)

collect=tripcounts - with call stacks disabled (default)

**Usage**

The usage scenario is similar to starting a target application with collection paused, except you can attach to an already running process.

Usage can decrease collection overhead.

Use the command action with the arguments shown below to:

- detach - the process continues running but analysis data collection stops.
• stop - kill the process, which also stops analysis data collection.

**Example**

Attach Survey collection to the running process with the process ID 5.

```
$ advixe-cl --collect=survey --project-dir=./myAdvisorProj --target-pid=5
```

**See Also**

- command: Control the Intel Advisor while running analyses.
- stacks: Perform advanced collection of callstack data during Roofline and Trip Counts & FLOP analysis.
- start-paused: Start executing the target application for analysis purposes, but delay data collection.

**Minimizing Analysis Overhead**

**Command Line Interface Reference**

**advixe-cl Command Action Reference**

**target-process**

*Attach Survey or Trip Counts & FLOP collection to a running process specified by the process name.*

**GUI Equivalent**

**Project Properties > Analysis Target > [Analysis Type] > Attach to Process > Process name**

**Syntax**

```
--target-process=<string>
```

**Arguments**

`<string>` is a process name.

**Default**

No default argument

**Actions Modified**

- `collect=survey` - with call stacks disabled (default)
- `collect=tripcounts` - with call stacks disabled (default)

**Usage**

The usage scenario is similar to starting a target application with collection paused, except you can attach to an already running process.

Usage can decrease collection overhead.

Use the `command` action with the arguments shown below to:

• detach - the process continues running but analysis data collection stops.
• stop - kill the process, which also stops analysis data collection.

**Example**

Attach Survey collection to the running process `MyProcess`.

```
$ advixe-cl --collect=survey --project-dir=./myAdvisorProj --target-process=MyProcess
```
See Also
command Control the Intel Advisor while running analyses.
stacks Perform advanced collection of callstack data during Roofline and Trip Counts & FLOP analysis.
start-paused Start executing the target application for analysis purposes, but delay data collection.

Minimizing Analysis Overhead
Command Line Interface Reference
advixe-cl Command Action Reference

target-system
Specify the hardware configuration to use for modeling purposes in a Suitability report.

GUI Equivalent
Suitability > Target System

Syntax
--target-system=<string>

Arguments
<string> is one of the following: cpu | xeon-phi | offload-to-xeon-phi

Default
cpu

Actions Modified
report=suitability

Example
Generate a Suitability report. Use Intel® Xeon Phi™ as the target configuration

$ advixe-cl --report=suitability --project-dir=./advi --target-system=xeon-phi

See Also
Command Line Interface Reference
advixe-cl Command Action Reference

threading-model
Specify the threading model to use for modeling purposes in a Suitability report.

GUI Equivalent
Suitability > Threading Model

Syntax
--threading-model=<string>

Arguments
<string> is one of the following: tbb | openmp | tpl (Windows* OS only) | other
**Default**
tbb

**Actions Modified**

*report* = *suitability*

**Usage**

Use only one threading model. Mixing threading models is not supported.

**Example**

Generate a Suitability report for the OpenMP* parallel framework.

```
$ advixe-cl --report=suitability --threading-model=openmp --project-dir=./advi
```

**See Also**

Command Line Interface Reference
advixe-cl Command Action Reference

**top-down**

*Generate a Survey report in top-down view.*

**GUI Equivalent**

Survey > Top Down

**Syntax**

```
--top-down
```

**Default**

Off

**Actions Modified**

*report* = *survey*

**Usage**

Equivalent to *report* = *top-down*

**Example**

Generate a Survey report in top-down view.

```
$ advixe-cl --report=survey --top-down --project-dir=./advi.
```

**See Also**

Command Line Interface Reference
advixe-cl Command Action Reference

**track-heap-objects (Beta)**

*Attribute heap-allocated objects to the analyzed loops that accessed the objects.*

**Syntax**

```
--track-heap-objects
--no-track-heap-objects
```
**Default**
Off (no-track-heap-objects)

**Actions Modified**
`collect=tripcounts --enable-data-transfer-analysis`

**Usage**
Enabling can increase analysis overhead.

**Important**
This option is available only in the Intel® Advisor Beta, which is part of Intel® oneAPI Base Toolkit. For more information, see https://software.intel.com/content/www/us/en/develop/tools/oneapi/components/advisor.html.

**Example**
Run a Trip Counts and FLOP analysis. Enable data transfer simulation and track accesses to heap-allocated objects.

```bash
$ advixe-cl --collect=tripcounts --project-dir=./advi --flop --enable-data-transfer-analysis --track-heap-objects -- myApplication
```

**See Also**
`enable-data-transfer-analysis` Model data transfer between host memory and device memory.

**Command Line Interface Reference**
`advixe-cl Command Action Reference`

**trace-mode**
Set how to trace loop iterations during Memory Access Patterns analysis.

**Syntax**
`--trace-mode=<string>`

**Arguments**
`<string>` is one of the following:

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>full</td>
<td>Trace all loop iterations.</td>
</tr>
<tr>
<td>linear</td>
<td>Trace loop iterations using linear step.</td>
</tr>
<tr>
<td>fibo</td>
<td>Trace loop iterations in Fibonacci sequence.</td>
</tr>
</tbody>
</table>

**Default**
fibo

**Actions Modified**
`collect=map`
Usage
Specifying a less extensive tracing method can decrease collection overhead.

Example
Run a Memory Access Patterns analysis on the specified loops. Trace all loop iterations.

```bash
$ advixe-cl --collect=map --mark-up-list=3,4,5 --trace-mode=full --project-dir=./myAdvisorProj -- ./bin/myTargetApplication
```

See Also
Command Line Interface Reference
advixe-cl Command Action Reference

trace-mpi
Configure collectors to trace MPI code and determine MPI rank IDs for non-Intel® MPI library implementations.

Syntax
```bash
--trace-mpi
--no-trace-mpi
```

Default
Off (no-trace-mpi)

Actions Modified
collect

Example
Run a Survey analysis. Use a non-Intel MPI library on a Windows* OS.

```bash
$ mpiexec -n 4 advixe-cl --collect=survey --trace-mpi --no-auto-finalize --project-dir=./new-advi -- ./bin/myApplication
```

See Also
Command Line Interface Reference
advixe-cl Command Action Reference

track-stack-accesses (Beta)
Track accesses to stack memory.

Syntax
```bash
--track-stack-accesses
--no-track-stack-accesses
```

Default
Off (no-track-stack-accesses)

Actions Modified
collect=tripcounts --enable-data-transfer-analysis
Usage

**Important**
This option is available only in the Intel® Advisor Beta, which is part of Intel® oneAPI Base Toolkit. For more information, see https://software.intel.com/content/www/us/en/develop/tools/oneapi/components/advisor.html.

By default, the Intel® Advisor Beta filters out all accesses to stack memory. When you enable this option, all accesses to stack memory are included in data transfer calculations.

**Example**
Run a Trip Counts and FLOP analysis. Enable data transfer simulation and analyze accesses to stack memory.

```bash
$ advixe-cl --collect=tripcounts --project-dir=./advi --flop --enable-data-transfer-analysis --track-stack-accesses -- myApplication
```

**See Also**

* enable-data-transfer-analysis (Beta) Model data transfer between host memory and device memory.

Command Line Interface Reference

**track-stack-variables**
Enable parallel data sharing analysis for stack variables during Dependencies analysis.

**GUI Equivalent**

**Project Properties > Analysis Target > Dependencies Analysis > Analyze stack variables**

**Syntax**

--track-stack-variables

--no-track-stack-variables

**Default**

On (track-stack-variables)

**Actions Modified**

* collect=dependencies

**Usage**

Disabling can decrease collection overhead.

**Example**

Run a Dependencies analysis. Disable parallel data sharing analysis for stack variables.

```bash
$ advixe-cl --collect=dependencies --no-track-stack-variables --project-dir=./advi --search-dir src:=./src -- ./bin/myApplication
```

**See Also**

Command Line Interface Reference

advixe-cl Command Action Reference
trip-counts
Collect loop trip counts data during Trip Counts & FLOP analysis.

GUI Equivalent
Workflow > Trip Counts
Project Properties > Analysis Target > Trip Counts and FLOP Analysis > Collect information about loop trip counts

Syntax
--trip-counts
--no-trip-counts

Default
On (trip-counts)

Actions Modified
collect=tripcounts

Usage
Use the option, which allows you to dynamically identify the number of times loops are invoked and executed, to:

- Detect loops with too-small trip counts and trip counts that are not a multiple of vector length.
- Analyze parallelism granularity more deeply.

Disabling can decrease analysis overhead.

Example
Run a Trip Counts & FLOP analysis. Collect trip counts, FLOP, and call stack data.

$ advixe-cl --collect=tripcounts --flop --stacks --project-dir=./advi -- ./bin/myApplication

Run a Trip Counts & FLOP analysis. Collect only FLOP data.

$ advixe-cl --collect=tripcounts --no-trip-counts --project-dir=./advi --search-dir src:=./src -- ./bin/myApplication

See Also
Command Line Interface Reference
advixe-cl Command Action Reference

user-data-dir
Specify a directory other than the project directory to save analysis results.

GUI Equivalent
File > Options > Result Location

Syntax
-user-data-dir=<PATH>

Arguments
The directory for saving the result
Default
project-dir

Actions Modified
collect, report

Usage
Save a file somewhere other than the default project-dir, such as a remote directory or simply another directory when there is not enough space in project-dir.

Example
Run Suitability analysis. Save the result in the directory specified by user-data-dir.

```
$ advixe-cl --collect=suitability --project-dir=./advi --user-data-dir=<PATH> --search-dir src:=./src -- myApplication
```

See Also
project-dir Specify the top-level directory where a result is saved if you want to save the collection somewhere other than the current working directory.
Command Line Interface Reference
advixe-cl Command Action Reference

verbose
Maximize status messages during command execution.

Syntax
-v
--verbose

Default
Off

Actions Modified
collect, command, import-dir, mark-up-loops, report, snapshot,

Example
Generate a Suitability report. Output to stderr. Show additional status output.

```
$ advixe-cl --report=suitability --project-dir=./advi --verbose
```

See Also
quiet Minimize status messages during command execution.
Command Line Interface Reference
advixe-cl Command Action Reference

with-stack
Show call stack data in a Roofline interactive HTML report (if call stack data is collected).

GUI Equivalent
Roofline> Default: FLOAT > With Callstacks
Syntax
--with-stack
--no-with-stack

Default
Off (no-with-stack)

Actions Modified
report=roofline

Example
Generate a Roofline interactive HTML report. Show call stack data.

```
$ advixe-cl --report=roofline --project-dir=./advi --report-output=./out/roofline.html --with-stacks
```

See Also
Command Line Interface Reference
advixe-cl Command Action Reference

Dependencies Problem and Message Types Reference
The Intel® Advisor Dependencies analysis identifies various data sharing problems and messages. This reference section describes these problems and messages, and offers possible correction strategies.

<table>
<thead>
<tr>
<th>Problem Type Name</th>
<th>Severity and Cause</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dangling Lock</td>
<td>Error. Occurs when a task does not release a lock before the task ends.</td>
</tr>
<tr>
<td>Data Communication</td>
<td>Error. Occurs when a task writes a value that a different task reads. If not fixed prior to conversion to parallel code, a Data Communication problem could result in a data race.</td>
</tr>
<tr>
<td>Data Communication, Child Task</td>
<td>Error. Occurs when a task writes a value that a different (child) task reads. If not fixed prior to conversion to parallel code, a Data Communication problem could result in a data race.</td>
</tr>
<tr>
<td>Inconsistent Lock Use</td>
<td>Warning. Occurs when a task execution accesses a memory location more than once, under the control of different locks.</td>
</tr>
<tr>
<td>Lock Hierarchy Violation</td>
<td>Warning. Occurs when two or more locks are acquired in a different order in two task executions, potentially leading to a deadlock when the program’s tasks execute in parallel.</td>
</tr>
<tr>
<td>Memory Reuse</td>
<td>Error. Occurs when two tasks write to a shared memory location. That is, a task writes to a variable with a new value but does not read the same value generated by a prior task. If not fixed prior to conversion to parallel code, this Memory Reuse problem could result in a data race.</td>
</tr>
<tr>
<td>Problem Type Name</td>
<td>Severity and Cause</td>
</tr>
<tr>
<td>--------------------------------</td>
<td>----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>Memory Reuse, Child Task</td>
<td>Error. Occurs when two tasks write to a shared memory location, where a parent task overwrites a variable with a new value that was read by a previously executed child task in the same site. If not fixed prior to conversion to parallel code, this Memory Reuse, Child Task problem could result in a data race.</td>
</tr>
<tr>
<td>Memory Watch</td>
<td>Remark. Occurs when a task accesses a memory location marked by an <code>ANNOTATE_OBSERVE_USES</code> annotation. In this case, this problem provides informational feedback only and no action is required. This is useful for finding uses of specified memory locations while a task is executing.</td>
</tr>
<tr>
<td>Missing End Site</td>
<td>Error. Occurs when a site-begin annotation is executed but the corresponding site-end annotation is not executed before the thread or application exits.</td>
</tr>
<tr>
<td>Missing End Task</td>
<td>Error. Occurs when a task-begin annotation is executed but the corresponding task-end annotation is not executed before the site, thread, or application exits.</td>
</tr>
<tr>
<td>Missing Start Site</td>
<td>Error. Occurs when an end-site annotation is executed but there is no active site.</td>
</tr>
<tr>
<td>Missing Start Task</td>
<td>Error. Occurs when an end task annotation is executed but there is no active task.</td>
</tr>
<tr>
<td>No tasks in parallel site</td>
<td>Warning. Occurs when a parallel site was executed but no task annotations were executed in the dynamic extent of the active parallel site.</td>
</tr>
<tr>
<td>One task instance in parallel site</td>
<td>Warning. Occurs when a parallel site was executed but annotations for only one task instance were executed in the dynamic extent of the active parallel site. This may be the expected behavior, or it may indicate an error in the placement of annotations or a data set that is not well suited for parallelism.</td>
</tr>
<tr>
<td>Orphaned Task</td>
<td>Error. Occurs when a task-begin annotation is executed that is not within an active parallel site.</td>
</tr>
<tr>
<td>Parallel Site Information</td>
<td>Remark. Occurs when execution enters a parallel site. This confirms that your program and its data are executing the annotations you inserted during execution of the Dependencies tool analysis. In this case, this message provides informational feedback only and no action is required.</td>
</tr>
<tr>
<td>Thread Information</td>
<td>Remark. In this case, this message provides informational feedback and no action is required.</td>
</tr>
<tr>
<td>Unhandled Application Exception</td>
<td>Error. Occurs when an unhandled exception is detected that causes the application program to crash.</td>
</tr>
</tbody>
</table>

**Dangling Lock**

*Occurs when a task does not release a lock before the task ends.*
Syntax

<table>
<thead>
<tr>
<th>ID</th>
<th>Code Location</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Allocation site</td>
<td>If present, represents the location and associated call stack when the lock was created.</td>
</tr>
<tr>
<td>2</td>
<td>Lock owned</td>
<td>If present, represents the location and its associated call stack when the lock was last acquired.</td>
</tr>
<tr>
<td>3</td>
<td>Parallel site</td>
<td>If present, represents the location and associated call stack of the site-begin annotation of the parallel site containing the task that acquired the lock.</td>
</tr>
</tbody>
</table>

Example

```c
void problem()
{
    ANNOTATE_SITE_BEGIN(dangle_site1);        // Parallel site
    ANNOTATE_TASK_BEGIN(task1);
    ANNOTATE_LOCK_ACQUIRE(&dangle);   // Lock owned
    ANNOTATE_TASK_END();
    // ...
    ANNOTATE_SITE_END();
}
```

In this example:
- There is a parallel site that contains a task.
- The lock is acquired in the task.
- The lock is not released before the end of the task - the `ANNOTATE_LOCK_RELEASE()` annotation is missing.

Possible Correction Strategies
- Make sure that an `ANNOTATE_LOCK_RELEASE(address)` annotation is executed on every flow path from an `ANNOTATE_LOCK_ACQUIRE(address)` annotation to the end of the task. If a code region has multiple exit flow paths, make sure the lock is released on all the paths.
- Consider putting the lock-acquire and lock-release in the constructor and destructor of a static object, so that lock release happens automatically.
- Consider putting a lock release in an exception handler if a the locked region might exit from an exception.

Data Communication

*Occurs when a task writes a value that a different task reads.*

Syntax
<table>
<thead>
<tr>
<th>ID</th>
<th>Code Location</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Allocation site</td>
<td>If present, represents the location and associated call stack when the memory block was allocated.</td>
</tr>
<tr>
<td>2</td>
<td>Parallel site</td>
<td>If present, represents the location and associated call stack of the parallel site containing the Data Communication problem.</td>
</tr>
<tr>
<td>3</td>
<td>Write</td>
<td>Represents the instruction and associated call stack where the memory was written.</td>
</tr>
<tr>
<td>4</td>
<td>Read</td>
<td>Represents the instruction and associated call stack where the memory was read in a different task execution.</td>
</tr>
</tbody>
</table>

### Example

In this example, the write to the heap variable in task1 might occur either before or after the read in task2.

```c
void problem() {
    int* pointer = new int;               // Allocation site
    ANNOTATE_SITE_BEGIN(datacomm_site1); // Begin parallel site
    ANNOTATE_TASK_BEGIN(task1);
    *pointer = 999;               // Write
    ANNOTATE_TASK_END();
    ANNOTATE_TASK_BEGIN(task2);
    assert(*pointer == 999);      // Read
    ANNOTATE_TASK_END();
    ANNOTATE_SITE_END();
}
```

In this example, each task execution reads the variable `communication`, adds one to its value, and writes the result back to the variable. The write in each task execution might occur either before or after the read in the other task instance execution.

```c
void data_communication() {
    ANNOTATE_SITE_BEGIN(site);       // Parallel site
    for (int i = 0; i < 2; i++) {
        ANNOTATE_TASK_BEGIN(task);   // Write and Read in different task execution
        communication++;     /* data communication */
        ANNOTATE_TASK_END();
    }
    ANNOTATE_SITE_END();
}
```

### Possible Correction Strategies

- If two accesses to the same memory must occur in a specific order, then the accesses must not be in different task executions in a single site execution. You will have to change the structure of your sites and tasks.
- If the order of memory modifications in two task executions is not important, but the executions of the modifications must not occur simultaneously, use locks to synchronize them.
- Induction and reduction annotations can tell the Dependencies tool about programs where the program behavior will be correct, regardless of the order of the accesses.

### See Also

Data Sharing Problems
Data Communication, Child Task

Occurs when a task writes a value that a different (child) task reads. A child task is a task nested inside another task.

Syntax

<table>
<thead>
<tr>
<th>ID</th>
<th>Code Location</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Allocation site</td>
<td>If present, represents the location and associated call stack when the memory block was allocated.</td>
</tr>
<tr>
<td>2</td>
<td>Parallel site</td>
<td>Represents the location and associated call stack of the parallel site containing the Data Communication problem.</td>
</tr>
<tr>
<td>3</td>
<td>Write</td>
<td>Represents the instruction and associated call stack where the memory was written.</td>
</tr>
<tr>
<td>4</td>
<td>Read</td>
<td>Represents the instruction and associated call stack where the memory was read in a different task execution.</td>
</tr>
</tbody>
</table>

Example

```c
void problem()
{
    int* pointer = new int;               // Allocation site
    ANNOTATE_SITE_BEGIN(datacomm_site1);  // Begin parallel site
    ANNOTATE_TASK_BEGIN(task1);
    *pointer = 999;               // Write
    ANNOTATE_TASK_END();
    assert(*pointer == 999);          // Read
    ANNOTATE_SITE_END();
}
```

In this example, one task writes a heap-allocated int, then an ancestor task reads it.

```c
void data_communication()
{
    ANNOTATE_SITE_BEGIN(data_communication_site);  // Parallel site
    for (int i=0; i<N; i++) {
        ANNOTATE_TASK_BEGIN(data_communication_task1);
        communication++; /* write in child */  // Write
        ANNOTATE_TASK_END();
        printf("%d\n", communication); /* read in parent */  // Read
    }
    ANNOTATE_SITE_END();
}
```

In this example, the incremented variable is read after each task. This creates a serial dependence.
Possible Correction Strategies
If you can preserve the application's integrity, consider moving the reads by the parent task into the child task. In the example above, this would result in non-deterministic output. If moving the read is not possible, you may need to use a different strategy, such as pipelining the loop.

Inconsistent Lock Use
Occurs when a task execution accesses a memory location more than once, under the control of different locks.

Syntax
One of the following has occurred:

<table>
<thead>
<tr>
<th>ID</th>
<th>Code Location</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Allocation site</td>
<td>If present, represents the location and associated call stack when the memory was allocated.</td>
</tr>
<tr>
<td>2</td>
<td>Parallel site</td>
<td>If present, represents the location and associated call stack of the parallel site containing the Inconsistent Lock Use problem.</td>
</tr>
<tr>
<td>3</td>
<td>Read</td>
<td>Represents the location and associated call stack of the first access if it is a memory read.</td>
</tr>
<tr>
<td>4</td>
<td>Write</td>
<td>Represents the location and associated call stack of the second access if it is a memory write.</td>
</tr>
<tr>
<td>5</td>
<td>Read</td>
<td>Represents the location and associated call stack of the second access if it is a memory read.</td>
</tr>
<tr>
<td>6</td>
<td>Write</td>
<td>Represents the location and associated call stack of the first access if it is a memory write.</td>
</tr>
</tbody>
</table>

Example
```c
// Parallel site
ANNOTATE_TASK_BEGIN(task);
for (int i = 0; i < N; i++) {
    ANNOTATE_LOCK_ACQUIRE(1);
    a[i][j0]++;               // Read and/or Write
    ANNOTATE_LOCK_RELEASE(1);
}
for (int j = 0; i < N; i++) {
    ANNOTATE_LOCK_ACQUIRE(2);
```
In this example, \(a[i0][j0]\) is accessed under lock 1 in the first loop and under lock 2 in the second loop. It is likely that an access in another task will not have the right combination of locks to avoid conflicting with both these accesses.

Possible Correction Strategies

Lock all accesses to the same memory location with the same lock.

Lock Hierarchy Violation

Occurs when two or more locks are acquired in a different order in two task executions, potentially leading to a deadlock when the program's tasks execute in parallel.

A Lock hierarchy violation problem indicates the following timeline:

Task 1
1. Acquire lock A.
2. Acquire lock B.
3. Release lock B.
4. Release lock A.

Task 2
1. Acquire lock B.
2. Acquire lock A.
3. Release lock A.
4. Release lock B.

If these time lines are interleaved when the two tasks execute in parallel, a Deadlock occurs:

1. Task 1: Acquire lock A.
2. Task 2: Acquire lock B.
3. Task 1: Try to acquire lock B; wait until task 2 releases it.
4. Task 2: Try to acquire lock A; wait until task 1 releases it.

The Dependencies tool reports a Lock hierarchy violation as multiple problems in a problem set. Each problem shows a portion of the Lock hierarchy violation from the perspective of a single thread.

Lock hierarchy violation problems are the most common cause of Deadlock problems, and a report of a Lock hierarchy violation problem indicates a Deadlock problem might occur when the target executes in parallel.

Syntax

<table>
<thead>
<tr>
<th>ID</th>
<th>Code Location</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Allocation site</td>
<td>If present, represents the location and its associated call stack where the synchronization object acquired by a thread (usually the object acquired first) was created.</td>
</tr>
<tr>
<td>2</td>
<td>Parallel site</td>
<td>If present, represents the location and associated call stack of the parallel site containing the Lock Hierarchy Violation problem.</td>
</tr>
<tr>
<td>ID</td>
<td>Code Location</td>
<td>Description</td>
</tr>
<tr>
<td>----</td>
<td>---------------</td>
<td>-------------</td>
</tr>
<tr>
<td>3</td>
<td>Lock owned</td>
<td>Represents the location and associated call stack where a task acquired a lock.</td>
</tr>
<tr>
<td>4</td>
<td>Lock owned</td>
<td>Represents the location and associated call stack where a task acquired a second lock while the task still held the first lock.</td>
</tr>
</tbody>
</table>

**Example**

```c
// in task 1
ANNOTATE_LOCK_ACQUIRE(&lahv_lock1);
ANNOTATE_LOCK_ACQUIRE(&lahv_lock2); /* lock hierarchy violation */
ANNOTATE_LOCK_RELEASE(&lahv_lock2);
ANNOTATE_LOCK_RELEASE(&lahv_lock1);

// in task 2
ANNOTATE_LOCK_ACQUIRE(&lahv_lock2);
ANNOTATE_LOCK_ACQUIRE(&lahv_lock1); /* lock hierarchy violation */
ANNOTATE_LOCK_RELEASE(&lahv_lock1);
ANNOTATE_LOCK_RELEASE(&lahv_lock2);
```

**Possible Correction Strategies**

Determine if interleaving is possible, or whether some other synchronization exists that might prevent interleaving. If interleaving is possible, consider the following options.

Use a single lock instead of multiple locks:

```c
// in task 1
ANNOTATE_LOCK_ACQUIRE(&lahv_lock1);
a++;
b += a;
ANNOTATE_LOCK_RELEASE(&lahv_lock1);

// in task 2
ANNOTATE_LOCK_ACQUIRE(&lahv_lock2);
b += x[i];
a -= b;
ANNOTATE_LOCK_RELEASE(&lahv_lock2);
```

Try to define a consistent order for your locks, so that any task that acquires the same set of locks, will acquire them in the same order:

```c
// in task 1
ANNOTATE_LOCK_ACQUIRE(&lahv_lock1);
a++;
b += a;
ANNOTATE_LOCK_RELEASE(&lahv_lock1);

// in task 2
ANNOTATE_LOCK_ACQUIRE(&lahv_lock2);
b += x[i];
a -= b;
ANNOTATE_LOCK_RELEASE(&lahv_lock2);
```

When a task acquires multiple locks, make sure that it always releases them in the opposite order that it acquired them.
Memory Reuse

Occurs when two tasks write to a shared memory location. That is, a task writes to a variable with a new value but does not read the same value generated by a prior task.

Syntax

One of the following has occurred:

<table>
<thead>
<tr>
<th>ID</th>
<th>Code Location</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Allocation site</td>
<td>If present, and if the memory involved is heap memory, represents the location and associated call stack when the memory was allocated.</td>
</tr>
<tr>
<td>2</td>
<td>Parallel site</td>
<td>If present, represents the location and associated call stack of the parallel site containing the Memory Reuse problem.</td>
</tr>
<tr>
<td>3</td>
<td>Read</td>
<td>Represents the instruction and associated call stack of the first access if it is a memory read.</td>
</tr>
<tr>
<td>4</td>
<td>Write</td>
<td>Represents the instruction and associated call stack of the second access if it is a memory write.</td>
</tr>
<tr>
<td>5</td>
<td>Write</td>
<td>Represents the instruction and associated call stack of the first access if it is a memory write.</td>
</tr>
</tbody>
</table>

Example

```c
int global;
void main()
{
    ANNOTATE_SITE_BEGIN(reuse_site);   // Begin parallel site
    ANNOTATE_TASK_BEGIN(task111);
    global = 111;                      // Read and/or Write
    assert(global == 111);
    ANNOTATE_TASK_END();
    ANNOTATE_TASK_BEGIN(task222);
    global = 222;                      // Write
    assert(global == 222);
    ANNOTATE_TASK_END();
    ANNOTATE_SITE_END();
}
```

In this example, two tasks use the same `global` variable. Each task does not read or communicate the value produced by the other task.
**Some Possible Correction Strategies**

Change the tasks to have their own private variables rather than sharing a variable.

**Memory Reuse, Child Task**

*Occurs when two tasks write to a shared memory location, where a parent task overwrites a variable with a new value that was read by a previously executed child task. A child task is a task nested inside another task.*

**Syntax**

<table>
<thead>
<tr>
<th>ID</th>
<th>Code Location</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Allocation site</td>
<td>If present, and if the memory involved is heap memory, represents the location and associated call stack when the memory was allocated.</td>
</tr>
<tr>
<td>2</td>
<td>Parallel site</td>
<td>If present, represents the location and associated call stack of the parallel site containing the Memory Reuse, Child Task problem.</td>
</tr>
<tr>
<td>3</td>
<td>Read</td>
<td>Represents the instruction and associated call stack of the first access if it is a memory read.</td>
</tr>
<tr>
<td>4</td>
<td>Write</td>
<td>Represents the instruction and associated call stack of the second access if it is a memory write.</td>
</tr>
</tbody>
</table>

**Example**

```c
int global;
void main()
{
    ANNOTATE_SITE_BEGIN(reuse_site); // Begin parallel site
    ANNOTATE_TASK_BEGIN(task111);
    assert(global == 111); // Read
    ANNOTATE_TASK_END();
    global = 222; // Write
    ANNOTATE_SITE_END();
}
```

In this example, a parent task is writing to a shared variable after a task that reads that same variable.

**Some Possible Correction Strategies**

Create a private copy of the variable before executing the child task. Use the private copy in the child task.

**Memory Watch**

*Occurs when a task accesses a memory location marked by an ANNOTATE_OBSERVE_USES annotation. In this case, this problem provides informational feedback only and no action is required. This is useful for finding uses of specified memory locations while a task is executing.*
Syntax
One of the following has occurred:

<table>
<thead>
<tr>
<th>ID</th>
<th>Code Location</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Parallel site</td>
<td>If present, represents the location and associated call stack of the parallel site containing the Memory Watch problem.</td>
</tr>
<tr>
<td>2</td>
<td>Watch start</td>
<td>Represents the location and its associated call stack where an <code>ANNOTATE_OBSERVEUSES()</code> annotation marks a memory location.</td>
</tr>
<tr>
<td>3</td>
<td>Read</td>
<td>Represents the location and associated call stack where a task read the watched memory location.</td>
</tr>
<tr>
<td>4</td>
<td>Write</td>
<td>Represents the location and associated call stack where a task wrote the watched memory location.</td>
</tr>
<tr>
<td>5</td>
<td>Update</td>
<td>Represents the location and associated call stack where a task read and wrote the watched memory location.</td>
</tr>
</tbody>
</table>

Example

```c
void watch_memory()
{
    ANNOTATE_OBSERVEUSES(&watch, sizeof(watch));  // Watch start
    ANNOTATE_SITE_BEGIN(watch_site);               // Parallel site
    {
        ANNOTATE_TASK_BEGIN(watch_task1);
        {
            ANNOTATE_LOCK_ACQUIRE(&watch);
            watch++; /* watch memory */  // Read and/or Write
            ANNOTATE_LOCK_RELEASE(&watch);
        }
        ANNOTATE_TASK_END();
    }
    ANNOTATE_TASK_END();
    ANNOTATE_TASK_BEGIN(watch_task2);
    {
        ANNOTATE_LOCK_ACQUIRE(&watch);
        watch++; /* watch memory */  // Read and/or Write
        ANNOTATE_LOCK_RELEASE(&watch);
    }
    ANNOTATE_TASK_END();
}```
This example reports all places that use the memory location referenced by `watch` during the call to `watch_memory()`.

**Possible Correction Strategies**

To use `ANNOTATE_OBSERVEUSES` to help you correct an incidental sharing problem, do the following to mark places where you may be able replace uses of a shared memory location with uses of a non-shared memory location:

1. Add an `ANNOTATE_OBSERVEUSES` annotation to the task.
2. Find all uses of the shared memory location in the dynamic extent of the task.

**Missing End Site**

Occurs when a `site-begin` annotation is executed but the corresponding `site-end` annotation is not executed before the thread or application exits.

### Syntax

```
1 Start site
```

<table>
<thead>
<tr>
<th>ID</th>
<th>Code Location</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Start site</td>
<td>Represents the location and the associated call stack when the parallel</td>
</tr>
<tr>
<td></td>
<td></td>
<td>site execution began.</td>
</tr>
</tbody>
</table>

**Example**

```c
void main()
{
    ANNOTATE_SITE_BEGIN(sitel); // Begin parallel site
    return;
    ANNOTATE_SITE_END();
}
```

This example's execution skips the end-site annotation, `ANNOTATE_SITE_END()`.

**Possible Correction Strategies**

Always execute an `ANNOTATE_SITE_END()` annotation after executing an `ANNOTATE_SITE_BEGIN(sitename)` annotation. This omission can be caused by throw exceptions, return, break, continue, and goto statements or keywords. All control flow paths out of a site need to use the `ANNOTATE_SITE_END()` annotations.

**Missing End Task**

Occurs when a task-begin annotation is executed but the corresponding task-end annotation is not executed before the site, thread, or application exits.
Syntax

<table>
<thead>
<tr>
<th>ID</th>
<th>Code Location</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Task start</td>
<td>Represents the location and associated call stack when the task began execution.</td>
</tr>
<tr>
<td>2</td>
<td>Parallel site</td>
<td>If present, represents the location and associated call stack of the beginning of the parallel site that contained the task.</td>
</tr>
</tbody>
</table>

Example

```c
void main()
{
    ANNOTATE_SITE_BEGIN(site_name);
    ANNOTATE_TASK_BEGIN(taskname1);
    ANNOTATE_SITE_END();
}
```

This example lacks an end-task annotation, `ANNOTATE_TASK_END()`.

**NOTE**
An error also occurs if your code branches around a single `ANNOTATE_TASK_END()` annotation.

Possible Correction Strategies

Always execute an `ANNOTATE_TASK_BEGIN(taskname)` annotation before executing an `ANNOTATE_SITE_END()` annotation. This omission can be caused by `throw exceptions`, `return`, `break`, `continue`, and `goto` statements or keywords. All control flow paths out of a site need to use the `ANNOTATE_TASK_END()` annotations.

Missing Start Site

Occurs when an end-site annotation is executed but there is no active site.

Syntax

<table>
<thead>
<tr>
<th>ID</th>
<th>Code Location</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>End site</td>
<td>Represents the location and associated call stack when the end site annotation was executed.</td>
</tr>
</tbody>
</table>
Example

```c
void main()
{
    ANNOTATE_SITE_END(); // End parallel site
}
```

This example executes an `ANNOTATE_SITE_END()` annotation before executing the corresponding (in this case, missing) `ANNOTATE_SITE_BEGIN(sitename)` annotation.

Possible Correction Strategies

Always execute an `ANNOTATE_SITE_BEGIN()` annotation before executing an `ANNOTATE_SITE_END()` annotation.

Missing Start Task

Occurs when an end task annotation is executed but there is no active task.

Syntax

<table>
<thead>
<tr>
<th>ID</th>
<th>Code Location</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Task end</td>
<td>Represents the location and associated call stack when the task end annotation was executed.</td>
</tr>
<tr>
<td>2</td>
<td>Parallel site</td>
<td>If present, represents the location and associated call stack of the beginning of the parallel site that contained the task end annotation.</td>
</tr>
</tbody>
</table>

Example

```c
void main()
{
    ANNOTATE_SITE_BEGIN(name_site1);
    ANNOTATE_TASK_END();
    ANNOTATE_SITE_END();
}
```

This example lacks an `ANNOTATE_TASK_BEGIN(taskname)` annotation.

**NOTE**

This error also occurs if your code branches around an `ANNOTATE_TASK_BEGIN(taskname)` annotation.

Possible Correction Strategies

Always execute an `ANNOTATE_TASK_BEGIN(taskname)` annotation before executing an `ANNOTATE_TASK_END()` annotation.
No Tasks in Parallel Site

Occurs when a parallel site was executed but no task annotations were executed in the dynamic extent of the active parallel site.

Syntax

<table>
<thead>
<tr>
<th>ID</th>
<th>Code Location</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Parallel site</td>
<td>Represents the location and associated call stack of the parallel site. No task annotations were executed in the dynamic extent of the active parallel site.</td>
</tr>
</tbody>
</table>

Example

```c
int global;
void main()
{
    ANNOTATE_SITE_BEGIN(reuse_site); // Parallel site
    assert(global == 111);
    global = 222;
    ANNOTATE_SITE_END();
}
```

In this example, the site begin and site end annotations are present, but the execution paths within the parallel site do not execute any task annotations.

Some Possible Correction Strategies

Check the execution paths within the parallel site and add task annotations to mark at least one task.

One Task Instance in Parallel Site

Occurs when a parallel site was executed but annotations for only one task instance were executed in the dynamic extent of the active parallel site. This may be the expected behavior, or it may indicate an error in the placement of annotations or a data set that is not well suited for parallelism.

Syntax

<table>
<thead>
<tr>
<th>ID</th>
<th>Code Location</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Parallel site</td>
<td>Represents the location and associated call stack of the parallel site. Occurs when a parallel site was executed but annotations for only one instance of a task's code were executed in the dynamic extent of the active parallel site. The warning is based on the site and task annotations detected during program execution. This may be the expected behavior. In other cases, this</td>
</tr>
</tbody>
</table>
warning may indicate an error in the placement of annotations or a data set that is not well suited for parallelism (a single instance of a task may not contribute to parallel execution speed-up).

**Example**

```c
int global;
extern arg_map parse_args(int argc, char ** argv);
void main(int argc, char * argv[])
{
  int x;
  parse_args(argc, argv);
  int y = arg_map.get("iterations"); //command line specifies 1 iteration

  ANNOTATE_SITE_BEGIN(loopsite);     // Parallel site
  for (x=0; x<y; x++)
  {
    ANNOTATE_ITERATION_TASK(task);
    ...
  }
  ANNOTATE_SITE_END();
}
```

In this example, the selected data set results in only a single iteration of the loop. No dependencies will be found between multiple iterations of the loop.

**Some Possible Correction Strategies**

Check the execution paths within the parallel site and confirm that you intended to have only one task for this parallel site. If needed, examine the loop structure and its scaling characteristics (reported by the Suitability tool) to ensure that this parallel site does not need additional tasks. If the problem is caused by using too small a data set, increase the size of your data set.

**Orphaned Task**

*Occurs when a task-begin annotation is executed that is not within an active parallel site.*
This example does not execute a `ANNOTATE_SITE_BEGIN(sitename)/ANNOTATE_SITE_END()` annotation pair required to wrap the execution of a task `ANNOTATE_TASK_BEGIN(taskname)/ANNOTATE_TASK_END()` annotation pair.

**Possible Correction Strategies**

Always execute an `ANNOTATE_SITE_BEGIN(sitename)` annotation before executing an `ANNOTATE_TASK_BEGIN(taskname)` annotation. You may need to add an `ANNOTATE_SITE_BEGIN(sitename)/ANNOTATE_SITE_END()` annotation pair in the same function, or in some calling function.

An orphaned task is effectively ignored by the Suitability and Dependencies tool analysis, so you should fix the orphaned task code and run the Suitability and Dependencies tools again.

**Parallel Site Information**

*Occurs when execution enters a parallel site.* This confirms that your program and its data are executing the annotations you inserted during execution of the Dependencies tool analysis. In this case, this message provides informational feedback only and no action is required.

**Syntax**

<table>
<thead>
<tr>
<th>ID</th>
<th>Code Location</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Parallel site</td>
<td>Represents the location and associated call stack of the parallel site.</td>
</tr>
</tbody>
</table>

**Example**

```c
ANNOTATE_SITE_BEGIN(name_site1); // Begin parallel site
for (i = 0; i < n; ++i)
{
    ANNOTATE_ITERATION_TASK(name_task1);
    process(i);
}
ANNOTATE_SITE_END(); // End parallel site
```

**See Also**

Dependencies Tool Overview

**Thread Information**

*Occurs when a thread is created.* In this case, this message provides informational feedback and no action is required.

Expect at least one such message when the main program's thread is created by the operating system. If you are running the Dependencies tool on a partially paralleled target, expect additional messages for each thread the program creates.

The creation site of the main program thread is the point where the `main()` function - or other standard entry point line `_wmain()` - is called from the startup initialization code.
Syntax

![Diagram: Creation site]

<table>
<thead>
<tr>
<th>ID</th>
<th>Code Location</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Creation site</td>
<td>Represents the location and call stack where a thread was created.</td>
</tr>
</tbody>
</table>

**Example**

ANNOTATE_SITE_BEGIN(name1);
for (int i = 0; i < n; ++i)
{
    ANNOTATE_ITERATION_TASK(task_process_array);
    process(array1[i]);
}
ANNOTATE_SITE_END();

// create thread using parallel framework code or CreateThread()
for (int i = 0; i < n; ++i)
{
    process(array2[i]);
}

**Unhandled Application Exception**

*Occurs when an unhandled exception is detected that causes the application program to crash.*

Syntax

![Diagram: Exception]

<table>
<thead>
<tr>
<th>ID</th>
<th>Code Location</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Exception</td>
<td>Represents the instruction that threw the exception.</td>
</tr>
</tbody>
</table>

**Example**

```c
void problem1(int *y)
{
    *y = 5;
}

void problem2()
{
    int x = new int;
}
```

In these (simplified) example functions, two exceptions are possible:
Variable $y$ may not reference a valid memory location and therefore the write may cause an exception to be thrown. If that exception is not properly handled, the Dependencies Report will show an Unhandled application exception pointing to the write of $y$.

If the process is out of memory, the allocation will throw an exception. If the exception is not handled, the Dependencies Report will show an Unhandled application exception associated with the allocation.

Because of the abnormal process termination (crash), the Dependencies tool may also report a Missing end task and Missing end site problem.

Possible Correction Strategies

This problem usually exposes an existing bug in your application that appears when the application is run with the Dependencies tool.

See Also

Dependencies Tool Overview

Troubleshooting Reference

You may encounter unexpected situations while using Intel® Advisor. This reference section describes symptoms and possible correction strategies for several such situations.

Analysis of Debug Build

Symptoms

A message appears when you start the Survey or Suitability tools and the current build options selected for the project is a Debug build.

Details

When measuring performance, using a version of the program for analysis that is close to the version that will be provided to customers provides the most accurate data. If you will provide a Release build for customers, run the Survey or Suitability tools with a Release build. You can run this tool with a Debug build if you will run it again with a Release build. To produce the best results, your build settings should specify debug information and moderate optimization.

When a tool is waiting for your input (click either Continue or Cancel), the result name has a [!] prefix. If you do not respond within several minutes, the tool implicitly chooses the Cancel button.

Cause

For the Survey or Suitability tools, you should use a Release build of your program, not a Debug build.

Possible Solution

- Before you respond to the message, change to Release build settings and build the target executable. When it completes, click Continue to run the analysis.
- Click Continue and ignore this message. Later, run this tool again with a target built using a Release build.
- Click Cancel. Change your build settings to use a Release build and build the target executable. Then run the Intel® Advisor tool.
Analysis of Release Build

Symptoms
A message appears when you start the Dependencies tool and the current build options selected for the project is a Release build.

Details
To produce the best results, your build settings for the Dependencies tool should specify debug information and no optimization. If possible, use a minimal data set for the Dependencies tool.

When a tool is waiting for your input (click either Continue or Cancel), the result name has a [!] prefix. If you do not respond within several minutes, the tool implicitly chooses the Cancel button.

Cause
For the Dependencies tool, you should use a Debug build of your program, not a Release build.

Possible Solution
• Before you respond to the message, change to use Debug build settings and build the target executable. When it completes, click Continue to run the analysis.
• Click Continue and ignore this message. Later, run this tool again with a target built from Debug build settings.
• Click Cancel. Change to use Debug build settings and build the target executable. Then run the Dependencies tool.

See Also
Debug Information Not Available
No Annotations Found

Debug Information Not Available

Symptoms
After running the Intel® Advisor analysis, the displayed report may contain information about your target's debug (symbol) information that is unexpected or does not make sense.

Details
When debug information is not available, the ability to use binary-to-source correlation prevents the display of source code. One or more of the following might occur only for the calls into third-party library routine code for which library sources are not available to the project:
• After running the Survey tool, a message may appear near the top of the Survey Report window indicating **Some target modules do not contain debug information**. After viewing the message and writing down module names that lack debug information, you can click the red X in the top-right corner to close it.

• When viewing a Report window, a column that should contain a Source location or a function name instead contains the target's executable-name in square brackets, [Unknown], a question mark ?, or is blank. Also, a broken or missing icon can indicate that sources are not available, such as ![source](image), ![source](image), ![source](image), or ![source](image) for a Survey loop.

• When viewing a Source window, a column that should contain source code or a source location instead contains **source is not available** or **Intel Advisor cannot show source code for this location** message instead of the expected data. Also, a broken or missing icon can indicate that sources are not available (listed above).

• When viewing a Source window, the Call Stack indicates that sources are not available for the starting (top) line, such as containing a broken icon (listed above).

• In a context menu, the items **View Source** or **Edit Source** may appear as dimmed.

However, if your application calls library functions, you should expect to see some symptoms about sources not being available. For example, the C/C++ sample application stats calls certain library functions to calculate standard deviation or similar values. Because the source code for the library functions is not available, you will see that source code is not available within the called library functions, but is available for the related project source files. In this case, see the help topic Sources Not Available.

**Cause**

The most common causes are:

• The build option(s) did not request debug information when building the target executable. Debug information must be present for Intel® Advisor tools to display source information. When building native code targets, specify the appropriate compiler and linker options to ensure the target executable contains debug information.

• The appropriate project properties in the Intel® Advisor GUI did not provide the correct binary/symbol or source locations.

**Possible Solution**

• Do the following:

  1. Check the build settings for the target to ensure they specify debug information option(s).
  2. Adjust your makefile, Microsoft Visual Studio* project properties, or build script to specify debug information option(s).
  3. Rebuild the target.
  4. Run the Intel® Advisor analysis tool(s) again.

• If you are using the Intel® Advisor GUI, check the **Project Properties** dialog to make sure that:

  • Binary and symbol files are specified in the **Binary/Symbol Search** tab.
  • Source locations are specified in the **Source Search** tab.

• Investigate other possible causes, such as the compiler not generating debug information for a source line or the source file, the linker not including debug information in the debug information database, or whether the debug information database was not being found during the finalization step by an Intel® Advisor analysis tool. For example, the last issue can occur if the debug information database was not moved to the location with the target executable.

**Tip**

For the most current information on optimal C/C++ and Fortran build settings, see **Build Your Target Application**.
No Annotations Found

Symptoms
- A message appears when you click the Annotation Report button and you have not yet added Intel®
  Advisor annotations to your program, or you have not yet run the Suitability and Dependencies tools after
  adding the annotations.
- When using the Intel® Advisor GUI while viewing the Summary window, you see a message No
  annotations detected in your project sources when your sources do contain annotations.

Cause
The Suitability and Dependencies tools use the annotations you added to your program to analyze your
running program and populate the Annotation Report window with data. However, before you run these
tools to collect data about your running program, you need to add annotations and perform related actions.

When using the Intel® Advisor GUI, make sure that the appropriate project properties have been specified so
the Intel® Advisor tools can find the correct source location(s).

Also, if your sources contain huge source files that contain annotations, be aware that only the first 8 MB of
each file will be parsed for annotations (for performance reasons). This could possibly cause mismatched or
no annotations found messages.

Possible Solution
- Do the following:
  1. Use the Survey analysis to find where your program spends its time. Choose at least one possible
     parallel code region (site) and identify code that might execute independently as a task.
  2. Use the code editor to add at least one pair of parallel site annotations that contain task annotation(s)
     into your program. You can copy annotation code using the bottom of the Survey Report or Survey
     Source windows.
  3. Make sure that these annotations are executed by the selected project or the selected startup project
     (Windows® OS).
  4. Make sure that you reference the annotations definitions file in the source modules where you added
     the annotations.
  5. Reference the annotations definition directory and provide other build settings.
  6. If you are using the Intel® Advisor GUI, check the Project Properties dialog to make sure that source
     locations are specified in the Source Search tab.
  7. If your sources include huge source files that contain annotations (more than 8 MB per file), consider
     breaking each huge source file into several source files.
- Windows OS only: If you selected the wrong startup project, select and build the correct startup project, run either the Suitability or Dependencies tool, and click the Annotation Report button.

Tip
For the most current information on optimal C/C++ and Fortran build settings, see Build Your Target
Application.

See Also
No Data
Source Search Tab
Copying Annotations and Build Settings Using the Annotation Assistant Pane
Survey Analysis
Annotating Code for Deeper Analysis
Intel Advisor Annotations Definition File
Annotations
Intel Advisor Workflow Tab

No Data

Symptoms
A **No Data** message appears when you click the **Survey Report**, **Suitability Report**, or the **Dependencies Report** button and you have not yet run these tools for the currently selected project or startup project (Windows* OS).

This message also appears if Intel® Advisor annotations were not executed by the Suitability or Dependencies tools. When using the Intel® Advisor GUI, this message may appear if the Suitability or Dependencies tools could not find the source files using the specified project properties.

To help you add annotations to your sources, the **No Data** message is accompanied by the annotation assistant pane.

Cause
After you run the Suitability or Dependencies tools, the data collected populates the corresponding **Suitability Report** and **Dependencies Report** window, and the list of annotations displayed in the **Annotation Report** window is updated.

- To use the Suitability or Dependencies tools, your project/startup project must execute Intel® Advisor parallel site and task annotations.
- When using the Intel® Advisor GUI, this message appears when the specified project properties do not provide a correct path to the source location(s).
- This message can appear with the Survey tool if your target executes quickly and you clicked the **Started Paused** button (or equivalent option or Pause Collection annotation) in the side command toolbar. That is, you paused data collection so that data collection did not start until after the target's execution completed, but the target executes too quickly for the Survey tool to analyze.

Possible Solution
- Use the **Advisor Workflow** tab to guide you through the steps needed to run these tools, which analyze your running program.
- Windows* OS only: If you selected the wrong startup project, select and build the correct startup project and run the tool again.
- If your project/startup project did not execute Intel® Advisor annotations, make sure you have added parallel site and task annotations to your program and that they get executed when you run the startup project (Windows* OS) / target executable (Linux* OS).
- When using the Intel® Advisor GUI, open the project and then open **Project Properties** dialog box. After checking that the path and target file name of the **Application** is correct in the **Analysis Target** tab, click the **Source Search** tab. Insert one or more new rows to specify the path to the source location(s). In this case, you do not need to rebuild your application.
- If your target executes quickly and you clicked the **Started Paused** button (or used the equivalent option or Pause Collection annotation) in the side command toolbar, click **Collect Survey Data** instead. Otherwise, the Survey tool cannot analyzer your target because it executes too quickly.
Source Not Available

Symptoms
After running the Intel® Advisor tools to analyze your running application's target, the displayed report may contain information about your target's source code that is unexpected or does not make sense.

Details

NOTE
When you see that source code is not available, be aware that improper build settings can also cause this symptom. Missing debug information symbols disable the binary-to-source correlation that allows the normal display of source code. If the source code for annotations is not displayed in Report and Source windows, this may indicate missing debug information symbols, as explained in the help topic Troubleshooting Debug Information Not Available.

One or more of the following might occur only for the calls into third-party library routine code for which library sources are not available to the project:

• When viewing a Report window, a column that should contain either a source location or a function name instead contains a question mark (?), [Unknown], the target's executable-name (rather than a source file name), is blank, or contains a broken icon, such as , , , or for a Survey loop.

• When viewing a Source window, you may see a message Intel Advisor cannot show source code for this location instead of the collected data. Also the Call Stack pane or a Function column may contain ?, [Unknown], or a broken icon (listed above). For example, if the top (starting) function line in a Call Stack pane contains source information, but calls to library routines lower in the calls stack do not contain source information.

• When using the Intel® Advisor GUI while viewing the Summary window, you see a message No annotations detected in your project sources when your sources do contain Intel® Advisor annotations.

If your application's target calls libraries for which sources are not found by the project, you may see some symptoms about source code not being available. For example, the C/C++ sample application stats calls certain library routines to calculate standard deviation or similar values. Because the library's source code for those functions is not available, the symptoms like a broken icon or the message Intel Advisor cannot show source code for this location are expected.

Cause
The most common causes are:

• The target's execution paths called library functions for which sources are not available.

• The cause may be that debug information symbols are not available for the main part of the program and annotated parallel sites and their tasks.

Possible Solution

• Do the following:

  1. If source and debug information symbols are available for the annotated parallel sites and their tasks, but not for calls into library code for which sources are not available, this is expected and is not a problem. If you were considering adding annotations to the library code, instead add site/task annotations to the code that calls the library routines.
2. If source and debug information symbols are not available for the main part of the program and annotated parallel sites and their tasks, see the help topic Troubleshooting Debug Information Not Available.

Tip
For the most current information on optimal C/C++ and Fortran build settings, see Build Your Target Application.

See Also
Survey Tool does not Display Survey Report
Debug Information Not Available

Survey Tool does not Display Survey Report

Symptoms
After you run the Survey tool, a message appears instead of the Survey Report. The message indicates that the specified target runs too quickly for the Survey tool to analyze or that the target does not contain debug symbol information.

Cause
After you run the Intel® Advisor Survey tool, if the specified target runs too quickly for the Survey tool to analyze using the sampling interval, there is insufficient data collected to provide a complete and meaningful Survey Report. So a message appears instead.

This message can also appear under certain conditions when the target was built without the required Debug symbol information.

Possible Solution
If the cause is that the specified target runs too quickly:

- Modify the target to increase its workload or data set so it executes longer, rebuild, and run the Survey tool again.
- Specify a different target that executes longer, build, and run the Survey tool again.
- When using the Intel Advisor GUI, if you cannot increase the target's workload or the data set to increase its execution time, consider specifying the Project Properties in the Analysis Target dialog box Advanced fields to slightly reduce the Sampling interval for this target. However, if you reduce the sampling interval so sampling occurs too often, this can cause the sampling activity itself to be measured (noise), reducing the accuracy of the measurements. The default Sampling interval works for most targets, but can be modified for specific targets (see Advanced options in the help topic Dialog Box: Project Properties - Analysis Target).
- Windows* OS only: If you selected the wrong Visual Studio startup project, select and build the correct startup project and run the tool again.

If you suspect that the target executable does not contain debug symbol information, please check your target's build settings and compare them with the recommended Build Settings for your language. Source- or target-related information may be missing from the Intel Advisor tools reports if the target executable does not contain debug symbol information.

Tip
For the most current information on optimal C/C++ and Fortran build settings, see Build Your Target Application.
Undefined Linker References to dlopen or dlsym

Symptoms
When linking your application program on Linux OS, you see linker (`ld`) messages such as:

- undefined reference to `dlopen`
- undefined reference to `dlsym`

Cause
Intel® Advisor uses dynamic loading. After you add the `#include` (C/C++) line to include the Intel Advisor annotation definition file, you must specify the linker option `-ldl` to enable dynamic loading.

Possible Solution
- Do the following:
  1. Add the linker option `-ldl` to your command line, script, or make file.
  2. Review the options listed in the Build Your Target Application to ensure that you specified all required compiler and linker options (use the link below under See Also). If omitted, add missing options to your command line, script, or make file.
  3. Rebuild your program.

See Also
Unexpected C/C++ Compilation Errors After Adding Annotations
Intel Advisor Annotations Definition File

Unexpected C/C++ Compilation Errors After Adding Annotations

Symptoms
After adding Intel® Advisor annotations, you see unexpected compiler messages when building your C/C++ target executable.

Details
After you add the annotations and the `#include` line to include the Intel Advisor annotation definition file, you see unexpected C/C++ compiler messages.

Cause
Possible causes:
- Type and debug symbol conflicts.
- `windows.h` file issues.

Possible Solution
Do the following:

1. Read the help topics starting with Tips for Annotation Use with C/C++ Programs to help you decide how to modify your sources, such as Handling Compilation Issues that Appear After Adding advisor-annotate.h.
2. Modify sources.
3. Rebuild your target.
Unexpected Unmatched Annotations in the Dependencies Report

Symptoms
When running Intel® Advisor Dependencies tool analysis, you see unmatched problems reported that are caused by unmatched annotations execution that you did not expect.

Details
The Dependencies Report window lists the problems and messages reported by Dependencies tool analysis in the Problems and Messages pane. You may see some unexpected problems related to unmatched annotations, such as the following problem types: Dangling Lock, Missing Begin Site, Missing Begin Task, Missing End Site, Missing End Task, or Orphaned Task. For example, within a parallel site, there is no annotation to mark the end of the parallel site for all possible the code execution paths.

Cause
Possible causes:

- You placed annotations inside macros.
- For Linux* OS: You placed parallel sites and their related annotations outside the project.
- For Windows* OS: You placed parallel sites and their related annotations outside a set of projects that the startup project depends on.
- Your sources contain huge source files that contain annotations. As only the first 8 MB of each file are parsed for annotations (for performance reasons), this could possibly cause mismatched annotations messages.

Possible Solution
Do the following:

1. Use the Dependencies tool to view the code region(s) causing the problem. Investigate whether some sites or tasks may have multiple exit points and whether end annotations cover all exit points. For example, code that returns or branches around an end annotation, or throws an exception. If you suspect the problem is caused by adding annotations inside macros, remove the annotations from the macros and add them to the final location in the sources - similar to the way breakpoints do not work in macros. Rebuild your target and run the Dependencies tool again.

2. Windows* OS only: Use the Dependencies tool to investigate by viewing the code region(s) causing the problem. If you suspect that parallel sites and their related annotations that are placed outside the set of projects that the startup project depends on, consider using the Visual Studio® Project Dependencies context menu item to add appropriate dependencies to cause Intel Advisor to scan sources in the additional project(s). Rebuild and run the Dependencies tool again.

3. If your sources include huge source files that contain annotations (more than 8 MB per file), consider breaking each huge source file into several source files.

See Also
Survey Analysis
Annotating Code for Deeper Analysis
Annotations
Annotation General Characteristics
Intel Advisor General Characteristics
Getting Help and Support

This topic explains the different options for accessing the Help documentation and technical support for Intel® Advisor.

Getting Help

The documents provided with this release are available in HTML format. You can access the documentation:

- **For Windows® OS only**: From the Start menu, or Start screen, under the Intel Parallel Studio XE [version] group.
- **Help > Intel Advisor [version]**
- Access context-sensitive Help on active GUI elements:
  - In the Advisor Workflow tab and in the Result tab, click certain links to get specific help related to the underlined word.
  - In the Result tab, you can right-click an element to display its context menu. Certain context menus display a What Should I Do Next? menu item. Choose this menu item to get help specific to the active user interface element.
  - **F1 Help**: Press F1 to get help for an active dialog box, property page, pane, or window.

Getting Support

The following links provide information and support on Intel® software products, including developer suite products such as Intel® Parallel Studio XE suites:

  At this site, you will find comprehensive product information, including:
  - Links to each product, where you will find technical information such as white papers and articles
  - Links to user forums
  - Links to news and events
  Intel® Developer Zone, product support page, with links to support forums, startup help, knowledge base, product documentation, and getting started.

For detailed system requirements and additional support information, see the product Release Notes.

Glossary

**Amdahl's law**: A theoretical formula for predicting the maximum performance benefits of parallelizing application programs. Amdahl's law states that run-time execution time speedup is limited by the part of the program that is not parallelized (executes serially). To achieve results close to this potential, overhead must be minimized and all cores need to be fully utilized. See also Using Amdahl's Law and Measuring the Program.

**annotation**: A method of conveying information about proposed parallel execution. In the Intel® Advisor, you create annotations by adding macros or function calls. These annotations are used by Intel Advisor tools to predict parallel execution. For example, the C/C++ ANNOUNCE_SITE_BEGIN(sitename) macro identifies where a parallel site begins. Later, to allow this code to execute in parallel, you replace the annotations with code needed to use a parallel framework. See also parallel framework and Annotation Types Summary.

**atomic operation**: An operation performed by a thread on a memory location(s) that is guaranteed not to be interfered with by other threads. See also synchronization.

**chunking**: The ability of a parallel framework to aggregate multiple instances of a task into groups for more efficient parallel processing. For tasks that do small amounts of computation and many iterations, task chunking can minimize task overhead. You can also restructure a single loop into an inner and outer loop (strip-mining). See also task and Enabling Task Chunking.
**code region:** A subtree of loops/functions in a call tree. Synonym whole Loopnest.

**critical section:** A synchronization construct that allows only one thread to enter its associated code region at a time. Critical sections enforce mutual exclusion on enclosed regions of code. With Intel Advisor, mark critical sections by using `ANNOTATE_LOCK_ACQUIRE()` and `ANNOTATE_LOCK_RELEASE()` annotations.

**data race:** When multiple threads share (read/write) a memory location, if the program does not implement controls to manage the sequence of concurrent memory accesses, one thread can inadvertently overwrite data written by another thread, or otherwise read or write stale data. This can produce execution errors that are difficult to detect and reproduce, such as obtaining different calculated results when the same executable is run on different systems. To prevent data races, you can add data synchronization constructs that restrict shared memory access to one thread at a time, or you might eliminate the sharing.

**data parallelism:** Occurs when a single portion of code is paired with multiple portions of data, and each pairing executes as a task. For example, tasks are made by pairing a loop body with each element of an array iterated by the loop, and the tasks execute in parallel. See also Task Patterns. Contrast task parallelism.

**data set:** A set of data to be used as input or with an interactive application the way you interact with the application to cause a portion of the application to be executed. Because the Dependencies tool watches each memory access in a parallel site in great detail, the parallel site's code takes much longer to run than usual. To limit the time needed to run Dependencies analysis, reduce the data (such as the number of loop iterations) and when using an interactive program, create a very small test case. See also Choosing a Small, Representable Data Set for the Dependencies Tool.

**deadlock:** A situation where a set of threads have each acquired some locks and are waiting for other locks to be released. All threads in the set are waiting for a lock held by a different thread, and since none can proceed and release their lock(s), they all remain waiting.

**dynamic extent:** All code that may possibly be executed by a parallel site or task. For example, a dynamic extent might include a loop, all functions called from the loop, all functions the called functions may in turn call, and so on. Contrast static extent. See also Task Organization and Annotations.

**false positive:** When viewing the Dependencies Report, a problem reported by the Dependencies tool that is not an actual problem.

**framework:** See parallel framework

**head:** A loop or function at the top of a subtree, which contains one or more child loops/functions.

**hotspot:** A small code region that consumes much of the program's run time. Hotspots can be identified by a profiler, such as the Intel Advisor Survey tool. See also Using Amdahl's Law and Measuring the Program.

**Intel® Threading Building Blocks (Intel® TBB):** A C++ template library for writing programs that take advantage of multiple cores. You can use this library to write scalable programs that specify tasks rather than threads, emphasize data parallel programming, and take advantage of concurrent collections and parallel algorithms. This is provided as an Intel® software product - Intel® Threading Building Blocks (Intel® TBB) - as well as open source. Intel® Threading Building Blocks (Intel® TBB) is one of several parallel frameworks. Abbreviation Intel TBB.

**load balancing:** The equal division of work among cores. If the load is balanced, the cores are busy most of the time.

**lock:** A synchronization mechanism that allows one thread to wait until another thread allows it to continue. A lock can be used to synchronize threads accessing a specific memory location. See also synchronization and nested lock.

**multi-core:** A processor that combines two or more independent cores. Although each core shares interconnection to the rest of the system, it executes instructions independently by using its dedicated CPU, architectural state, and interrupt controllers, as well as private and/or shared cache. Most multi-core systems use identical cores. The number of cores used determines whether it is called dual-core (2), quad-core (4), or many-core system.

**multithreaded processing:** See parallel processing

**mutual exclusion:** A type of locking typically used to prevent actions occurring at the same time. Abbreviation mutex. See also synchronization
**nested lock**: A type of `lock` that can be locked again by a task when the task already owns the lock. Nested locks are convenient when several inter-related functions use the same lock. See also `synchronization` and `lock`.

**node**: A loop or function.

**OpenMP**: A high-level parallel framework and language extension designed to support shared-memory parallel programming that consists of compiler directives (C/C++ pragmas and Fortran directives), library functions, and environment variables. The OpenMP specification was developed by multiple hardware and software vendors to provide a scalable, portable interface for parallel programming on a variety of platforms. OpenMP is one of several parallel frameworks. See also http://openmp.org.

**parallel framework**: A combination of libraries, language features, or other software techniques that enable code for a program to execute in parallel. Examples include OpenMP, Intel® Threading Building Blocks (Intel® TBB), Message Passing Interface (MPI), Intel® Concurrent Collections for C/C++, Microsoft Task Parallel Library* (TPL), and low-level, basic threading APIs, like POSIX* threads (Pthreads). Some parallel frameworks support shared-memory parallel processing, while others like MPI support non-shared-memory parallel processing. See also Intel® Threading Building Blocks (Intel® TBB) and Parallel Frameworks Overview.

**parallel processing**: The use of multiple threads during execution of a program. Intel Advisor focuses on parallel processing for `shared-memory systems`. There are other types of parallel processing, such as for clusters or grids and vector processing. Shortened version is `parallelism`. See also `hotspot` and `thread`.

**parallel region**: Offload Advisor term. A code region that starts with a specific parallel framework construction. Treading Building Blocks (TBB), Intel® Data Analytics Acceleration Library (Intel® DAAL), OpenMP*, Data Parallel C++ (DPC++) parallel frameworks are supported.

**parallel site**: A region of code that contains tasks that can execute in parallel. See also `annotation` and Task Organization and Annotations.

**pipeline**: An approach to organizing task computations that uses both data parallelism and task parallelism, and organizes the computation into stages that run in a predetermined order.

**self time**: In the Survey Report window, how much time was spent in a particular function or loop.

**site**: See parallel site

**shared-memory parallelism**: See parallel processing

**static extent**: The code between a site’s or a task’s `_BEGIN` and `_END` annotations. A static extent might not be lexically paired; for example, a parallel site may have one `_BEGIN` point, but may require multiple independent `_END` exit points. Contrast with dynamic extent. See also annotation, parallel site, and Task Organization and Annotations.

**synchronization**: Coordinating the execution of multiple threads. In some cases, you can provide synchronization within a task by using a private memory location instead of a shared memory location. In other cases, a `lock` or `mutex` can be used to restrict access to a shared data. See also Data Sharing Problem Types.

**task**: A portion of code and its data that can be given to a thread to execute. See also Task Organization and Annotations, Choosing the Tasks, and chunking.

**task parallelism**: Occurs when two different portions of the code are made into tasks and execute in parallel. For example, a task is made by pairing a display algorithm with the state to display, another task by pairing a compute-next-state algorithm with the same state, and the two tasks execute in parallel. See also Task Patterns. Contrast data parallelism

**Intel TBB**: See Intel® Threading Building Blocks (Intel® TBB)

**thread**: A thread executes instructions within a process. Each process has one or more threads active at a time. Threads share the address space of the process, but have their own stack, program counters, and other registers.

**total time**: In the Survey Report window, how much time was spent in a particular function or loop, plus the time spent by anything that entity calls.

**vector processing**: A form of parallel processing where multiple data items are packed together in vector registers to allow vector instructions to operate on the packed data with a single instruction. Reducing the number of instructions needed to process the packed vector data minimizes memory use and latency, and
provides good locality of reference and data cache utilization. Vector instructions are Single Instruction Multiple Data (SIMD) instructions. Some SIMD vector instructions support large register sizes to accommodate more packed data, such as Intel® Advanced Vector Extensions (Intel® AVX).

**Key Concepts**

This group of topics introduces you to the key concepts and terms needed to add parallelism to a program. A list of key terms is also provided.

Over the last few years, processor technology found in personal laptops, desktops, and enterprise servers has shifted from making single-core processors faster to having multiple cores in each processor.

In a parallel program, portions of the program (tasks) may execute at the same time. On multi-core systems, this can provide better performance.

To parallelize your application, you need to identify the potential parallel tasks, modify your code to run correctly when these tasks execute in parallel, and add code to execute them in parallel. Intel® Advisor combines a methodology with a set of tools to help you add this parallelism to your program. You work on the sequential version of your program and the tools model how it would behave if it was parallelized in the ways you specify. As an add-in to Microsoft Visual Studio®, Intel Advisor fits right into a Windows* OS development environment.

Your final step will be to express the parallelism in your program using a high-level parallel framework (threading model) like Intel® Threading Building Blocks (Intel® TBB) or OpenMP*, or low-level threading APIs.

For native C/C++ or Fortran code, Intel recommends using the high-level Intel TBB or OpenMP frameworks, which are included with most Intel® Parallel Studio XE editions. Intel Advisor's documentation shows you how to introduce parallelism into your program using these frameworks. Intel Advisor provides multiple C/C++ samples and several Fortran samples.

For managed C# code on Windows* OS, use the Microsoft Task Parallel Library* (TPL). Intel Advisor provides a C# nqueens sample.

This is an interactive process, where you repeat these basic steps as you identify more sites for adding parallelism.

The related group of topics provides an introduction to parallelism, and to parallel framework implementations.

**See Also**

Parallelism

Glossary

**Notational Conventions**

The following conventions may be used in this document.

<table>
<thead>
<tr>
<th>Convention</th>
<th>Explanation</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td><em>Italic</em></td>
<td>Used for introducing new terms, denotation of terms, placeholders, or titles of manuals.</td>
<td>The filename consists of the <em>basename</em> and the <em>extension</em>. For more information, see the Intel® Advisor User Guide.</td>
</tr>
<tr>
<td><em>Bold</em></td>
<td>Denotes GUI elements</td>
<td>Click <em>Cancel</em>.</td>
</tr>
<tr>
<td>&gt;</td>
<td>Indicates a menu item inside a menu.</td>
<td><em>File &gt; Close</em> indicates to select <em>Close</em> from the <em>File</em> menu.</td>
</tr>
</tbody>
</table>
### Related Information

A variety of resources provide additional information on a number of topics.

You are strongly encouraged to read the following books for in-depth understanding of parallelism (multi-threaded execution). These books discuss general concepts of parallel programming by explaining a particular programming technology:

<table>
<thead>
<tr>
<th>Technology</th>
<th>Resource</th>
</tr>
</thead>
<tbody>
<tr>
<td>OpenMP*</td>
<td>Chapman et al. <em>Using OpenMP: Portable Shared Memory Parallel Programming</em> and Chandra, Rohit, et al. <em>Parallel Programming in OpenMP</em></td>
</tr>
</tbody>
</table>

The following book describes Intel® Parallel Studio XE and will help you understand the software components of Intel® Parallel Studio XE:

<table>
<thead>
<tr>
<th>Technology</th>
<th>Resource</th>
</tr>
</thead>
</table>

### Additional Reference Material

In addition, the following resources may be helpful:

<table>
<thead>
<tr>
<th>To View:</th>
<th>Access:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Intel® C++ Compiler Classic and Intel® Fortran Compiler Classic documentation</td>
<td>See the Intel® C++ Compiler Classic and Intel® Fortran Compiler Classic help.</td>
</tr>
<tr>
<td>Intel TBB documentation</td>
<td>See the Intel® C++ Compiler Classic documentation for Intel TBB .</td>
</tr>
<tr>
<td>Intel TBB documentation and resources</td>
<td><a href="https://www.threadingbuildingblocks.org/documentation">https://www.threadingbuildingblocks.org/documentation</a></td>
</tr>
</tbody>
</table>
### Parallelism

The following topics describe some key terms related to multithreaded parallel processing (parallelism), an overview of multithreaded parallelism, and common issues when adding multithreaded parallelism to your program:

- If you are just learning about adding multithreaded parallel processing to application programs, please read these topics carefully.
- If you have advanced knowledge about multithreaded parallel processing and are familiar with the concepts, quickly read (scan) these topics so you are familiar with the terms used.

### Parallel Processing Terminology

A serial (non-parallel) program uses a single thread, so you do not need to control the side-effects that can occur when multiple threads interact with shared resources.

A program takes time to run to completion. A serial program only uses a single core, so its run time will not decrease by running it on a system with multiple cores. However, if you add parallel processing (parallelism) to parts of the program, it can use more cores, so it finishes sooner.

### Threads and Tasks

An operating system process has an address space, open files, and other resources. A thread executes instructions within a process. Each process has one or more threads active at a time. Threads share the address space of the process, but have their own stack, program counter, and other registers. A program that uses multiple threads is called a multithreaded or parallel program.

A task is a portion of a program that can be run in parallel with other portions of the program and other instances of that task. Each task instance is run by a thread, and the operating system assigns threads to cores.

### Hotspots - Find Where a Program Spends Its Time

A hotspot is a small code region that consumes much of the program’s run time. You can use profiling tools such as the Survey tool provided with Intel Advisor to identify where your program spends it time. To improve your program’s performance when you add parallelism:

- Find the hotspots and hot parts of the call tree, such as hot loops or hot routines. The Intel Advisor Survey tool’s report provides an extended top-down call tree that identifies the top hot loops.
Examine all the functions in the call tree from `main()` to each hot routine or loop. You want to distribute frequently executed instructions to different tasks that can run at the same time.

**Data and Task Parallelism**

If the hot part of the call tree is caused by executing the same region of code many times, it may be possible to divide its execution by running multiple instances of its code, each on a separate core. This is called **data parallelism** because each execution is processing different parts of the same composite data item. Compute-intensive loops over arrays are often good candidates for data parallelism. For example, the line `process(a[i]);` below is a possible task:

```c
for (int i = 0; i != n; ++i) {
    process(a[i]);
}
```

If two or more hotspots are close to each other in the serial execution, and do not share data, it may be possible to execute the hotspots as tasks. This is **task parallelism**. For example:

```c
initialize(data);
while (!done) {
    old_data = data;
    display_on_screen(old_data);
    update(data);
}
```

Making effective use of multiple cores may require both data-level parallelism to process large amounts of data, and task-parallelism to overlap the execution of unrelated portions of the program.

**Adding Parallelism**

The best performance improvements from adding parallel execution (parallelism) to a program occur when many cores are busy most of the time doing useful work. Achieving this requires a lot of analysis, knowledge, and testing.

Because your serial program was not designed to allow parallel execution, as you convert parts of it to use parallel execution, you may encounter unexpected errors that occur only during parallel execution. Instead of wasting effort on portions of a program that use almost no CPU time, you must focus on the hotspots, and the functions between the main entry point and each hotspot.

If you naively add parallel execution to a program without proper preparation, unpredictable crashes, program hangs, and wrong answers can result from incorrect parallel task interactions. For example, you may need to add **synchronization** to avoid incorrect parallel task interactions, but this must be done carefully because locking overhead and serial synchronization can reduce the benefits of the parallel execution.

Intel Advisor helps you:

- Find the possible code regions where you could add parallel execution.
- Choose the code regions best-suited for parallel execution. This includes measuring approximate parallel performance so you can experiment with different possible parallel code regions.
- Find and eliminate potential data sharing problems before parallel execution is introduced.

**See Also**

*Common Issues When Adding Parallelism*

**Common Issues When Adding Parallelism**

The types of problems encountered by parallel programs include shared memory data conflicts and incorrect locking.
Shared Memory Problems

Introducing parallelism can result in unexpected problems when parallel tasks access the same memory location. Such problems are known as data races. For example, in the Primes sample, the following line calls the function `Tick()`:

```c
if (IsPrime(p)) Tick();
```

The called function `Tick()` increments the global variable `primes`:

```c
void Tick() { primes++; }
```

Consider the following scenario, where the value of `primes` is incremented only once instead of twice:

<table>
<thead>
<tr>
<th>Time</th>
<th>Thread 0</th>
<th>Thread 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>T1</td>
<td>Enters function <code>Tick()</code></td>
<td>Enters function <code>Tick()</code></td>
</tr>
<tr>
<td>T2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>T3</td>
<td>Load value of <code>primes</code></td>
<td>Load value of <code>primes</code></td>
</tr>
<tr>
<td>T4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>T5</td>
<td>Increment loaded value</td>
<td>Increment loaded value</td>
</tr>
<tr>
<td>T6</td>
<td>Store value of <code>primes</code></td>
<td>Store value of <code>primes</code></td>
</tr>
<tr>
<td>T7</td>
<td></td>
<td></td>
</tr>
<tr>
<td>T8</td>
<td>Store value of <code>primes</code></td>
<td></td>
</tr>
<tr>
<td>T9</td>
<td>Return</td>
<td>Return</td>
</tr>
<tr>
<td>T10</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

If you run this as a serial program, this problem does not occur. However, when you run it with multiple threads, the tasks may run in parallel and `primes` may not be incremented enough.

Such problems are non-deterministic, difficult to detect, and at first glance might seem to occur at random. The results can vary based on multiple factors, including the workload on the system, the data being processed, the number of cores, and the number of threads.

It is possible to use locks to restrict access to a shared memory location to one task at a time. However, all implementations of locks add overhead. It is more efficient to avoid the sharing by replicating the storage. This is possible if data values are not being communicated between the tasks, even though the memory locations are being reused.

Lock Problems

One thread (thread A) may have to wait for another thread (thread B) to release a lock before it can proceed. The core executing thread A is not performing useful work. This is a case of lock contention. In addition, thread B may be waiting for thread A to release a different lock before it can proceed. Such a condition is called a deadlock.

Like a data race, a deadlock can occur in a non-deterministic manner. It might occur only when certain factors exist, such as the workload on the system, the data being processed, or the number of threads.
Ensuring the Parallel Portions of a Program are Thread Safe

Intel® Advisor can detect many problems related to parallelism. Because it only analyzes the serial execution of your program, Intel Advisor cannot detect all possible errors. When you have finished using Intel Advisor to introduce parallelism into your program, you should use the Intel® Inspector and other Intel software suite products. These tools and using a debugger can detect parallelism problems that normal testing will not detect, and can also identify times when the cores are idle.

See Also
Parallel Programming Implementations
Using Intel® Inspector and Intel® VTune™ Profiler
Debugging Parallel Programs
Data Sharing Problem Types

Parallel Programming Implementations

There are two popular approaches for adding parallelism to programs. You can use either:

- A high-level parallel framework like Intel® Threading Building Blocks (Intel® TBB) or OpenMP®. Of these parallel frameworks for native code, Intel TBB supports C++ programs and OpenMP supports C, C++, or Fortran programs. For managed code on Windows* OS such as C#, use the Microsoft Task Parallel Library* (TPL).
- A low-level threading API like Windows* threads or POSIX* threads. In this case, you directly create and control threads at a low level. These implementations may not be as portable as high-level frameworks.

There are several reasons that Intel recommends using a high-level parallel framework:

- **Simplicity**: You do not have to code all the detailed operations required by the threading APIs. For example, the OpenMP® `#pragma omp parallel for` (or Fortran `!$OMP PARALLEL DO`) and the Intel TBB `parallel_for()` are designed to make it easy to parallelize a loop (see Reinders Ch. 3). With frameworks, you reason about tasks and the work to be done; with threads, you also need to decide how each thread will do its work.

- **Scalability**: The frameworks select the best number of threads to use for the available cores, and efficiently assign the tasks to the threads. This makes use of all the cores available on the current system.

- **Loop Scalability**: Intel TBB and OpenMP assign contiguous chunks of loop iterations to existing threads, amortizing the threading overhead across multiple iterations (see Intel TBB `grain size`: Reinders Ch. 3).

- **Automatic Load Balancing**: Intel TBB and OpenMP have features for automatically adjusting the grain size to spread work amongst the cores. In addition, when the loop iterations or parallel tasks do uneven amounts of work, the Intel TBB scheduler will dynamically reschedule the work to avoid idle cores.

To implement parallelism, you can use any parallel framework you are familiar with.

The high-level parallel frameworks available for each programming language include:

<table>
<thead>
<tr>
<th>Language</th>
<th>Available High-Level Parallel Frameworks</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>OpenMP</td>
</tr>
<tr>
<td>C++</td>
<td>Intel® Threading Building Blocks (Intel® TBB)</td>
</tr>
<tr>
<td>C#</td>
<td>Microsoft Task Parallel Library* (Windows* OS only)</td>
</tr>
<tr>
<td>Fortran</td>
<td>OpenMP</td>
</tr>
</tbody>
</table>

See Also
Parallel Frameworks
Other Parallel Frameworks
Data Sharing Problems

In a serial program, the order of the operations during program execution are known. However, when code executes as multiple parallel tasks, an operation can execute before, after, or simultaneously with an operation in the other task. For example, when parallel tasks access or modify a shared memory location, data sharing problems can occur.

The Intel® Advisor Dependencies tool performs extensive analysis of your running serial program to help you predict data sharing problems. Use the Dependencies Report window and the topics introduced by this section to help you understand and decide how to fix the reported data sharing problems.

For each data sharing problem, you can either:

• Modify the sources to fix incidental or accidental data sharing by privatizing shared data use. This type of data sharing occurs when tasks use the same memory location, but do not communicate about using that memory location. If the data written by one is not needed by the other, each task could use a private copy of the data.
• Add lock annotations to implement synchronization for independent updates. This type of sharing occurs when multiple tasks contribute to determining the final value of a memory location.
• Recognize that the order of the operations cannot change, and consider modifying the chosen parallel sites and their tasks. When shared data access must occur in the original sequential order, this is called true dependence.

The following sections explain how to understand and fix sharing problems.

Data Sharing Problem Types

A data sharing problem happens when two tasks access the same memory location, and the behavior of the program depends on the order of accesses. This group of topics describes two common data access patterns - incidental sharing and independent updates - that result in data sharing problems that are relatively easy to fix. The fixes are described in Problem Solving Strategies.

The task’s code is called its static extent. You need to understand all the data accesses that might be executed during the execution of the task. You are interested in accesses to memory locations in the dynamic extent of a task, which includes all functions called from the task’s static extent, all functions the called functions may in turn call, and so on.

Incidental Sharing

Sharing is incidental when tasks use the same memory location, but do not communicate any information using it.

The Basic Pattern

Suppose that a task always writes to a memory location before reading from it, and that the value that it writes is not read again outside the task. For example:

```c
extern int x;
// ...
ANNOTATE_SITE_BEGIN(site1);
for (i = 0; i != n; ++i) {
    ANNOTATE_ITERATION_TASK(task1);
    x = a[i];
    b[i] = x * b[i];
}
ANNOTATE_SITE_END(site1);
```

The variable x is both read and written in the task, so there will be a sharing problem when multiple copies of the task execute at the same time. For example:

1. Task 0 sets x to a[0].
2. Task 1 sets x to a[1].
3. Task 0 computes $x \times b[0]$.

What is interesting is that the sharing is incidental to the logic of the program. Each iteration of the loop uses $x$, but its use in each iteration is totally independent. Memory locations used in this way are called \textit{privatizable}, because giving each task its own private memory location will eliminate the sharing without changing the program behavior.

\textbf{Memory Allocators}

The use of dynamically allocated memory is a special case of incidental sharing. Consider this task:

```
ANNOTATE_TASK_BEGIN(task2);
Type *ptr = allocate_Type();
// some code that uses the object pointed to by ptr
free_Type(ptr);
ANNOTATE_TASK_END();
```

If \textit{allocate Type()} returns the same address to one task that was used and freed by another task, then those tasks will both access the same memory location, but the sharing is incidental. The memory allocator will never return a pointer to memory that has been allocated and not freed, so the tasks will not use the same dynamically allocated memory location at the same time, and the appearance of sharing is an illusion.

The Dependencies tool understands the standard memory allocators such as \texttt{C/C++ new/delete} and \texttt{malloc/free}, but it does not know about any custom memory allocators that your program might have. If your code has custom memory allocators, you can mark their uses with the special-purpose \texttt{C/C++ annotations} \texttt{ANNOTATE_RECORD_ALLOCATION} and \texttt{ANNOTATE_RECORD_DEALLOCATION}.

\textbf{See Also}

Independent Updates
Special-purpose Annotations

\textbf{Independent Updates}

Independent updates can occur when multiple tasks contribute to determining the final value of a memory location.

\textbf{The Basic Pattern}

Suppose that multiple tasks write to a memory location, that the value written by each task is computed using the previous value in that location, and that the order in which the tasks update the memory location does not matter.

For example, consider a loop that sums all the values in an array:

```
extern int x;
// ...
ANNOTATE_SITE_BEGIN(site1);
for (i = 0; i != n; ++i) {
    ANNOTATE_ITERATION_TASK(task1);
    x = x + a[i];
}
ANNOTATE_SITE_END(site1);
printf("%d\n", x);
```

The sharing problem looks like this:

1. Task 0 reads $x$.
2. Task 1 reads $x$.
3. Task 0 adds $a[0]$ to the value it read and stores the result back in $x$.
4. Task 1 adds $a[1]$ to the value it read and stores the result back in $x$, overwriting the value stored by task 0.
The important fact is that the updates of $x$ can be performed in any order. All you need to do is to make sure that no task can write to $x$ between the read from $x$ and the write to $x$ in any other task; the uses of $x$ in the tasks are otherwise independent.

**Reductions**

*Reductions* are a special case of the independent update pattern. The reduction pattern occurs when a loop combines a collection of values using a commutative, associative function.

In *Adding Parallelism to Your Program*, you will see that the Intel® Threading Building Blocks and OpenMP* parallel frameworks have special features for writing parallel reductions.

**Transactions**

In a more general form of this pattern, there may be multiple memory locations which must be updated together.

```c
void insert_node_in_list(T *new_node, T *insert_after)
{
    new_node->next = insert_after->next;
    new_node->prev = insert_after->next->prev;
    insert_after->next->prev = new_node;
    insert_after->next = new_node;
}
```

Two insertions must not occur simultaneously, but the insertions may occur in any order, as long as the final list order does not matter.

A collection of updates that must all occur together is referred to as a *transaction*.

**Guard Variables**

A special case is the use of a shared memory location to control some additional code. The update and the code that depends on it may be treated as a transaction.

```c
bool initialized = false;
void do_something()
{
    if (!initialized) {
        do_the_initialization();
        initialized = true;
    }
    do_the_real_work();
}
```

If `do_something()` is called from multiple tasks, then the sharing problem is:

1. Task 0 reads `initialized`, which is false, and enters the body of the if statement.
2. Task 1 reads `initialized`, which is false, and enters the body of the if statement, so `do_the_initialization()` is called twice.

It does not matter which task the initialization occurs in, so your only problem is to make sure that other tasks wait until this initialization has happened.
Independent Writes

The simplest case occurs when the value that the tasks write to the memory location does not depend on its previous value:

```c
bool found = false;
ANNOTATE_SITE_BEGIN(site1);
for (i = 0; i != n; ++i) {
    ANNOTATE_ITERATION_TASK(task1);
    if (a[i] == b) found = true;
}
if (found) printf("found\n");
ANNOTATE_SITE_BEGIN(site1);
```

There is no read to keep together with the write, and it does not matter what order the writes to found occur in, so the tasks are totally independent, and can execute concurrently without restrictions. If a task writes to found at all, it will write the value true.

Note that if the task body were the following, then this example would fit the reduction pattern:

```c
found = found || (a[i] == b);
```

This is also called a benign race because the program will always compute the same value, regardless of which thread does the last write.

See Also

Problem Solving Strategies
Adding Parallelism to Your Program
Eliminating Incidental Sharing

Problem Solving Strategies

Data Sharing Problem Types describes the kinds of problems that can occur when tasks access the same memory locations. Two common strategies are used to deal with the following sharing problems:

- **Incidental sharing**: If a memory location is shared, but it is not used to communicate data between tasks, then you can eliminate the sharing by giving each task its own copy of the shared memory. This rarely causes significant increases in execution time or memory consumption. See Eliminating Incidental Sharing.

- **Independent updates**: If the reads and writes of the memory location occur in updates which can be done in any order, then you can add synchronization code to guarantee that the updates and related code in different tasks cannot be intermingled. This can increase execution time because only one task at a time can be accessing the shared memory location. This limits parallel execution. See Synchronizing Independent Updates.

If neither of these applies, you might be able to restructure your program to avoid the sharing problem; otherwise you may have to change your task structure. See Difficult Problems: Choosing a Different Set of Tasks.

Eliminating Incidental Sharing

Sharing problems involving a task and a memory location are incidental if the memory location does not carry information into or out of the task. Therefore, if you replace all uses of the shared memory location in the task with uses of some non-shared memory location, you eliminate the sharing problem without changing the behavior of the program.

The following sections describe incidental sharing problems and their solutions.

Examine the Task's Static and Dynamic Extent

Consider the example of incidental sharing from the help topic Incidental Sharing:
Examining the Static Extent of the Task

If you define a substitute variable inside the static extent, then each task will get its own private storage for it:

```c
extern int x;
// ...
ANNOTATE_SITE_BEGIN(site2);
for (i = 0; i != n; ++i) {
    ANNOTATE_ITERATION_TASK(task2);
    int x_sub;
    x_sub = a[i];
    b[i] = x_sub * b[i];
}
ANNOTATE_SITE_END();
```

Examining the Dynamic Extent of the Task

In the simplest cases, like the example above, the task’s dynamic extent is the same as its static extent - it does not contain any function calls. When it does contain function calls, all the functions that might be called while the task is executing are part of its dynamic extent, and you need to consider all reads and writes of the memory location in all of those functions.

So, you need to examine not only the static extent, but also the dynamic extent of a task.

See Also

Verify Whether Incidental Sharing Exists
Data Sharing Problem Types

Verify Whether Incidental Sharing Exists

Sharing is incidental only if the task writes to the memory location before any read of the memory location anywhere in the dynamic extent of the task. This is easy to check when the task is a few lines of code in a single function. It is much harder when the task is hundreds or thousands of lines of code, and involves calls to many functions in many source files.

Even worse, the sharing is not incidental if any code that might execute after the task completes, or in any other task that might run at the same time as the task, could read a value written by the task to that memory location.

There is no "magic bullet" to prove that the requirements are met, but there is a simple technique that you might find useful. Add statements that write a known bad value into the memory location immediately after the `ANNOTATE_ITERATION_TASK(taskname);`, and then test your serial program. If the sharing is incidental, these assignments will have no effect. If not, there is a good chance that the changes will change the program behavior. Of course, the effectiveness of this technique depends on how good your test system is at detecting the resulting bugs.

For example, if you want to confirm that the variable `x` is incidentally shared in `the_task()`:

```c
extern int x;
// ...
ANNOTATE_SITE_BEGIN(site1);
for (i = 0; i != n; ++i) {
    ANNOTATE_ITERATION_TASK(task1);
    x = 0xdeadbeef;
    the_task();
```
To identify stray memory references, consider using the C/C++ special-purpose annotations `ANNOTATE_OBSERVEUSES()` and `ANNOTATECLEARUSES()`.

**See Also**

Creating the Private Memory Location

Special-purpose Annotations

**Creating the Private Memory Location**

The important thing is that every execution of a task must get its own private memory location to take the place of the shared memory location in the original program. This involves:

- Creating the private memory location.
- Replacing all uses of the shared memory location with uses of the private memory location.

How you do this will depend on what kind of shared memory location you have.

**Replacing a Local Variable**

If the shared memory location is a local variable in the function containing the task’s static extent, the fix is simple:

1. Add braces around the static extent, if necessary, to make sure that it is a block.
2. Define a new variable at the beginning of the block.
3. Replace every use of the shared variable in the static extent with a use of the new variable.

Now each occurrence of the task will have its own copy of the local variable.

**Replacing a Static or Global Variable or Class Static Data Member**

Using global variables is usually a bad idea in large-scale software design. Global variables often seem like the easiest solution to a design problem, but they create obscure dependencies between parts of a program, making it harder to understand the program and harder to make changes to it. When you convert a program to run in parallel, these problems are compounded, as global variables are a prolific source of data sharing problems.

Therefore, the changes that you will make to eliminate uses of global variables are not only necessary to fix sharing problems and allow your program to run correctly in parallel. You will probably find that they would make your program more understandable and maintainable, even if you were not parallelizing it.

For example:

```c
extern int global;
// ...
ANNOTATE_SITE_BEGIN(site1);
    ANNOTATE_TASK_BEGIN(taskname);
    foo(i);
    bar(i);
    ANNOTATE_TASK_END();
// ...
ANNOTATE_SITE_END(site1);
void foo(int i)
{
    global = x*3 - a[i];
}
void bar(int i)
```

```c
$x = 0x$deadbeef;
}
ANNOTATE_SITE_BEGIN();
```
The approach to creating a private replacement for a static or global variable or a class static data member is the same as for a local variable. The important difference is that global variables may be accessed in other functions in the dynamic extent of the task, so you will have to make the private replacement variable accessible to those functions, too.

1. Define a local variable in the static extent to take the place of the shared variable, just as you do when replacing a local variable.
2. Replace all uses of the shared variable in the static extent with uses of the new local variable.
3. If the task calls other functions, add an additional reference parameter to each one, and pass the private variable to it. If you are programming in C, you will have to use a pointer parameter and pass the address of the variable to it.
4. Replace all uses of the shared variable in the called functions with uses of the reference parameter.

Applying these rules to the example above, we get:

```c
// ...
ANNOTATE_SITE_BEGIN(site1);
ANNOTATE_TASK_BEGIN(taskname);
int replacement;
foo(i, replacement);
bar(i, replacement);
ANNOTATE_TASK_END();
ANNOTATE_SITE_END(site1);
// ...
void foo(int i, int& replacement)
{
    replacement = x*3 - a[i];
}
void bar(int i, int& replacement)
{
    b[i] = b[i] - replacement;
}
```

**Mixed-caller functions**: If there are functions that are called both from the dynamic extent of the task and from outside the task, steps 3 and 4 will fix the sharing problem in the task, but they will break the calls outside the task. There are two possible solutions:

- Modify all the calls to such functions from outside the task to pass the original variable to the new parameter.
- Make two copies of the function: the original version, to be called from outside the task, and the one with the new parameter, to be called from inside the task.

**Variables whose address is taken**: The special problems of variables that are accessed through pointers are discussed in the help topic Pointer Dereferences.

**More than one shared variable**: The strategy described above is simple, but if you have a task with several incidentally shared variables, the multiple extra parameters are clumsy. A cleaner solution is to define a local structure variable with a field for each incidentally shared variable. Modify the functions and calls in the task’s dynamic extent to pass the structure to them, and replace uses of the shared variables with uses of the appropriate field of the structure.
Creating a Task Class: If the functions in the task's dynamic extent are closely related, you might be able to create a new class which has the functions as member functions and the replacement shared variables as data members. Then the class's `this` pointer takes the place of the added reference parameter. Using the same example:

```cpp
class TaskClass {
public:
    ANNOTATE_SITE_BEGIN(site1);
    void the_task()
    {
        ANNOTATE_TASK_BEGIN(taskname);
        foo(i);
        bar(i);
        ANNOTATE_TASK_END();
    }
    ANNOTATE_SITE_END(site1);
private:
    int replacement;
    void foo(int i)
    {
        replacement = x*3 - a[i];
    }
    void bar(int i)
    {
        b[i] = b[i] - replacement;
    }
};
```

Replacing a Structure Field

Sometimes you may have sharing problems with one or more fields in an object:

```cpp
struct Point { float x, y, z; }
extern Point p;
// ...
ANNOTATE_SITE_BEGIN(site1);
    ANNOTATE_TASK_BEGIN(taskname);
    p.x = a[i].x * scale_x;
    p.y = a[i].y * scale_y;
    foo(i);
    ANNOTATE_TASK_END();
ANNOTATE_SITE_END(site1);
// ...
void foo(int i)
{
    b[i].x = b[i].x - p.x;
    b[i].y = b[i].y - p.y;
}
```

The most straightforward solution is to introduce a new local variable for the shared field:

```cpp
struct Point { float x, y; }
extern Point p;
// ...
ANNOTATE_SITE_BEGIN(site1);
    ANNOTATE_TASK_BEGIN(taskname);
    float sub_p_x = a[i].x * scale_x;
    float sub_p_y = a[i].y * scale_y;
    foo(i, sub_p_x, sub_p_y);
    ANNOTATE_TASK_END();
```
If every shared field of the object is incidentally shared, then it will be simpler to make a single local replacement variable for the entire structure rather than a separate replacement variable for each shared field.

```c
struct Point { float x, y; };
extern Point p;
// ...
void foo(int i, Point& sub_p)
{
    b[i].x = b[i].x - sub_p.x;
    b[i].y = b[i].y - sub_p.y;
}
```

See Also

Pointer Dereferences

It may be tedious to find all the uses of a shared variable in a task's dynamic extent, but at least it is relatively straightforward. The situation is much worse when the Dependencies tool reports that you have a sharing problem on a pointer dereference. In general:

- A dereference of a pointer expression may or may not refer to the same object as some other dereference of a pointer expression with the same type.
- Different executions of a pointer expression dereference may or may not refer to the same object.
- If you have a variable whose address is taken, a dereference of a pointer expression may or may not refer to that variable.

However, suppose that your program has an abstract data type whose objects are implemented as dynamically allocated data structures. You may be able to step back from the individual pointer dereferences involved in a sharing problem and say: "These are just implementation details in an access to an abstract object." If you can prove that the access pattern for the abstract object satisfies the incidental sharing pattern, you can apply the techniques from this topic:

- Within the task, create a private object of the abstract data type.
- Make a reference to the private object available throughout the task.
- Replace references to the original object with references to the private object.
- Destroy the private object before the task exits.

The point is to ignore the pointer dereferences, and solve the problem in terms of the abstraction that they are implementing.

Additional suggestions for dealing with sharing problems with pointer-accessed memory locations can be found in Memory That is Accessed Through a Pointer.
Synchronizing Independent Updates
Memory That is Accessed Through a Pointer

**Synchronizing Independent Updates**
In the independent update pattern, both of the following occur:

- Two tasks contain regions of code that update the same memory locations.
- It does not matter what order the code regions execute in, as long as the regions do not execute in parallel.

For example, suppose that multiple tasks call `do_something()`:

```c
void do_something()
{
    static bool initialized = false;
    if (!initialized) {
        do_the_initialization();
        initialized = true;
    }
    do_the_real_work();
}
```

The function `do_something()` updates the variable `initialized` as well as the initialized memory locations. The function `do_the_real_work()` will never be called before the initialization happens; and the initialization will only happen once, regardless of which task calls `do_something()` first, as long as two tasks do not try to execute the `if` statement at the same time. If two tasks do try to execute the `if` statement at the same time, they could both see that `initialized` is false and both try to do the initialization.

The following sections describe several aspects of synchronizing independent updates, including explicit locking, assigning locks, and potential problems of using synchronization.

**Synchronization**
You can fix independent update sharing problems by synchronizing the execution of code that uses the same memory locations. The key idea is that when two or more tasks contain groups of operations which should not execute at the same time, there must be a lock which controls the execution of all of these groups of operations. Such a group of operations is called a *transaction*, and may be anything from a read/modify/write of a single variable to a collection of related modifications to multiple data structures.

Before beginning a transaction, a task must *acquire* the lock that controls it, and when the transaction is done, the task must *release* it. If one task has already acquired a lock, then another task that tries to acquire the same lock will stop executing until the first task has released it. This guarantees that two transactions controlled by the same lock cannot execute at the same time.

Use the Advisor lock annotations `ANNOTATE_LOCK_ACQUIRE` and `ANNOTATE_LOCK_RELEASE` to describe a transaction you intend to lock. Later, you will modify the lock annotations to actual code that implements a lock using the chosen parallel framework code:

```c
void do_something()
{
    static bool initialized = false;
    ANNOTATE_LOCK_ACQUIRE(0);
    if (!initialized) {
        do_the_initialization();
        initialized = true;
    }
    ANNOTATE_LOCK_RELEASE(0);
    do_the_real_work();
}
```

Locks are identified by a lock address.
Explicit Locking

Use **ANNOTATE_LOCK_ACQUIRE** and **ANNOTATE_LOCK_RELEASE** to specify explicit locking. These annotations are simple executable statements you can put wherever is most convenient. For example:

```c
if (synchronization_needed) ANNOTATE_LOCK_ACQUIRE(0);
x = f(x, a);
if (synchronization_needed) ANNOTATE_LOCK_RELEASE(0);
```

You must make sure you match the lock acquires and releases, and both occur in the same task. Your program will get synchronization errors if a task releases a lock that it does not own, or acquires a lock and fails to release it. You can acquire a lock in one function and release it in a different function, but it is a poor practice.

Assigning Locks to Transactions

A transaction updates a set of shared memory locations and is controlled by a lock. In general, you need to be sure that if two transactions both access the same memory location, they will not run simultaneously. What is the best way to associate locks with transactions to accomplish that? Consider:

```c
// transaction 1
if (a > b) { a -= b; b = b / 2; }
...
// transaction 2
if (c > d) { c -= d; d = d / 2; }
...
// transaction 3
if (a > c) { a -= c; c = c / 2; }
...
// transaction 4
temp = x;
x = y;
y = temp;
```

You must ensure that if two transactions can access the same memory location, they are controlled by the same lock. The simplest way to do this is to assign locks to sets of memory locations, so that if a transaction accesses two or more memory locations, all of the memory locations accessed in the transaction have the same lock. Then a transaction must be controlled by the lock that is assigned to all of the variables it accesses.

In the example above, variables `a` and `b` are both accessed in transaction 1, so they must have the same lock. Variables `c` and `d` are both accessed in transaction 2, so they must have the same lock. Variables `a` and `c` are both accessed in transaction 3, so they must have the same lock, which must be the same as the locks for `b` and `d`. Transaction 4 accesses `x` and `y`, so they must have the same lock, which is different from the lock for `a`, `b`, `c`, and `d`:

```c
int abcd_lock;
int xy_lock;
// ...
ANNOTATE_LOCK_ACQUIRE(&abcd_lock);
if (a > b) { a -= b; b = b / 2; }
```
See Also
Pitfalls from Using Synchronization

Pitfalls from Using Synchronization

Synchronization is a relatively simple way to eliminate sharing problems, but it must be used very carefully.

Performance

The purpose of synchronization is to let tasks run safely in parallel, but it does this by not letting them run in parallel when it would be unsafe. A task that is waiting for a lock is not doing any work at all.

Also, acquiring and releasing locks take a non-trivial amount of time. It is easy to write tasks that spend more time doing synchronization than doing useful work.

Taken together, these two issues mean that you need to be careful how much you use synchronization. Synchronization should be used carefully to solve specific problems. If you find yourself synchronizing large portions of your tasks, you may need to rethink your task structure so that you can get useful tasks that can run safely without so much synchronization.

One strategy is for a task to synchronize its import of the data it needs into private memory locations, work on this private data, and then synchronize the export of the results.

Synchronization Errors

The final problem with synchronization is the danger of deadlocks. A deadlock happens when one or more threads cannot make progress. This can happen, for example, when a task has acquired one lock and is trying to acquire another, while another task has acquired this second lock and is trying to acquire the first. This situation is called deadlock, and it could cause a program to hang forever.

After adding synchronization, to see whether your changes have solved the problems, run the Dependencies tool again. This may reveal previously hidden and newly introduced problems. For example, after you add locks, run the Dependencies tool again to make sure you have not accidentally introduced deadlocks into your program or unbalanced pairs of annotations. The Dependencies tool can detect potential deadlocks because in addition to memory data accesses, it observes the lock events when the annotated program runs.

See Also
Difficult Problems: Choosing a Different Set of Tasks

Difficult Problems: Choosing a Different Set of Tasks

If you find a conflict that cannot be resolved using the above techniques, you should consider the following alternatives.

- Merge the two tasks involved in the conflict into a single task.
- Divide the tasks into smaller tasks and do the work preceding the conflict in parallel, the work involving the conflict serially, and the work after the conflict in parallel.
• Find a different site to introduce parallelism.

• Intel Advisor presents a simplified model of what is possible with parallel programming. Occasionally it will be beneficial to take advantage of more advanced techniques that are available in the Intel® Threading Building Blocks (Intel® TBB), OpenMP®, or native threading APIs.

Any changes - other than lock annotations - you have made to fix incidental sharing problems should be left in the code. These changes will not harm performance, they have improved maintainability, and they may be useful at the new site or in future parallelization efforts.

See Also
Fixing Problems in Code Used by Multiple Parallel Sites

Fixing Problems in Code Used by Multiple Parallel Sites

If the Dependencies tool reports a problem(s) in one or more common functions used by multiple parallel sites, you need to investigate and consider several options. In general, keep in mind that the performance impact for privatization is usually less than synchronization, and the performance impact for synchronization is usually less than not adding parallelism. Thus, the general approach is:

• Evaluate whether it is possible to privatize the data causing the Dependencies problem for both sites. For example, you can often use privatization if the cause is incidental (accidental) sharing. Usually, providing each task with its own private copy of a variable provides the best performance.

• If you cannot fix the problem by privatizing, consider using synchronization (such as using locks or mutexes). For example, synchronization is often needed if the Dependencies problem is caused by independent updates that have true dependence.

• In some cases, it may not be feasible to add parallelism to a site. That is, after you to modify the annotations for the parallel site or its tasks and check the Suitability and Dependencies again, you might find a parallel site has negative or minimal performance gain and/or complex data sharing problems. In this case, you may need to remove the site and task annotations and add a comment that states that this code location could not be parallelized.

If you cannot eliminate a Dependencies problem for a common function called by multiple parallel sites by using the approach described above, consider adding a cloned function. That is, one of the parallel sites calls the cloned function and the other parallel site calls the original function. This allows you to implement different fixes to the Dependencies problem(s) for the original function and the cloned function. For example, this approach might allow you to privatize data in either the original function or the cloned function, which was not possible originally.

As with any program, after you modify the code within a parallel site or the annotations, you should run and analyze your program again using the Dependencies and Suitability tools.

See Also
Memory That is Accessed Through a Pointer
Data Sharing Problem Types

Memory That is Accessed Through a Pointer

In the topic Pointer Dereferences, we saw that there are techniques for dealing with incidental sharing of pointer-accessed storage in particular cases.

In general, to deal with sharing problems at indirect references you have to really understand what your program is doing. You cannot just do a text search for all the uses of a shared location and apply some transformation mechanically.

Although you may not know which memory location is being accessed by an indirect reference, you may be able to tell that a set of indirect references using the same pointer value implement an independent update pattern. The process for synchronizing independent updates of indirect references is the same as for variables. The only special concern is that you need to use the same lock for all data accesses that might be accessing the same memory locations. Using the same lock in more places means that your tasks will spend more time waiting for it.

Finally, your design may have tasks working on separate parts of a larger data structure. If you find sharing problems, it may be that the parts are not as independent as designed. In that case, you are likely to get the best results by disentangling the data structures to resolve the sharing problems.
Adding Parallelism to Your Program

Once you have completed the previous steps in the Threading Advisor workflow and have tested and approved a serial version of your application program, you can add parallelism to a selected parallel site. Before you add parallel framework code, complete developer/architect design and code reviews about the proposed parallel changes.

To add parallelism to your program, perform the following steps:

1. Choose one parallel programming framework (threading model) for your application, such as Intel Threading Building Blocks (Intel TBB), OpenMP*, Microsoft Task Parallel Library* (TPL) (on Windows* OS systems only), or some other parallel framework. To learn about the parallel framework(s) available for your application's language, see the help topic Parallel Frameworks.

2. Add the parallel framework to your build environment.

3. Add parallel framework code to synchronize access to the shared data resources, such as Intel TBB or OpenMP locks.

4. Add parallel framework code to create the parallel tasks.

In the last two steps, as you add the appropriate parallel code from the chosen parallel framework, you can keep, comment out, or replace the Intel Advisor annotations.

You should add the synchronization code - such as Intel TBB or OpenMP locks or mutexes - before adding the parallelism. Synchronized code without parallelism works correctly. In contrast, parallel code without synchronization works incorrectly.

With the synchronization in place, introduce the parallelism. This will cause the operations of multiple tasks to execute in parallel. If you have any remaining bugs caused by data sharing and synchronization problems, they will begin to appear and must be debugged.

Before You Add Parallelism: Choose a Parallel Framework

After you decide on parallel sites and tasks, select a parallel framework so you can replace Intel Advisor annotations with parallel framework code.

The available high-level parallel frameworks depend on the language whose code you will add parallelism to:

- For managed code such as C#, use the Microsoft Task Parallel Library* (TPL).
- For C/C++ native code, there are several choices as explained in Parallel Frameworks
- For Fortran native code, use OpenMP.

If you are not familiar with high-level parallel frameworks, read Parallel Frameworks.

To use a different framework, read Other Parallel Frameworks.

Parallel Frameworks

Before you can add parallel code, you must first choose a parallel framework.

There are two popular mechanisms for using threads - either use high-level parallel frameworks or explicit threading APIs. Intel recommends using parallel frameworks for both ease of use and their ability to optimize for different situations.

For managed code such as C#, use the Microsoft Task Parallel Library* (TPL).

This document shows how to use the widely-used parallel frameworks for native code, which are included with most Intel Parallel Studio XE editions and may be included with other compilers:

- Intel Threading Building Blocks
- OpenMP*

Intel Threading Building Blocks (Intel TBB) is a parallel programming framework for C++ code. Intel TBB is structured as a traditional C++ library, consisting of header files and a run-time library, so it can be used with any C++ compiler. Intel recommends that you consider using Intel TBB for introducing parallelism into C
++ programs. Intel TBB programs can be run on any platform (OS/architecture pair) to which the Intel TBB library has been ported. For example, the Intel® C++ Compiler Classic includes Intel TBB and supports Windows®, Linux®, and Apple OS X® operating systems. The open source version is available at https://github.com/intel/tbb.

OpenMP is a high-level framework that supports C, C++, and Fortran. OpenMP is provided by compiler support, so you modify your sources by using compiler directives rather than using types, variables, and calls. An OpenMP program can often be changed from parallel execution to serial execution by setting an environment variable or omitting a compiler option so the compiler ignores the directives. OpenMP 2 is supported by the Microsoft, the Intel, and the GNU® C, C++ and Fortran compilers. The OpenMP 3.0 standard adds TASK support and is supported by the Intel compilers, which also support parts of OpenMP 4.0. For Microsoft and GNU compilers, consult your compiler documentation for the current level of OpenMP support.

You can also use a different parallel framework.

Windows OS: Support for Parallel Frameworks by Microsoft and Intel Compilers

With a Fortran program, the only high-level parallel framework available is OpenMP. The following table summarizes the support by Microsoft and Intel Compilers for the recommended parallel frameworks for C/C++ programs on Windows OS systems.

<table>
<thead>
<tr>
<th>Language and Compiler</th>
<th>Intel TBB</th>
<th>OpenMP</th>
</tr>
</thead>
<tbody>
<tr>
<td>C programs, Intel® C++ Compiler Classic</td>
<td>Supported</td>
<td></td>
</tr>
<tr>
<td>C++ programs, Intel® C++ Compiler Classic</td>
<td>Supported</td>
<td>Supported</td>
</tr>
<tr>
<td>C programs, Microsoft Visual C++ Compiler</td>
<td>Supported</td>
<td></td>
</tr>
<tr>
<td>C++ programs, Microsoft Visual C++ Compiler</td>
<td>Supported</td>
<td>Supported</td>
</tr>
</tbody>
</table>

For more information about Intel TBB and OpenMP, see the corresponding sections in this Intel Advisor help system. For detailed instructions, see the Intel® Parallel Studio XE documentation and the resources listed in Related Information.

Linux OS: Support for Parallel Frameworks by GNU® and Intel Compilers

With a Fortran program, the only high-level parallel framework available is OpenMP. The following table summarizes the support by GNU gcc® and Intel compilers for the recommended parallel frameworks for C/C++ programs on Linux OS systems.

<table>
<thead>
<tr>
<th>Language and Compiler</th>
<th>Intel TBB</th>
<th>OpenMP</th>
</tr>
</thead>
<tbody>
<tr>
<td>C programs, Intel® C++ Compiler Classic (icc)</td>
<td>Supported</td>
<td></td>
</tr>
<tr>
<td>C++ programs, Intel® C++ Compiler Classic (icc)</td>
<td>Supported</td>
<td>Supported</td>
</tr>
<tr>
<td>C programs, GNU gcc Compiler (gcc)</td>
<td>Supported</td>
<td></td>
</tr>
<tr>
<td>C++ programs, GNU gcc Compiler (gxx)</td>
<td>Supported</td>
<td>Supported</td>
</tr>
</tbody>
</table>

For more information about Intel TBB and OpenMP, see the following sections in this Intel Advisor help system. For detailed instructions, see your compiler documentation and the resources listed in Related Information.

Intel® Threading Building Blocks (Intel® TBB)

Intel® Threading Building Blocks (Intel® TBB) is a high-level parallel programming framework for C++ code that uses a template-based runtime library to help you harness the performance of multi-core processors. Intel TBB lets you specify logical parallelism instead of threads. You specify potential parallelism - what can
be run in parallel. The library decides the actual parallelism at run-time, matching it to the available hardware. The library has templates that simplify using high level parallel patterns such as parallel loops.

Intel TBB programs are implemented by a library that has been ported to multiple C++ compilers.

Use Intel TBB to write scalable programs that:

- Specify parallel work instead of managing threads.
- Emphasize data parallel programming.
- Take advantage of high-level parallel patterns.

Intel TBB consists of header files and shared libraries, so it can be used with any C++ compiler.

Intel recommends that you consider using Intel TBB for introducing parallelism into C++ programs. It has a small cost of entry and provides excellent initial performance with a lot of additional capabilities that can be used for future refinements.

It also has many powerful features that can make it possible to easily parallelize more places in your application. These features include:

- Parallel algorithmic patterns
- Concurrency-friendly containers
- Scalable memory allocation
- Synchronization primitives
- Timing

**OpenMP**

OpenMP* is a parallel programming framework for C, C++, or Fortran code. Using OpenMP requires few source changes and is supported by multiple compilers. Because OpenMP is supported by OpenMP libraries, you modify your source code with compiler directives rather than using types, variables, and calls. An OpenMP program can often be changed from parallel execution to serial execution by omitting a compiler option so the compiler ignores the OpenMP directives.

OpenMP 2 is very good at using several cores on loops that process arrays, but does not support irregular parallelism through general tasking. It is supported by the Microsoft, the Intel, and the GNU* C, C++ and Fortran compilers. It is difficult to use OpenMP version 2 for situations other than simple divisions of statement sequences or complete loop bodies.

The OpenMP 3.0 specification adds **TASK** support. The **TASK** directives enable performing arbitrary pieces of an algorithm in parallel. The Intel® C++ Compiler Classic and Intel® Fortran Compiler Classic support OpenMP 3.0 and some parts of OpenMP 4.0. For Intel, Microsoft, and GNU* compilers, consult your compiler documentation for the level of OpenMP support.

If your application is written in Fortran, OpenMP is the only high-level parallel framework available.

**Microsoft Task Parallel Library** (TPL)

Microsoft Task Parallel Library* (TPL) in the Microsoft .NET* Framework is a combination of public types and APIs that allow addition of parallelism and concurrency on Windows* OS systems. For Intel Advisor users, use Microsoft TPL for C# and managed C++ libraries.

Microsoft TPL is a high-level parallel programming framework for .NET code to help you harness the performance of multi-core processors. It lets you specify logical parallelism instead of threads. That is, you specify potential parallelism – what can be run in parallel. The library decides the actual parallelism at run-time, matching it to the available hardware.

Microsoft TPL provides two main classes:

- `System.Threading.Tasks.Parallel`: includes **For** and **ForEach** loops.
- `System.Threading.Tasks.Task`: is the preferred way to express asynchronous operations.

Other classes are also available. For example, `System.Collections.Concurrent` provides for concurrent collections that do not require external locking.

You can use Microsoft TPL for introducing parallelism into either C# programs or managed C++ code.

Please refer to your Microsoft MSDN* help documentation for information about this parallel framework. For example: MSDN Library > .NET Development > .NET Framework 4 > .NET Framework Advanced Development > Parallel Programming > Task Parallel Library
Other Parallel Frameworks

Intel Advisor helps you prepare your program for adding parallelism, regardless of the parallel framework you choose. Intel Advisor provides the ability to predict the parallel behavior of your serial program and lets you determine the feasibility of possible parallel regions before you actually add parallelism.

Intel Advisor does not perform analysis of your parallel program, so you can use any parallel framework. You can use Intel Advisor with high-level parallel frameworks that use a fork-join model, or with low-level APIs that provide explicit thread control.

Intel recommends using the parallel frameworks Intel® Threading Building Blocks (Intel® TBB) or OpenMP®, which are included with most Intel® Parallel Studio XE editions. These high-level frameworks provide parallel features well-suited for most multi-core computer systems.

If you decide to use a different parallel framework or a low-level threading API, please be aware of the following considerations:

- Some parallel frameworks have limited abilities to scale with different number of cores, handle load balancing, and handle loop scalability (chunking), and so on.
- As part of your planning, you might create a mapping of at least Intel Advisor Site, Task, and Lock annotations to the equivalent code constructs in the chosen parallel framework. That is, create a list that maps the Intel Advisor annotations to your parallel framework’s features. Thus, you need to be aware of all annotations. For example, all Intel Advisor programs need to use parallel site and task annotations. Most programs will also use lock annotations. For a complete list of annotations, see the help topic Summary of Annotation Types.
- Some parallel frameworks require that you use certain compilers that recognize the parallel framework’s keywords, while others are libraries that can be used with multiple compilers.
- Some parallel frameworks may not correctly handle multi-program workloads.

In all cases, you need to learn how to use the parallel framework that you select. The current Add Parallelism workflow step involves replacing annotations with chosen parallel framework code.

Adding the Parallel Framework to Your Build Environment

After you choose the parallel framework, you need to add the parallel framework to your build environment.

Adding the parallel framework to your build environment can require installing additional software, as well as modifying build scripts, modifying project properties or Microsoft Visual Studio* project properties (on Windows* OS systems), and so on.

Later, after you add the parallel framework to your build environment, you can begin to make source code changes that use the parallel framework to add synchronization (such as locks) or parallelism to your program.

The following sections describe adding the Intel® Threading Building Blocks (Intel® TBB) and OpenMP* parallel frameworks to your build environment, including adding C++11 (formerly C++0x) Lambda Expression Support that simplifies the use of Intel TBB.

Enabling Intel® Threading Building Blocks (Intel® TBB) in your Build Environment

If you use the Intel® C++ Compiler Classic (icc on Linux* OS) from the command line, specify the following option when you build your program:

- For Windows* OS: /Qtbb
- For Linux OS: -tbb

This option tells the compiler to link with the Intel® Threading Building Blocks (Intel® TBB) libraries. If you use other compilers, please see your Intel TBB or compiler documentation.
**NOTE**
If you are using a version of Intel TBB that is not included with most Intel® Parallel Studio XE editions, please see the Intel Advisor release notes for instructions to modify several project properties and the Intel TBB environment variable. With Intel Advisor samples, to use the Intel TBB project (_tbb), you need to define the TBBROOT environment variable (see the help topic Defining the TBBROOT Environment Variable) and specify the TBBROOT/include directory as an additional include path when compiling (in build properties on Windows OS).

The following instructions are for using the Visual Studio development environment on a Windows OS system.

Modify the project properties for each of your Visual Studio* project build configurations (debug, release, and so on). You can set multiple properties by using the Configuration Properties with Visual Studio:

1. In Solution Explorer, select (click) the name of one or more projects. To select multiple projects, hold down the Ctrl key.
2. With Visual Studio:
   - Right-click the project name(s) and select Configuration Properties > Intel Performance Libraries > Intel Threading Building Blocks.
   - On the Use Intel TBB line, specify Yes.

**NOTE**
If you change the version of Intel TBB or the Visual Studio version installed on your system, you may see build errors related to Intel TBB libraries. In this case, reset the integration by repeating the above steps to uncheck, and then check, the Use TBB box. See the Intel Advisor release notes for more information.

3. Click OK to save the specified properties.
4. Repeat the steps above for other configurations.

This procedure defines multiple properties to set up your build environment to use Intel TBB.

**See Also**
Defining the TBBROOT Environment Variable
Parallel Frameworks Overview

**Defining the TBBROOT Environment Variable**

With Intel® Advisor samples, to build the Intel® Threading Building Blocks (Intel® TBB) project (_tbb), you need to define the TBBROOT environment variable.

To define this environment variable:

**On Linux* OS:**

1. Open a command line window.
2. Use the export command to set the TBBROOT environment variable, type: export TBBROOT=<tbb_install_dir>. If you used the default path during installation, the <tbb_install_dir> is inside:
   - For root users: /opt/intel/
   - For non-root users: $HOME/intel/
For example, if you installed the Threading Building Blocks as a part of Intel® oneAPI Base Toolkit, the <tbb_install_dir> may be /opt/intel/oneapi/tbb/<version>. If you installed the Threading Building Blocks as a part of Intel® Parallel Studio XE, the <tbb_install_dir> may be /opt/intel/compilers_and_libraries/linux/tbb.

3. To always set this variable on the current system, add this definition to your .login or similar shell initialization file.

On Windows® OS:

1. Open the control panel and access: Control Panel > System and Security > System > Advanced system settings > Environment Variables....
2. Locate any existing definition of the TBBROOT user or system environment variable. If present, verify that it value is correct if you encountered build errors and either click Cancel or OK as needed to exit the dialog box.
3. If it is not present, under System variables or User variables, click New.
4. Specify the Variable name as: TBBROOT.
5. Specify the Variable value as the path of the installed Intel® Parallel Studio XE or Intel® oneAPI Base Toolkit files, including the \tbb directory.
   - If you installed the product as part of a Intel® Parallel Studio XE installation and used the default path, files are installed below: C:\Program Files (x86)\IntelSWTools\ or C:\Program Files \IntelSWTools, for example: C:\Program Files (x86)\IntelSWTools \compilers_and_libraries\windows\tbb.
   - If you installed the product as part of a Intel® oneAPI Base Toolkit installation and used the default path, files are installed below: C:\Program Files (x86)\Intel\oneAPI, for example C: \Program Files (x86)\Intel\oneAPI\tbb\<version>.
6. Click OK several times.
7. For the change to take effect:
   - If using Microsoft Visual Studio*: close and reopen Visual Studio.
   - If using command window: close and reopen your command window.

   In some cases, you may need to log off and log on for this change to take effect.
8. If needed, you can test the definition by opening a command window and typing set TBBROOT.

You have defined the TBBROOT environment variable.

See Also

Enabling C++11 Lambda Expression Support with Intel® Threading Building Blocks (Intel® TBB)
Parallel Frameworks Overview
Setting and Using Intel Advisor Environment Variables

Enabling C++11 Lambda Expression Support with Intel® Threading Building Blocks (Intel® TBB)

The C++11 (new standard for the C++ language, formerly C++0x) lambda expression support makes many Intel® Threading Building Blocks (Intel® TBB) constructs easier to program because it avoids the need to introduce extra classes to encapsulate code as functions. If you decide to use this feature, you need a compiler that supports it, such as the Intel® C++ Compiler Classic. For more information about C++11 lambda expression support in the other compilers, please see your compiler documentation (online help).

When using the command line with the Intel® C++ Compiler Classic, specify the following option to enable lambda expression support:

- For Windows® OS: /Qstd=c++0x
- For Linux® OS: -std=c++0x

To enable the C++11 support in Visual Studio on a Windows OS system:

1. In Solution Explorer, select (click) the name of one or more projects. To select multiple projects, hold down the Ctrl key.
2. Select **Project > Properties**, or right-click the project name and select **Properties** from the context menu.

3. From the context menu, select **Intel Parallel Studio XE [version] > Use Intel C++**

4. Specify the following Configuration Properties:

| C++ > Language | Under Intel Specific, select **Enable C++0x Support** as Yes |

5. Click **OK** to save the specified properties.

6. Repeat the steps above for other configurations.

You have set up your environment to use the C++11 lambda expression support.

**See Also**

Adding Intel® Threading Building Blocks (Intel® TBB) to your Build Environment

Enabling OpenMP* in your Build Environment

OpenMP* is supported by certain versions of the Microsoft Visual C++* compiler, the GNU* compilers, the Intel® C++ Compiler Classic, and Intel® Fortran Compiler Classic:

- Most recent versions of the Microsoft Visual C++* compiler include OpenMP support.
- Certain editions of the Intel® Parallel Studio XE software suite includes the Intel® C++ Compiler Classic and the Intel® Fortran Compiler Classic, which supports the TASK feature introduced with OpenMP 3.0.

For information about OpenMP support for the Microsoft compilers, see your Microsoft Visual Studio help. For information about OpenMP support for the GNU compilers, see your compiler help or the appropriate man page, such as `gcc(1)`.

To enable OpenMP on the command line, specify the appropriate compiler option (see your compiler documentation), such as the `-openmp` (for Linux* OS) or `/Qopenmp` (for Windows* OS) option when using the Intel compilers.

To enable OpenMP on a **Windows OS system** using Microsoft Visual Studio*:

1. In Solution Explorer, select (click) the name of one or more projects. To select multiple projects, hold down the Ctrl key.

2. Select **Project > Properties** or right-click the project name and select **Properties** from the pop-up menu.

3. Specify the Configuration Properties for your C/C++ or Fortran project(s):

<table>
<thead>
<tr>
<th>C/C++ &gt; Language</th>
<th>Specify <strong>OpenMP Support</strong> as Yes</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

| Fortran > Language | Specify **OpenMP Support** as Yes |

4. Click **OK** to save the specified properties.

5. Repeat the steps above for other configurations.

6. You should check your startup project properties before starting a build.

You have set up your environment for OpenMP support on a Windows OS system.

**NOTE**

Even if you are only using the #pragma omp pragmas within your source, Visual C++ sources compiled with the Microsoft compilers need to #include <omp.h>. Otherwise, running the application will be missing a .dll at load time.
To include the appropriate OpenMP environment when using the Intel® Fortran Compiler Classic, specify the `use omp_lib` statement.

**See Also**
Annotation Report Overview

**Annotation Report**
Annotation Report Overview

The *Annotation Report* window displays the annotations for your program. Intel Advisor updates the listed annotations when changes occur to the specified source directories. For example, when you save a source file with a code editor.

The first three columns show the Annotation type, the source location, and the annotation label. To view or hide a source code snippet, click the icon in the *Annotation* column (as shown for the *Site* annotation). To display the source code associated with each annotation, either double-click in these columns or right-click and select *View Source* or *Edit Source*.

**See Also**
Locating Annotations with the Annotation Report
Troubleshooting No Annotations Found
Troubleshooting Sources Not Available
Troubleshooting Debug Information Not Available

**Locating Annotations with the Annotation Report**

The *Annotation Report* window lists all the Intel Advisor annotations found in your project and their types. Each annotation appears as a separate row in a table-like grid.

To use the list of annotations in the *Annotation Report* window to find annotations as you replace annotations with parallel framework code:

1. To display the *Annotation Report* window, click the *Annotation Report* tab or - if you are using the *Advisor Workflow* tab - click the (View Annotations) button below 2. *Annotate Sources* or 5. *Add Parallel Framework*. The annotations associated with the selected start-up project appear. If you have run the Suitability and Dependencies tools for this start-up project, the most recent relevant data also appears in their respective columns.

2. To sort the annotations by type, click the column heading Annotations. The suggested way to replace annotations is to replace lock annotations first, and then site and task annotations (this is because synchronized code without parallelism works correctly, but parallel code without synchronization works incorrectly). To show or hide a code snippet showing an annotation, click the icon next to its name in the Annotations column.
3. To open the code editor with the corresponding source file, double-click an annotation type (data row) in the Annotations column or a line in its code snippet (or use the Edit Source context menu item). When using the Intel Advisor GUI on Linux* OS, the editor defined by the Options > Editor dialog box appears with the file open at the corresponding location. When using the Intel Advisor GUI on Windows* OS, the file type association (or Open With dialog box) determines the editor used. When using Visual Studio*, the Visual Studio code editor appears with the file open at the corresponding location.

4. Read the documentation associated with the parallel framework as well as the relevant information in Intel Advisor help so you understand what parallel framework code to insert. In many cases, you need to insert parallel framework declarations at the start of the source file, as well as parallel framework code that replaces the annotations.

5. Repeat the steps above for each lock annotation.

6. Repeat the steps above for each site and task annotation.

You have used the Annotation Report window to help you locate and replace the Intel Advisor annotations with parallel framework code.

**Replacing Annotations with Intel® Threading Building Blocks (Intel® TBB) Code**

Annotations and Intel® Threading Building Blocks (Intel® TBB) Code

This topic explains the steps needed to implement parallelism proposed by the Intel Advisor annotations by adding Intel® Threading Building Blocks (Intel® TBB) parallel framework code.

- Add Intel TBB code to add appropriate synchronization of shared resources, using the LOCK annotations as a guide. The following topics cover the Intel TBB synchronization options:
  - Intel® Threading Building Blocks (Intel® TBB) Mutexes
  - Intel® Threading Building Blocks (Intel® TBB) Mutex - Example
- Add code to create Intel TBB tasks, using the SITE/TASK annotations as a guide. The following topics cover the Intel TBB task creation options:
  - Parallelize Functions - Intel® Threading Building Blocks (Intel® TBB) Tasks
  - Parallelize Data - Intel® Threading Building Blocks (Intel® TBB) Counted Loops
  - Parallelize Data - Intel® Threading Building Blocks (Intel® TBB) Loops with Complex Iteration Control

This is the recommended order of tasks for replacing the annotations with Intel TBB code:

1. Add appropriate synchronization of shared resources, using LOCK annotations as a guide.
2. Test to verify you did not break anything, before adding the possibility of non-deterministic behavior with parallel tasks.
3. Add code to create Intel TBB tasks or loops, using the SITE/TASK annotations as a guide.
4. Test with one thread, to verify that your program still works correctly.
5. Test with more than one thread to see that the multithreading works as expected.

The Intel TBB parallel framework creates worker threads automatically. In general, you should concern yourself only with the tasks, and leave it to the framework to create and destroy the worker threads.

If you do need some control over creation and destruction of worker threads, read about task_scheduler_init in the Intel TBB Reference manual.

The table below shows the serial, annotated program code in the left column and the equivalent Intel TBB parallel code in the right column for some typical code to which parallelism can be applied.

<table>
<thead>
<tr>
<th>Serial Code with Intel Advisor Annotations</th>
<th>Parallel Code using Intel TBB</th>
</tr>
</thead>
<tbody>
<tr>
<td>// Locking</td>
<td>// Locking can use various mutex types provided</td>
</tr>
<tr>
<td>ANNOTATE_LOCK_ACQUIRE();</td>
<td>// by Intel TBB. For example:</td>
</tr>
<tr>
<td>Body();</td>
<td>#include &lt;tbb/tbb.h&gt;</td>
</tr>
<tr>
<td>ANNOTATE_LOCK_RELEASE();</td>
<td>...</td>
</tr>
<tr>
<td></td>
<td>tbb::mutex g_Mutex;</td>
</tr>
<tr>
<td></td>
<td>(</td>
</tr>
</tbody>
</table>
Serial Code with Intel Advisor Annotations

```
// Do-All Counted loops, one task
ANNOTATE_SITE_BEGIN(site);
    For (I = 0; I < N; ++) {
        ANNOTATE_ITERATION_TASK(task);
        {statement;}
    }
ANNOTATE_SITE_END();
```

Parallel Code using Intel TBB

```
tbb::mutex::scoped_lock lock(g_Mutex);
Body();
```

```
// Do-All Counted loops, using lambda
// expressions
#include <tbb/tbb.h>
...
tbb::parallel_for(0,N,[&](int I) {
    statement;
});
```

```
// Create Multiple Tasks
ANNOTATE_SITE_BEGIN(site);
    ANNOTATE_TASK_BEGIN(task1);
    statement-or-task1;
    ANNOTATE_TASK_END();
    ANNOTATE_TASK_BEGIN(task2);
    statement-or-task2;
    ANNOTATE_TASK_END();
ANNOTATE_SITE_END();
```

```
// Create Multiple tasks, using lambda
// expressions
#include <tbb/tbb.h>
...
tbb::parallel_invoke(
    [&]{statement-or-task1;},
    [&]{statement-or-task2;}
);
```

For information about common parallel programming patterns and how to implement them in Intel TBB, see the Intel TBB help topic Design Patterns.

**Intel® Threading Building Blocks (Intel® TBB) Mutexes**

With Intel® Threading Building Blocks (Intel® TBB), you can associate a mutex with a shared object to enforce mutually exclusive access to that object. A mutex is either locked or unlocked. For a thread to safely access the object:

- The thread **acquires a lock** on the mutex.
- The thread accesses the associated shared object.
- The thread **releases its lock** on the mutex.

When a mutex is locked, if another thread tries to also acquire a lock on it, this second thread is stalled until the first thread releases its lock on the mutex. This functionality provided by a mutex is exactly the semantic function intended by the Intel Advisor annotations `ANNOTATE_LOCK_ACQUIRE()` and `ANNOTATE_LOCK_RELEASE()`.

With Intel TBB, the annotation lock address becomes the mutex object. The `ANNOTATE_LOCK_ACQUIRE()` and `ANNOTATE_LOCK_RELEASE()` annotations become operations on this mutex.

Intel TBB provides several classes for locking, each with different properties. For more information, refer to the Intel TBB documentation. If you are not sure what type of a mutex is most appropriate, consider using `tbb::mutex` as your initial choice.

**See Also**

Intel® Threading Building Blocks (Intel® TBB) Simple Mutex - Example

Intel® Threading Building Blocks (Intel® TBB) Simple Mutex - Example

The following examples shows basic usage of a Intel® Threading Building Blocks (Intel® TBB) mutex to protect a shared variable named `count` using simple mutexes and scoped locks:
**Simple Mutex Example**

```c
#include <tbb/mutex.h>

int count;
tbb::mutex countMutex;

int IncrementCount() {
    int result;
    // Add Intel TBB mutex
    countMutex.lock(); // Implements ANNOTATE_LOCK_ACQUIRE()
    result = count++;
    // Save result until after unlock
    countMutex.unlock(); // Implements ANNOTATE_LOCK_RELEASE()
    return result;
}
```

The semantics of `countMutex.lock()` and `unlock()` on `countMutex` correspond directly to the annotations `ANNOTATE_LOCK_ACQUIRE()` and `ANNOTATE_LOCK_RELEASE()`. However, it is generally better to use the *scoped locking* pattern.

**Scoped Lock Example**

With a scoped lock, you construct a temporary *scoped_lock* object that represents acquisition of a lock. Destruction of the *scoped_lock* object releases the lock on the mutex.

The following code shows the previous example rewritten using scoped locking:

```c
#include <tbb/mutex.h>

int count;
tbb::mutex countMutex;

int IncrementCount() {
    int result;
    {
        // Add Intel TBB scoped lock at location of ANNOTATE_LOCK annotations
        tbb::mutex::scoped_lock lock(countMutex); // Implements ANNOTATE_LOCK_ACQUIRE()
        result = count++;
        // Implicit ANNOTATE_LOCK_RELEASE() when leaving the scope below.
    } // scoped lock is automatically released here
    return result;
}
```

The *scoped_lock* pattern is preferred because it releases the lock no matter how control leaves the block. The *scoped_lock* is released when destruction of the *scoped_lock* object occurs. In particular, it releases the lock even when control leaves because an exception was thrown.

Intel TBB also has a `tbb::atomic` template class that can be used in simple cases such as managing a shared integer variable. Check the Related Information for details.

**See Also**

Testing the Intel® Threading Building Blocks (Intel® TBB) Synchronization Code

**Related Information**

**Testing the Intel® Threading Building Blocks (Intel® TBB) Synchronization Code**

After you add Intel® Threading Building Blocks (Intel® TBB) synchronization code (such as mutexes), but before adding the constructs that cause the program to use parallel execution, you should test your serial program. The synchronization code may introduce problems if you have inadvertently used a non-recursive mutex in a recursive context, or if your edits accidentally changed some other piece of program behavior.
It is much easier to find these problems in the serial version of your program than it will be in the parallel version.

**See Also**
Parallelize Functions - Intel® Threading Building Blocks (Intel® TBB) Tasks

Parallelize Functions - Intel® Threading Building Blocks (Intel® TBB) Tasks
The following sections describe various alternatives, depending on how the tasks fit within the surrounding parallel site.

**Two or More Parallel Statements**
When the outermost statements in the annotation site have been placed into tasks, as shown in this serial example, it is easy to execute them in parallel.

```c
ANNOTATE_SITE_BEGIN(sitename);
  ANNOTATE_TASK_BEGIN(task1);
    statement_1
  ANNOTATE_TASK_END();
  ANNOTATE_TASK_BEGIN(task2);
    statement_2
  ANNOTATE_TASK_END();
ANNOTATE_SITE_END();
```

Two or More Parallel Statements - Intel® Threading Building Blocks (Intel® TBB)
The easiest way to cause several sequential statements to be executed as independent tasks is to change your program as follows using `parallel_invoke`.

Both of the following examples use the C++11 lambda expression feature - you need to use the Intel® C++ Compiler Classic and enable the C++11 support to compile it.

```c
#include <tbb/tbb.h>
...
  tbb::parallel_invoke(
    [&]{statement_1;},
    [&]{statement_2;}
  )
```

A variable used inside a lambda expression but declared outside it is said to be captured. The `[]` in the example specifies capture by reference. It is also possible to capture by value `[]=`, or even capture different variables different ways. See the compiler documentation on lambda expressions for details.

Using C++ structs Instead of Lambda Expressions
Any code that can be written with a lambda expression can be written without one - it is just more work. All a lambda expression does is:

1. Define a class with operator() defined to execute the body of the lambda expression.
2. Define a class constructor that captures variables into fields of the class.
3. Construct an instance of that class.

The constructor can capture any of the surrounding locals that are needed and save them in data members.

```c
{ struct S1 { void operator()() { statement_1; };
    struct S2 { void operator()() { statement_2; };
    tbb::parallel_invoke(S1(), S2());
  };
```
See Also
Parallelize Data - Intel® Threading Building Blocks (Intel® TBB) Counted Loops

Parallelize Data - Intel® Threading Building Blocks (Intel® TBB) Counted Loops

When tasks are loop iterations, and the iterations are over a range of values that are known before the loop starts, the loop is easily expressed in Intel® Threading Building Blocks (Intel® TBB).

Consider the following serial code and the need to add parallelism to this loop:

```c
ANNOTATE_SITE_BEGIN(sitename);
for (int i = lo; i < hi; ++i) {
    ANNOTATE ITERATION TASK(taskname);
    statement;
}
ANNOTATE_SITE_END();
```

Here is the serial example converted to use Intel® Threading Building Blocks (Intel® TBB), after you remove the Intel Advisor annotations:

```c
#include <tbb/tbb.h>
...
  tbb::parallel_for( lo, hi,
    [&] (int i) {statement;}
  );
```

The first two parameters are the loop bounds. As is typical in C++ (especially STL) programming, the lower bound is inclusive and the upper bound is exclusive. The third parameter is the loop body, wrapped in a lambda expression. The loop body will be called in parallel by threads created by Intel TBB. As described before in Create the Tasks, Using C++ structs Instead of Lambda Expressions, the lambda expressions can be replaced with instances of explicitly defined class objects.

See Also
Parallelize Data - Intel® Threading Building Blocks (Intel® TBB) Loops with Complex Iteration Control

Parallelize Functions - Intel® Threading Building Blocks (Intel® TBB) Tasks for information on using C++ structs instead of lambda functions

Parallelize Data - Intel® Threading Building Blocks (Intel® TBB) Loops with Complex Iteration Control

Sometimes the loop control is spread across complex control flow. Using Intel® Threading Building Blocks (Intel® TBB) in this situation requires more features than the simple loops. Note that the task body must not access any of the auto variables defined within the annotation site, because they may have been destroyed before or while the task is running. Consider this serial code:

```c
extern char a[];
int previousEnd = -1;
ANNOTATE SITE BEGIN(sitename);
for (int i=0; i<=100; i++) {
    if (!a[i] || i==100) {
        ANNOTATE TASK BEGIN(do_something);
        DoSomething(previousEnd+1,i);
        ANNOTATE TASK END();
        previousEnd=i;
    }
}
ANNOTATE SITE END();
```

In general, counted loops have better scalability than loops with complex iteration control, because the complex control is inherently sequential. Consider reformulating your code as a counted loop if possible.
The prior example is easily converted to parallelism by using the `task_group` feature of Intel TBB:

```cpp
#include <tbb/tbb.h>
...
extern char a[];
int previousEnd = -1;
task_group g;
for (int i=0; i<=100; i++) {
    if (!a[i] || i==100) {
        g.run([=]{DoSomething(previousEnd+1,i);}
             previousEnd=i;
    }
} 
g.wait(); // Wait until all tasks in the group finish
```

Here the lambda expression uses capture by value ` [=]` because it is important for it to grab the values of `i` and `previousEnd` when the expression constructs its functor, because afterwards the value of `previousEnd` and `i` change.

For more information on `tbb::task_group`, see the Intel TBB documentation.

**See Also**

Using Intel® Inspector and Intel® VTune™ Profiler

**Replacing Annotations with OpenMP® Code**

**Annotations and OpenMP® Code**

This topic explains the steps needed to implement parallelism proposed by the Intel Advisor annotations by adding OpenMP® parallel framework code.

- Add OpenMP code to provide appropriate synchronization of shared resources, using the LOCK annotations as a guide.
- Add code to create OpenMP tasks, using the SITE/TASK annotations as a guide.

The recommended order for replacing the annotations with OpenMP code:

1. Add appropriate synchronization of shared resources, using LOCK annotations as a guide.
2. Test to verify you did not break anything, before adding the possibility of non-deterministic behavior with parallel tasks.
3. Add code to create OpenMP parallel sections or equivalent, using the SITE/TASK annotations as a guide.
4. Test with one thread to verify that your program still works correctly. For example, set the environment variable `OMP_NUM_THREADS` to 1 before you run your program.
5. Test with more than one thread to see that the multithreading works as expected.

OpenMP creates worker threads automatically. In general, you should concern yourself only with the tasks, and leave it to the parallel frameworks to create and destroy the worker threads.

If you do need some control over creation and destruction of worker threads, see the compiler documentation. For example, to limit the number of threads, set the `OMP_THREAD_LIMIT` or the `OMP_NUM_THREADS` environment variable.

The table below shows the serial, annotated program code in the left column and the equivalent OpenMP C/C++ and Fortran parallel code in the right column for some typical code to which parallelism can be applied.

<table>
<thead>
<tr>
<th>Serial C/C++ and Fortran Code with Intel Advisor Annotations</th>
<th>Parallel C/C++ and Fortran Code using OpenMP</th>
</tr>
</thead>
</table>
| // Synchronization, C/C++
  ANNOTATE_LOCK_ACQUIRE(0);
  Body();
  ANNOTATE_LOCK_RELEASE(0); | // Synchronization can use OpenMP
  // critical sections, atomic operations, locks,
  // and reduction operations (shown later) |
### Serial C/C++ and Fortran Code with Intel Advisor Annotations

```c
! Synchronization, Fortran
call annotate_lock_acquire(0)
  body
  call annotate_lock_release(0)
```

```fortran
! Synchronization can use OpenMP
! critical sections, atomic operations, locks,
! and reduction operations (shown later)
```

### Parallel C/C++ and Fortran Code using OpenMP

```c
// Synchronization can use OpenMP
// critical sections, atomic operations, locks,
// and reduction operations (shown later)
```

```c
// Parallelize data - one task within a
// C/C++ counted loop
ANNOTATE_SITE_BEGIN(site);
for (i = lo; i < n; ++i) {
  ANNOTATE_ITERATION_TASK(task);
  statement;
}
ANNOTATE_SITE_END();
```

```c
// Parallelize data - one task, C/C++ counted loops
#pragma omp parallel for
for (int i = lo; i < n; ++i) {
  statement;
}
```

```c
! Parallelize data - one task within a
! Fortran counted loop
call annotate_site_begin("site1")
do i = 1, N
call annotate_iteration_task("task1")
  statement
end do
call annotate_site_end
```

```c
! Parallelize data - one task with a!
! Fortran counted loop
!$omp parallel do
do i = 1, N
  statement
end do
!$omp end parallel do
```

```c
// Parallelize C/C++ functions
ANNOTATE_SITE_BEGIN(site);
  ANNOTATE_TASK_BEGIN(task1);
    function_1();
  ANNOTATE_TASK_END();
  ANNOTATE_TASK_BEGIN(task2);
    function_2();
  ANNOTATE_TASK_END();
ANNOTATE_SITE_END();
```

```c
// Parallelize C/C++ functions
#pragma omp parallel //start parallel region
{
  #pragma omp sections
    {
      #pragma omp section
        function_1();
      #pragma omp section
        function_2();
    }
} // end parallel region
```

```c
! Parallelize Fortran functions
call annotate_site_begin("site1")
call annotate_task_begin("task1")
  call subroutine_1
call annotate_task_end
call annotate_task_begin("task2")
  call subroutine_2
call annotate_task_end
call annotate_site_end
```

```c
! Parallelize Fortran functions
!$omp parallel ! start parallel region
!$omp sections
  !$omp section
    call subroutine_1
  !$omp section
    call subroutine_2
!$omp end sections
!$omp end parallel ! end parallel region
```

---

### Adding OpenMP Code to Synchronize the Shared Resources

OpenMP provides several forms of synchronization:

- A **critical section** prevents multiple threads from accessing the critical section's code at the same time, thus only one active thread can update the data referenced by the code. A critical section may consist of one or more statements. To implement a critical section:

  - With C/C++: `#pragma omp critical`
  - With Fortran: `!$omp critical and !$omp end critical`
Use the optional named form for a non-nested mutex, such as (C/C++) 
#pragma omp critical(name) 
or (Fortran) !$omp critical(name) and !$omp end critical(name). If the optional (name) is omitted, it locks a single unnamed global mutex. The easiest approach is to use the unnamed form unless performance measurement shows this shared mutex is causing unacceptable delays.

- **An atomic operation** allows multiple threads to safely update a shared numeric variable on hardware platforms that support its use. An atomic operation applies to only one assignment statement that immediately follows it. To implement an atomic operation:
  - With C/C++: insert a #pragma omp atomic before the statement to be protected.
  - With Fortran: insert a !$omp atomic before the statement to be protected.

The statement to be protected must meet certain criteria (see your compiler or OpenMP documentation).

- **Locks** provide a low-level means of general-purpose locking. To implement a lock, use the OpenMP types, variables, and functions to provide more flexible and powerful use of locks. For example, use the omp_lock_t type in C/C++ or the type=omp_lock_kind in Fortran. These types and functions are easy to use and usually directly replace Intel Advisor lock annotations.

- **Reduction operations** can be used for simple cases, such as incrementing a shared numeric variable or summing an array into a shared numeric variable. To implement a reduction operation, add the reduction clause within a parallel region to instruct the compiler to perform the summation operation in parallel using the specified operation and variable.

- OpenMP provides other synchronization techniques, including specifying a barrier construct where threads will wait for each other, an ordered construct that ensures sequential execution of a structured block within a parallel loop, and master regions that can only be executed by the master thread. For more information, see your compiler or OpenMP documentation.

The following topics briefly describe these forms of synchronization. Check your compiler documentation for details.

**See Also**

Testing the OpenMP Synchronization Code

**OpenMP Critical Sections**

Use OpenMP critical sections to prevent multiple threads from accessing the critical section’s code at the same time, thus only one active thread can update the data referenced by the code. Critical sections are useful for a non-nested mutex.

Unlike OpenMP atomic operations that provide fine-grain synchronization for a single operation, critical sections can provide course-grain synchronization for multiple operations.

Use:

- #pragma omp critical with C/C++.
- !$omp critical and !$omp end critical with Fortran.

If the optional (name) is omitted, it locks an unnamed global mutex. The easiest approach is to use the unnamed form unless this shared mutex is causing unacceptable performance delays.

For example, consider this annotated C/C++ serial program:

```c
int count;
void Tick() {
    ANNOTATE_LOCK_ACQUIRE(0);
    count++;
    ANNOTATE_LOCK_RELEASE(0);
}
```

The parallel C/C++ code after adding #include <omp.h> and #pragma omp critical:

```c
#include <omp.h>  //prevents a load-time problem with a .dll not being found
int count;
void Tick() {
```

504
Consider this annotated Fortran serial code:

```fortran
program ABC
  integer(kind=4) :: count = 0
  ...
contains
  subroutine Tick
    call annotate_lock_acquire(0)
    count = count + 1
    call annotate_lock_release(0)
  end subroutine Tick
  ...
end program ABC
```

The parallel Fortran code after adding `use omp_lib`, `!$omp critical`, and `!$omp end critical`:

```fortran
program ABC
  use omp_lib
  integer(kind=4) :: count = 0
  ...
contains
  subroutine Tick
    !$omp critical
    count = count + 1
    !$omp end critical
  end subroutine Tick
  ...
end program ABC
```

**See Also**

Testing the OpenMP Synchronization Code

Related Information

**Basic OpenMP Atomic Operations**

Use OpenMP atomic operations to allow multiple threads to safely update a shared numeric variable, such as on hardware platforms that support atomic operation use. An atomic operation applies only to the single assignment statement that immediately follows it, so atomic operations are useful for code that requires fine-grain synchronization.

Before the statement to be protected, use:

- `#pragma omp atomic` for C/C++.
- `!$omp atomic` for Fortran.

For example, consider this annotated C/C++ serial code:

```c
int count;
void Tick() {
    ANNOTATE_LOCK_ACQUIRE(0);
    count = count+1;
}
The parallel C/C++ code after adding `#include <omp.h>` and `#pragma omp atomic`:

```c
#include <omp.h>  //prevents a load-time problem with a .dll not being found
int count;
void Tick() {
  // Replace lock annotations
  #pragma omp atomic
  count = count+1;
}
```

The parallel Fortran code after adding `use omp_lib` and the `!$omp atomic` directive:

```fortran
program ABC
  use omp_lib
  integer(kind=4) :: count = 0
  ...
  contains
  subroutine Tick
    call annotate_lock_acquire(0)
    count = count + 1
    call annotate_lock_release(0)
  end subroutine Tick
  ...
end program ABC
```

Consider this annotated Fortran serial code:

```fortran
program ABC
  integer(kind=4) :: count = 0
  ...
  contains
  subroutine Tick
    call annotate_lock_acquire(0)
    count = count + 1
    call annotate_lock_release(0)
  end subroutine Tick
  ...
end program ABC
```

The Intel Advisor Fortran sample `nqueens.f90` demonstrates the use of an atomic operation.

This topic introduces basic OpenMP atomic operations. For advanced atomic operations that use clauses after the `atomic` construct, see Advanced OpenMP Atomic Operations.

See Also

Advanced OpenMP Atomic Operations
Testing the OpenMP Synchronization Code
Related Information

Advanced OpenMP Atomic Operations

This topic provides advanced examples of OpenMP* atomic operations.

These advanced atomic operations use clauses after the `atomic` construct, such as `read`, `write`, `update`, `capture`, and `seq_cst`. If you do not add a clause after `atomic`, the default is `update`. 
Because these clauses are part of OpenMP 3.1 and 4.0 specification, you need a compiler that supports these advanced atomic clauses, such as the Intel® C++ Compiler Classic or the Intel® Fortran Compiler Classic.

**Example Using the read and write Clauses**
The following C/C++ example uses separate read and write clauses:

```c
int atomic_read(const int *x)
{
    int value;
    /* Ensure that the entire value of *x is read atomically. */
    /* No part of *x can change during the read operation. */
    #pragma omp atomic read
    value = *x;
    return value;
}

void atomic_write(int *x, int value)
{
    /* Ensure that value is stored atomically into *x. */
    /* No part of *x can change until after the entire write operation has completed. */
    #pragma omp atomic write
    *x = value;
}
```

The following Fortran example uses the read and write clauses:

```fortran
function atomic_read(x)
    integer :: atomic_read
    integer, intent(in) :: x
    ! Ensure that the entire value of x is read atomically. No part of x can change during
    ! the read operation.
    !$omp atomic read
    atomic_read = x
    return
end function atomic_read

subroutine atomic_write(x, value)
    integer, intent(out) :: x
    integer, intent(in) :: value
    ! Ensure that value is stored atomically into x. No part of x can change
    ! until after the entire write operation has completed.
    !$omp atomic write
    x = value
end subroutine atomic_write
```

**Example Using the Basic capture Clause**
The following C/C++ example uses the capture clause:

```c
#pragma omp parallel for shared (pos)
for (int i=0; i < size; i++)
{
    if (isValid(data[i]))
    {
        int tmpPos;
        // Using omp atomic capture pragma
        #pragma omp atomic capture
        {
            tmpPos = pos;
            pos = pos+1;
        }
    }
```

Intel® Advisor User Guide
Example Using the Swap Form of the capture Clause

The capture clause example above might be modified to use the following code snippet:

```c
// with introduction of "atomic swap" you can also use forms like:
newPos = foo();
.
.
#pragma omp atomic capture
{
    tmpPos = pos;
    pos = newPos;
}
```

See Also

Testing the OpenMP Synchronization Code
Basic OpenMP Atomic Operations
Related Information

OpenMP Reduction Operations

OpenMP reduction operations can be used for simple cases, such as incrementing a shared numeric variable or the summation of an array into a shared numeric variable. To implement a reduction operation, add the reduction clause within a parallel region to instruct the compiler to perform the summation operation in parallel using the specified operation and variable.

Consider this annotated C/C++ serial code:

```c
int i, n=500000;
float *array, total=0.0;
...
for (i=0; i <n ; ++i
{
    ANNOTATE_LOCK_ACQUIRE(0);
    total+ = array[i];
    ANNOTATE_LOCK_RELEASE(0);
}
...
```

The parallel C/C++ code after adding `#include <omp.h>` and `#pragma omp parallel for reduction:`

```c
#include <omp.h> // prevents a load-time problem with a .dll not being found
int i, n=500000;
float *array, total=0.0;
...
#pragma omp parallel for reduction(+:total)
for (i=0; i <n ; ++i
{
    total+ = array[i];
}
...
```
Consider this annotated Fortran serial code:

```fortran
integer(4) n
real(4) array(50000), total = 0.0
n = 500000
...
  do i=1, n
    call annotate_lock_acquire(0)
    total = total + array(i)
    call annotate_lock_release(0)
  ...
end do
```

Consider this parallel Fortran code after adding `use omp_lib, !$omp parallel do reduction(+:total), and !$omp end parallel do`:

```fortran
use omp_lib
integer(4) n
real(4) array(50000), total = 0.0
n = 500000
...
$omp parallel do reduction(+:total)
  do i=1, n
    total = total + array(i)
$omp end parallel do
...
end do
```

**See Also**

Testing the OpenMP Synchronization Code

Related Information

**OpenMP Locks**

Consider the following annotated C/C++ serial code:

```c
int count;
void Tick() {
  ANNOTATE_LOCK_ACQUIRE(0);
  count++;
  ANNOTATE_LOCK_RELEASE(0);
}
```

To implement a lock, use the OpenMP types, variables, and functions to provide more flexible and powerful use of locks. For example, for simple locks, use the `omp_lock_t` type in C/C++ or the `type=omp_lock_kind` in Fortran.

Locks can be wrapped inside C++ classes, as shown in the following parallel C/C++ code:

```c
#include <omp.h>
int count;
omp_lock_t countMutex;

struct CountMutexInit {
  CountMutexInit() { omp_init_nest_lock   (&countMutex);   }
  ~CountMutexInit() { omp_destroy_nest_lock(&countMutex); }
} countMutexInit;
```
// The declaration of the above object causes countMutex to
// be initialized on program startup, and to be destroyed when
// the program completes, via the constructor and destructor.

struct CountMutexHold {
    CountMutexHold() { omp_set_nest_lock (&countMutex); }
    ~CountMutexHold() { omp_unset_nest_lock (&countMutex); }
};

void Tick() {
    // unlocks on scope exit
    CountMutexHold releaseAtEndOfScope;
    count++;
    ...
}

Consider the following annotated Fortran serial code:

program BBB
    integer(kind=4) :: count = 0
    ... contains
    subroutine Tick
        call annotate_lock_acquire(0)
        count = count + 1
        call annotate_lock_release(0)
    end subroutine Tick
    ... subroutine Tick
end program BBB

For simple locks with Fortran code, use the type=omp_lock_kind. The parallel Fortran code follows after adding use omp_lib and the integer declaration for count:

program BBB
    use omp_lib
    integer(kind=4) :: count = 0
    integer(kind=omp_lock_kind) countMutex
    call omp_nest_lock_init(countMutex)
    ... contains
    subroutine Tick
        call omp_set_nest_lock(countMutex)
        count = count + 1
        call omp_unset_nest_lock(countMutex)
    end subroutine Tick
    ... subroutine Tick
end program BBB

See Also
Testing the OpenMP Synchronization Code
Related Information
Testing the OpenMP Synchronization Code

After you have added OpenMP synchronization code (such as locks, critical sections, or atomic operations), but before adding the constructs that cause the program to use parallel execution, you should test your serial program. The synchronization code may introduce problems if you have inadvertently used a non-recursive mutex in a recursive context, or if your edits accidentally changed some other piece of program behavior. It is much easier to find these problems in the serial version of your program than it will be in the parallel version.

See Also
Parallelize Functions - OpenMP Tasks

Parallelize Functions - OpenMP Tasks

You can enable multiple function calls to run in parallel as two or more tasks. This is useful for functions in library code for which the source is not available. The statements to run in parallel are not limited to function calls (see the help topic Data and Task Parallelism).

When the outermost statements in the annotation site have been placed into tasks, as shown in the following serial example, it is easy to execute them in parallel.

Consider the C/C++ annotated code:

```c
ANNOTATE_SITE_BEGIN(sitename);
  ANNOTATE_TASK_BEGIN(task1);
    statement-1;
  ANNOTATE_TASK_END();
  ANNOTATE_TASK_BEGIN(task2);
    statement-2;
  ANNOTATE_TASK_END();
  ANNOTATE_TASK_BEGIN(task3);
    statement-3;
  ANNOTATE_TASK_END();
ANNOTATE_SITE_END();
```

For the C/C++ parallel code, OpenMP provides explicit support using `#pragma omp parallel sections` and related pragmas within a parallel code region:

```c
#pragma omp parallel sections
{
  #pragma omp section
  {
    statement-1;
  } #pragma omp section
  {
    statement-2;
  }
  ... #pragma omp section

Consider the annotated Fortran code:

```fortran
call annotate_site_begin("sitename")
call annotate_task_begin("task_1")
call subroutine_1
call annotate_task_end
call annotate_task_begin("task_2")
call subroutine_2
call annotate_task_end
call annotate_site_end()
```
For the parallelized Fortran code, OpenMP provides the !$omp sections and related directives that can often replace the corresponding annotations within a parallel code region:

```fortran
!$omp parallel
!$omp sections
!$omp section
call subroutine_1
!$omp section
call subroutine_2
!$omp end sections
!$omp end parallel
```

---

**See Also**

Parallelize Data - OpenMP Counted Loops

Data and Task Parallelism

**Parallelize Data - OpenMP Counted Loops**

When tasks are loop iterations, and the iterations are over a range of values that are known before the loop starts, the loop is easily expressed in OpenMP.

Consider the following annotated serial C/C++ loop:

```c
ANNOTATE_SITE_BEGIN(sitename);
for (int i = lo; i < hi; ++i) {
    ANNOTATE_ITERATION_TASK(taskname);
    statement;
}
ANNOTATE_SITE_END();
```

OpenMP makes it easy to introduce parallelism into loops. With C or C++ programs, add the `omp parallel for` pragma immediately before the C/C++ for statement:

```c
... #pragma omp parallel for
    for (int i = lo; i < hi; ++i) {
        statement;
    }
```

Consider the following annotated Fortran serial loop:

```fortran
call annotate_site_begin("sitename")
    do i = 1, N
        call annotate_iteration_task("taskname")
        statement
    end do
call annotate_site_end
```

With Fortran programs, add the !$omp parallel do directive immediately before the Fortran do statement:

```fortran
... !$omp parallel do
    do i = 1, N
```
The OpenMP compiler support encapsulates the parallel construct. The rules for capturing the locals can be defaulted or specified as part of the pragma or directive. The loop control variable defaults to being private so each iteration sees its allotted value.

**See Also**

Parallelize Data - OpenMP Loops with Complex Iteration Control
Parallelize Functions - OpenMP Tasks

Parallelize Data - OpenMP Loops with Complex Iteration Control

Sometimes the loop control is spread across complex control flow. Using OpenMP in this situation requires more features than the simple loops. The task body must not access any of the auto variables defined within the annotation site, because they may have been destroyed before or while the task is running. Also, note that variables referenced within the task construct default to firstprivate.

Consider this serial C/C++ code:

```c
extern char a[];
int previousEnd = -1;
ANNOTATE_SITE_BEGIN(sitename);
for (int i=0; i<=100; i++) {
  if (!a[i] || i==100) {
    ANNOTATE_TASK_BEGIN(do_something);
    DoSomething(previousEnd+1,i);
    ANNOTATE_TASK_END();
    previousEnd=i;
  }
}
ANNOTATE_SITE_END();
```

This is done using the OpenMP task pragma. Without using this feature, such loops are extremely difficult to parallelize. One approach to the adding parallelism to the loop is to simply spawn each call to DoSomething():

```c
extern char a[];
int previousEnd = -1;
#pragma omp parallel
{
  #pragma omp single
  {
    ...  
    for (int i=0; i<=100; i++) {
      if (!a[i] || i==100)
      {  
        #pragma omp task
        DoSomething(previousEnd+1,i);
      }  
    }
  }
}
```

It is important that the parameters to DoSomething be passed by value, not by reference, because previousEnd and i can change before or while the spawned task runs.
Consider this serial Fortran code:

```fortran
...  
logical(1) a(200) 
integer(4) i, previousEnd  
...  
previousEnd=0  
call annotate_site_begin(functions)  
do i=1,101  
   if a(.not. a(i)) .or. (i .eq. 101) then  
      call annotate_task_begin(do_something)  
      call DoSomething(previousEnd+1, i)  
      call annotate_task_end  
   endif  
end do  
call annotate_site_end
```

This is easily done using the OpenMP task directive. Without using this feature, such loops are extremely difficult to parallelize. One approach to the parallelize the above loop is simply to spawn each call to DoSomething():

```fortran
...  
logical(1) a(200) 
integer(4) i, previousEnd  
...  
previousEnd=0  
!$omp parallel  
!$omp single  
do i=1,101  
   if a(.not. a(i)) .or. (i .eq. 101) then  
      !$omp task  
      call DoSomething(previousEnd+1, i)  
      !$omp end task  
   endif  
end do  
!$omp end parallel
```

There is no requirement that the omp task pragma or directive be within the surrounding parallel directive's static extent.

**See Also**

Using Intel® Inspector and Intel® VTune™ Profiler

**Next Steps for the Parallel Program**

After you add parallel framework code to your program, use the related Intel® software products to check for parallel thread errors and improve the performance of your parallel program. Tips for debugging parallel code are also provided.

**Using Intel® Inspector and Intel® VTune™ Profiler**

Intel® Advisor helps you:

- Discover where to add parallelism to your program by identifying where your program spends its time. You propose parallel code regions when you annotate the parallel sites and tasks.
- Predict the performance you might achieve with the proposed parallel code regions.
- Predict the data sharing problems that could occur in the proposed parallel code regions.
Intel Advisor does not catch all problems, and it cannot ensure that you have correctly implemented the parallelism. Before deploying your parallel program, you need to test it for Dependencies and verify its performance. To do this, you can use analyzer tools provided in certain editions of the Intel® Parallel Studio XE suite.

The thread error analysis provided by Intel® Inspector and the Intel Advisor Dependencies analysis tool use similar technology. Intel Inspector includes a data race and deadlock detection tool that works on the parallel code. It can find more errors because it operates on the parallel code instead of working on the annotated serial code analyzed by the Dependencies tool. Intel Inspector also can find problems with memory: memory leaks, references to freed storage, references to uninitialized memory, and so forth. The memory-checking tool works on serial or parallel code.

Similarly, the Intel Advisor Survey and Suitability tools provide features found in the Intel® VTune™ Profiler. The Survey tool profiles your program to find hotspots and the Suitability tool makes predictions of approximate parallel performance including overhead costs based on the Intel Advisor annotations. When you have a working parallel program, you should use Intel VTune Profiler to measure the parallel program gain and core utilization, as well as check whether the parallel framework overhead is acceptable.

Once you have parallel code, you should:

- Measure the speedup.
- Make adjustments if locks are causing excessive delays, or if one task runs much longer than others.

Intel VTune Profiler has many features to help you find and fix performance problems in your parallel code. It also helps you check:

- Where are the hotspots now?
- Am I missing opportunities for more parallelism?
- Is my program spending a lot of time waiting?
- How does the performance compare to that of prior versions?

Another technique is to use a debugger to debug a serial version of your parallel program with the parallel constructs in reverse order (see Debugging Parallel Programs).

See Also

Maintaining Results

Debugging Parallel Programs

Your program might have bugs that are now being exposed during parallel execution because of changes in order, memory allocation, uninitialized memory contents, and so on.

Such bugs are debugged in the same way as a serial (single-threaded) application, with the following challenges:

1. The program does not run in exactly the same order each time. Possible causes include:
   - Locks may be acquired first by different threads.
   - Pointers returned by new and malloc may differ from one run to the next.
   - Random number sequences observed by a thread may differ from those observed in the serial version, and from run to run.
   - Items removed from a shared list by a thread may differ from run to run.

2. The debugger can interact in strange manners with the threads.
   - Breakpoints can appear to be hit multiple times by a thread, even though the thread only make progress through the breakpoint on last hit of the series.

3. Thread local storage can be difficult to examine.

To determine whether you can reproduce the bug with a single thread, run the parallel version of your program as a serial program by limiting the number of threads to 1. Use such methods as setting an environment variable before you run your program or by using the Intel® Threading Building Blocks (Intel® TBB) tbb::task_scheduler_init init(1); object.

Before spending a significant amount of time debugging the parallel code, you should try running the parallel loops and other parallel constructs as serial code but in reverse order. This may expose the bugs caused by your program depending on the order of execution of these statements, without requiring you to debug a parallel program.
Debugging the Remaining Sharing Problems

After your program works in serial mode, and in serial mode with the parallel constructs in reverse order, use the Intel® Inspector tool to find any remaining conflicts.

See Also
Using Intel Inspector and Intel® VTune™ Profiler
Using Partially Parallel Programs with Intel Advisor Tools

Notices and Disclaimers

Intel technologies may require enabled hardware, software or service activation.

No product or component can be absolutely secure.

Your costs and results may vary.

© Intel Corporation. Intel, the Intel logo, and other Intel marks are trademarks of Intel Corporation or its subsidiaries. Other names and brands may be claimed as the property of others.

Microsoft, Windows, and the Windows logo are trademarks, or registered trademarks of Microsoft Corporation in the United States and/or other countries.

Java is a registered trademark of Oracle and/or its affiliates.

No license (express or implied, by estoppel or otherwise) to any intellectual property rights is granted by this document.

The products described may contain design defects or errors known as errata which may cause the product to deviate from published specifications. Current characterized errata are available on request.

Intel disclaims all express and implied warranties, including without limitation, the implied warranties of merchantability, fitness for a particular purpose, and non-infringement, as well as any warranty arising from course of performance, course of dealing, or usage in trade.

Flow Graph Analyzer ships Qt*(version 5.12.2) libraries licensed under the GNU Lesser Public License (LGPL) or Runtime General Public License. Their source code can be downloaded from https://download.01.org/qt/source/

Flow Graph Analyzer ships Boost* C++ libraries licensed under the Boost Software License. Their source code can be downloaded from http://www.boost.org/users/history/version_1_62_0.html