Legal Information

INFORMATION IN THIS DOCUMENT IS PROVIDED IN CONNECTION WITH INTEL(R) PRODUCTS. NO LICENSE, EXPRESS OR IMPLIED, BY ESTOPPEL OR OTHERWISE, TO ANY INTELLECTUAL PROPERTY RIGHTS IS GRANTED BY THIS DOCUMENT. EXCEPT AS PROVIDED IN INTEL'S TERMS AND CONDITIONS OF SALE FOR SUCH PRODUCTS, INTEL ASSUMES NO LIABILITY WHATSOEVER, AND INTEL DISCLAIMS ANY EXPRESS OR IMPLIED WARRANTY, RELATING TO SALE AND/OR USE OF INTEL PRODUCTS INCLUDING LIABILITY OR WARRANTIES RELATING TO FITNESS FOR A PARTICULAR PURPOSE, MERCHANTABILITY, OR INFRINGEMENT OF ANY PATENT, COPYRIGHT OR OTHER INTELLECTUAL PROPERTY RIGHT. UNLESS OTHERWISE AGREED IN WRITING BY INTEL, THE INTEL PRODUCTS ARE NOT DESIGNED NOR INTENDED FOR ANY APPLICATION IN WHICH THE FAILURE OF THE INTEL PRODUCT COULD CREATE A SITUATION WHERE PERSONAL INJURY OR DEATH MAY OCCUR.

Intel may make changes to specifications and product descriptions at any time, without notice. Designers must not rely on the absence or characteristics of any features or instructions marked "reserved" or "undefined." Intel reserves these for future definition and shall have no responsibility whatsoever for conflicts or incompatibilities arising from future changes to them. The information here is subject to change without notice. Do not finalize a design with this information.

The products described in this document 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.

Contact your local Intel sales office or your distributor to obtain the latest specifications and before placing your product order.

Copies of documents which have an order number and are referenced in this document, or other Intel literature, may be obtained by calling 1-800-548-4725, or by visiting Intel's Web Site.

Parts of this product were built using third party libraries. Pursuant to the licenses ruling these libraries, Intel makes them available to users of this product. The libraries can be downloaded from Intel Software Development Products Knowledge Base article http://software.intel.com/enus/articles/open-source-downloads/ . Please note that download of these libraries is not required to use the product.

Celeron, Centrino, Intel, Intel logo, Intel386, Intel486, Intel Atom, Intel Core, Itanium, MMX, Pentium, VTune, and Xeon are trademarks of Intel Corporation in the U.S. and other countries.

* Other names and brands may be claimed as the property of others.
Copyright © 2010 Intel Corporation. All Rights Reserved.
# Contents

Legal Information .................................................................................................................. 2
Contents .................................................................................................................................. 3
Table of Figures ..................................................................................................................... 4
How to use this document ...................................................................................................... 5
Introduction ............................................................................................................................ 5
Section 1 – The debuggers key features ................................................................................ 6
  Requirements ....................................................................................................................... 6
  Overview ............................................................................................................................... 6
  General Features .................................................................................................................. 7
  Advanced Features .............................................................................................................. 8
Section 2 – Hands-on Tutorial .............................................................................................. 9
  Before you begin ................................................................................................................ 9
  Getting Started .................................................................................................................. 9
  Thread Awareness ............................................................................................................... 13
Thread Data Sharing Detection ............................................................................................ 17
Data Sharing Filters Usage Example .................................................................................... 18
  Use case 1: poking .......................................................................................................... 18
  Use case 2: Reproducing Inspector findings .................................................................... 23
Re-entrant Function Call Detection ..................................................................................... 25
Debugging SIMD Code ......................................................................................................... 26
Support for the Standard Template Library ........................................................................ 28
MPI Cluster Support ............................................................................................................. 29
How to get the debugger ....................................................................................................... 31
Appendix A – The Graphical User Interface ........................................................................ 33
# Table of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Open Executable dialog</td>
<td>10</td>
</tr>
<tr>
<td>2</td>
<td>Attach To Process dialog</td>
<td>11</td>
</tr>
<tr>
<td>3</td>
<td>Source Window in Intel® Debugger</td>
<td>11</td>
</tr>
<tr>
<td>4</td>
<td>Breakpoint dialog and thread filters</td>
<td>12</td>
</tr>
<tr>
<td>5</td>
<td>Variable view in source window and in the Locals (Local variable) window</td>
<td>13</td>
</tr>
<tr>
<td>6</td>
<td>Thread Filters</td>
<td>14</td>
</tr>
<tr>
<td>7</td>
<td>Threads window context menu</td>
<td>15</td>
</tr>
<tr>
<td>8</td>
<td>An Overview of Syncpoint Usage and Serialized Execution.</td>
<td>16</td>
</tr>
<tr>
<td>9</td>
<td>OpenMP Windows Access</td>
<td>16</td>
</tr>
<tr>
<td>10</td>
<td>Thread Data Sharing Event Filters to suppress unwanted event logging</td>
<td>17</td>
</tr>
<tr>
<td>11</td>
<td>Source code for barrier.cpp</td>
<td>18</td>
</tr>
<tr>
<td>12</td>
<td>Adding Data Sharing Filter</td>
<td>19</td>
</tr>
<tr>
<td>13</td>
<td>Data race detected but no symbol found for the memory location</td>
<td>20</td>
</tr>
<tr>
<td>14</td>
<td>Data race in compute called by new_phase3()</td>
<td>20</td>
</tr>
<tr>
<td>15</td>
<td>Set a focus filter on function compute()</td>
<td>21</td>
</tr>
<tr>
<td>16</td>
<td>Data race in compute() function</td>
<td>21</td>
</tr>
<tr>
<td>17</td>
<td>OpenMP Task Spawn Tree</td>
<td>22</td>
</tr>
<tr>
<td>18</td>
<td>Identify the coding error</td>
<td>23</td>
</tr>
<tr>
<td>19</td>
<td>Intel(R) Parallel Inspector findings</td>
<td>24</td>
</tr>
<tr>
<td>20</td>
<td>Data Sharing Events Report and Task Spawn Tree</td>
<td>25</td>
</tr>
<tr>
<td>21</td>
<td>Parallel Feature Pull down Menu</td>
<td>26</td>
</tr>
<tr>
<td>22</td>
<td>Vector Registers window</td>
<td>27</td>
</tr>
<tr>
<td>23</td>
<td>Register Data Organization in Vector Registers Window</td>
<td>27</td>
</tr>
<tr>
<td>24</td>
<td>Vector Evaluation window</td>
<td>28</td>
</tr>
<tr>
<td>25</td>
<td>STL Containers in the local variable window</td>
<td>28</td>
</tr>
</tbody>
</table>
How to use this document

This document has two sections. **Section 1 – The debuggers key features** describes the key features of the debugger. **Section 2 – Hands-on Tutorial** contains step-by-step instructions show these key features. **Appendix A – The Graphical User Interface** summarizes all the buttons and menus that are in the debugger.

Before using the debugger please read the paragraph **Before you begin**.

Introduction

This tutorial describes the Intel® Debugger for Linux* included with the Intel® Composer XE 2011 for Linux* distribution. With the debugger the application developer can debug the most complex of today’s multicore programs.

The Intel® Debugger for Linux* is an Eclipse* based debug solution. Its advanced features provide easy access to the thread-specific properties of the program under test.
Section 1 – The debuggers key features

Requirements

- Hardware Platform: IA-32, Intel®64 architecture or compatible.

- Software Platform: Linux* OS based environment
  - Java runtime environment (JRE 1.5 or JRE 1.6) to use the Eclipse* framework

- The Intel® Debugger works with the following compilers:
  - Intel® C++ Compilers and Intel® Fortran Compilers
  - gcc, g++, and g77 compilers

See the Intel® Debugger Manual for more information. Please also see the release notes for platform specific restrictions and known limitations of this release.

Overview

The Intel® Debugger for Linux* is a part of the Intel® C++ Composer XE 2011 for Linux* and Intel® Fortran Composer XE 2011 for Linux*. It is a symbolic source-code application debugger that helps programmers find run-time programming errors. These errors include wrong code, memory leaks, stack overflows, unexpected exceptions or other algorithmic problems. The Intel® Debugger can debug both single and multithreaded applications, serial and parallel code. In its latest version we added improved thread awareness and thread-specific breakpoint handling. The application run-time control supports lockstep stepping following a thread syncpoint, defining thread teams and even forced serialized execution of parts of your threaded application. Also we now provide OpenMP windows with information about current tasks, teams, task waits, barriers, task spawn trees and locks. A SSE Windows that gives you access to SSE registers commonly used for data vectorization and single instruction multiple data (SIMD) handling. It displays vectors as rows (or columns) in a table display. You can change layout the data in those registers to match what the application is using or view the data in any other way you like. Finally the debugger supports STL code.
General Features

The Intel® Debugger has a wide range of standard and advanced features. The standard features normally expected in a debugger include:

- Attaches to (and detaches from) a running process and debugs the matching program
- Loads a program into (and unloads a program from) the debugger, automatically creating and deleting processes as necessary
- Supports multiple-process debugging, associating with one or more programs:
  - Actively run one process at a time
  - Switch focus between processes
  - See processes and examine detailed process state
  - Set breakpoints for a specific process
- Supports remote debugging of applications on embedded Intel® architecture (using a remote agent)
- Debugs programs with shared libraries
- Debugs core files
- Provides language-specific command-expression evaluation
- Provides ability to “call” functions in a target process from within a command expression
- Catches or ignores unaligned access
- Displays the source listing of a program
- Sets breakpoints to stop program execution when specified sections of program code execute
- Sets watchpoints to stop program execution when the program writes to a memory address
- Adds conditions to breakpoints and watchpoints so program execution will only stop when the condition is true
- Supports setting of pending breakpoints if a breakpoint location cannot be resolved to an address at the time it is being set
- Steps both into or over calls to routines
- Steps through the execution of a program one source line or one machine instruction at a time
- Examines the stack of currently active functions
- Examines and changes program variables and data structure values in same or in different scopes
- Examines and changes the contents of memory in various formats (including international character strings)
- Disassembles and examines machine code
- Examines and changes machine register values
Tutorial Intel® Debugger

- Supports mixed-language applications, C++ templates, C++ user-defined operators, and Fortran modules
- Provides a customizable debugging environment by using environment variables, initialization files, sourced scripts, aliases (i.e., parameterized macros), and debugger variables for commands and command sequences
- Debug at both the source and assembly level.
- Debug from the command line or from the windowed version of the tool.

Advanced Features

Intel Debugger also provides:

- Regular expression searches of the symbol table
- Tracks breakpoints and watchpoints in shared libraries across program calls to dlopen() and dlclose()
- Debugs optimized code:
  - In-lined instances of functions (show in backtrace and selectable for current focus)
  - Registerized variables
  - Semantic stepping
  - PC-to-source column mapping (for multi-statement lines)
- Supports multi-core architecture:
  - Debugs thread-parallel applications that make use of pthread and OpenMP*
  - Provides details on OpenMP* locks, teams, and threads when debugging OpenMP* applications
  - Stops all threads when one stops, and restarts all when one restarts
  - Switches focus between threads
  - Sees all threads or individual threads, and examines detailed thread state
  - Sets breakpoints for all threads or for a subset of all threads
  - Detects thread data sharing events in parallel applications that make use of pthread, OpenMP*, Intel® Cilk™ Plus and Intel® Threading Building Blocks
- Supports cluster architecture:
  - Debugs cluster-parallel applications that make use of MPI-1
  - Supports debugging of MPI jobs that use mpich or the Intel MPI 3.0 libraries
  - Offers built-in cluster aggregation network
  - Includes user defined process sets

These advanced features extend the effectiveness of Intel Debugger well into problem areas that are traditionally difficult to debug.
Section 2 – Hands-on Tutorial

Before you begin

By default the debugger is installed in /opt/intel/composerxe-2011.0.xxx/bin/intel64 and /opt/intel/composerxe-2011.0.xxx/bin/ia32.

The easiest way of launching the Intel® Debugger is to first ‘source the compiler’. This will set up all the necessary paths so you can simply just invoke idb in any working directory:

$ source /opt/intel/composerxe-2011/bin/compilervars.sh ia32<cr>

Getting Started

There are three different way to start a debug session. The next few steps show how to debug the tachyon_with_cilk example using these three different ways:

a) In a terminal:

$ cd <path to tachyon_compiler directory> <cr>

$ idb -args tachyon_with_cilk dat/balls3.dat <cr>

This will open the debugger GUI and load the application.

b) You can achieve the same by invoking the debugger without arguments and in the GUI select ‘File/Open Executable’ from the menu or using the Load icon. In both cases an ‘Open Executable’ dialog (Figure 1) will be open:
Fill in all three entries: Executable File, Arguments and Working directory. Unless you use full pathnames, all references will be relative to the working directory.

Click OK and the debugger will load the application.

c) A third method would be to attach to an already running process. You can do that from the debugger GUI by selecting ‘File/Attach to Process’ from the menu or using the attach process icon. In both cases the ‘Attach To Process’ dialog (Figure 2) will open:
Figure 2 - Attach To Process dialog

Just double click on the process in the dialog list you would like to debug.

Once the debugger is attached to the process, the source file listing [if open] will be populated. The source file view will be opened at the code location where execution was halted.

Figure 3 - Source Window in Intel® Debugger

Source lines with symbol line information are marked with small blue dots. A double click on these dots will set a breakpoint. A right-click opens a context menu. Use this menu to set code breakpoints; run the application to current source line; set an advanced breakpoint; and more.
The Intel® Debugger features a breakpoint dialog for data and code breakpoints. The code breakpoint tab supports regular breakpoints as well as thread syncpoints (Figure 4).

![Create Breakpoint dialog](image)

**Figure 4 - Breakpoint dialog and thread filters**

Breakpoints can be conditional, for example the breakpoint only happening if a variable has a certain value. When a breakpoint is hit an action can be triggered. The Skip Count is used to allow the breakpoint to be ignored until the skip count value is reached.

Data structures such as variables, templates, structures and methods can be monitored and are updated any time you stop program execution. In Fig. 7 below you can see two ways of quickly finding out the value of a variable for example. On the bottom half of the screen shot we have opened the Local Variable window which will be updated each time you suspend the application or hitting a breakpoint. Any value changes will be shown in red colour. The other way would be to place the cursor over the variable you are interested in inside the source window. This is illustrated on the top half of the Figure 5 screen shot.
Figure 5 - Variable view in source window and in the Locals (Local variable) window

The arrow in front of the pointer ‘cur’, in Figure 5, indicates that this pointer can be expanded into a tree view of the structure it is pointing to.

Thread Awareness

Section 1 – The debugger’s key features outlines some of the key features of the Intel® Debugger. The debugger supports a variety of threading models including pthreads, OpenMP, Intel® Threading Building Blocks (Intel®TBB) and Intel® Cilk™ Plus.

The debugger supports debugging of both individual threads and groups of threads. A set of default thread groups is already predefined. The user can also define a custom group of threads (Figure 6).
Figure 6 shows the standard thread filter dialog with predefined thread sets that all the debugger run control features can be applied to. When defining your own custom set of threads, threads can be chosen from the currently active threads.

In the breakpoint dialog discussed when we talked about the graphical user interface (Figure 4) you can specify which threads will be stopped when the breakpoint is hit. The field ‘Stopping Threads’ contains a list of thread ids – default is all. All threads can be specified as comma separated thread id list or by using the new ThreadSet syntax (t:[list-expression]). If you select Browse for Thread Filer or Stopping Threads entries in the breakpoint dialog, the Thread Set filter will open (Figure 4).

A custom set of threads can be created from threads window’s context-sensitive menu (Figure 7). The custom set can then be used in the thread filter dialog (Figure 6) for breakpoints and run control.

The thread properties can be set in the threads window (Figure 7). An individual thread can be set to:

- Frozen – it will no longer execute until explicitly ‘Thawed’
- Thawed – release a thread which was earlier frozen
- Uninterrupted – this thread will run until it finished and no interrupt will stop it
Syncpoints are used to modify the parallel execution flow of all the threads in a given thread group. A Syncpoint allows having all threads in a trigger group stop execution at one defined point. The breakpoint is triggered when all selected threads have reached the syncpoint. From this syncpoint the developer can then do synchronized locked stepping. This behavior of a syncpoint is similar to how a thread barrier would act in regular parallel code execution flow. The difference is the debugger allows you to temporarily simulate this execution flow behavior to better understand interactions between different threads and identify possible threading issues.

Parallel applications implemented with OpenMP or Cilk Plus can be serialized. By selecting ‘Parallel/Serialize Parallel Regions’ menu entry or click on the ‘Serialize’ icon it is possible to force the parallel section to run serially. This can be useful to clarify whether a runtime issue is caused by a serial algorithmic problem or has been introduced when OpenMP or Cilk Plus directives were added. For applications built using OpenMP the code segment of the parallel region associated with the current program location will be put into serial single-threaded execution mode. If you are inside a parallel region the serialization will take effect when you next time enter a parallel region [not nested].

For applications built using Cilk Plus directives no new steals will happen when serialization is enabled.
Stepping parallel loops

Problem:
State investigation difficult. Threads stop at arbitrary positions (red line)

Parallel Debugger Support
Add Syncpoint to stop team threads at same location
Apply execution to $LockStep set.

User Benefit
Get and keep defined program State
Operations like private data comparison now meaningful

Serial Execution

Problem:
Parallel loop computes a wrong result. Is it a concurrency or algorithm issue?

Parallel Debugger Support
Runtime access to the OpenMP* num_thread property
Set to 1 for serial execution of next parallel block

User Benefit
Verification of an algorithm "on-the-fly" without slowing down the entire application to serial execution
On demand serial debugging without recompile/restart

Figure 8 - An Overview of Syncpoint Usage and Serialized Execution.

The Intel® Debugger has windows keeping the developer informed about any active OpenMP directives (Figure 9).
It is possible to display the status of OpenMP tasks, task spawn trees, teams and task waits. Barriers and locks can be examined which can help to understand the impact they have on the application’s performance.

**Thread Data Sharing Detection**

The debugger can detect thread data sharing problems. Detection is supported for programs built using native threads, OpenMP, Threading Building Blocks or Cilk Plus. Sometimes false positives may be reported with Threading Building Block and Cilk Plus programs. To enable this datarace detection feature, use the compiler option `-debug parallel`. When such an application is loaded in the debugger the “Enable Detection” menu entry and the “Enable Detection” icon becomes active.

You can enable data sharing event detection by clicking on the icon, which will change colour to indicate its enabled state. At the same time a dedicated data sharing event window will open. As soon as the debugger detect two [or more] different threads accessing the same memory location with at least one write, a data sharing event is reported. You can select the behavior of the debugger when such an event is triggered. Default the debugger will behave in the same way as a data access breakpoint, i.e. suspending execution. The Thread Data Sharing Events window will contain a reference to the accessed location and a tree view of which threads were involved. From any of the data sharing events displayed in the window you can link back to the associated source code or assembly code by double click on it.

![Thread Data Sharing Event Filters](image)

*Figure 10 - Thread Data Sharing Event Filters to suppress unwanted event logging.*

A data sharing event that has been detected does not indicate that an actual violation causing faulty program execution occurred. It does however indicate that you may want to have a closer look at these events as more than likely they could point to a potential problem.
You can suppress the reporting of an event from the context menu or by taking advantage of Data Sharing Event Filtering (Figure 10).

**Data Sharing Filters Usage Example**

I will use the below program (Figure 11) which contains a simple programming bug that turns into a data race. For this example, we assume that we know the program to be racy and that we added `new_phase3()`. So, we’re interested in debugging any new data race we might have added while ignoring the existing races that somebody else is working on.

```c
#include <omp.h>

#define N 2
#define M 4

static int data[M][N];
static int initial;
static int computations;

static void compute(int arr[N]) {
  omp_set_dynamic(false);
  omp_set_nested(true);

  #pragma omp parallel num_threads(N)
  {
    int i = omp_get_thread_num();
    int left = (i > 0 ? arr[i-1] : 0);
    int right = (i < N ? arr[i+1] : 0);

    #pragma omp barrier
    arr[i] = left + right;

    #pragma omp single
    computations++;
  }
}

static void phase1(void) {
  #pragma omp parallel for
  for (int i = 0; i < N; i++) {
    initial = N;
    for (int j = 0; j < N; j++) {
      data[i][j] = initial;
    }
  }
}

static void phase2(void) {
  int magic;
  for (int i = 0; i < N; i++) {
    magic = data[0][i];
    for (int j = 0; j < N; j++) {
      data[i][j] = magic;
    }
  }
}

static void new_phase3(void) {
  #pragma omp parallel for
  for (int i = 0; i < N; i++) {
    compute(data[0]);
  }
}

extern int main(void) {
  phase1();
  phase2();
  new_phase3();
  return 0;
}
```

**Figure 11 - Source code for barrier.cpp**

**Use case 1: poking**
In this example, we start with a vague idea of the problem. We will use suppress filters to ignore what we’re currently not interested in.

We enable data sharing detection, select the suppress filter set [default], and start debugging. We will soon get the first data sharing report; a read/write race on the variable initial in phase1(). This is one of the known races and we want to ignore it right now to focus on any problem in new_phase3(). We use the context menu to set a filter on the variable and continue debugging – Figure 12.

Instead of the data object filter, we could have used function or source line filters. Which filter to use depends on the application and on our debug goal. As a rule of thumb, we try to use as few filters as possible. This makes it easier for us to keep track of what we filtered and it does not unnecessarily slow down the analysis.

As we continue execution the next data race in phase2() appear. The array data is written to by two different threads. Again we apply a data object filter.

We repeat this process once more for a race on a local variable in phase2(). We note the access is reported as “???” – see Figure 13. This indicates the debugger could not find a symbol at the given address.
We set another data object filter and continue debugging. The filter is set on an address, this time. When the debugger cannot find a symbol, it uses the information provided in the detection report to set a filter. Such filters, like disassembly breakpoints, only make sense for the current debug session and should be removed before restarting the debuggee.

The next break is already at a location we might be interested in. The debugger reports an update race in compute() and the call stack tells us the function has been called by new_phase3() – Figure 14.
It is not clear from this data race detection what the real problem is. Let us restart the program and set a focus filter – focus on the compute function Figure 15. Delete the old information in the Thread data Sharing Filter window and use the context menu to define a new filter.

![New Code Range Filter](image)

*Figure 15 - Set a focus filter on function compute()*

When we now run the program the following race is reported:

The current thread executed a write to `arr[i]` at line 22. The debugger reports the accessed location as data ([0][0]) since that is the global variable `arr[i]` references - Figure 16 below.

![Barrier CPP Code](image)

*Figure 16 - Data race in compute() function*

The detection report further lists a read access at line 18 by a different thread. Line 18 reads `arr[i-1]` and thus accesses the neighboring thread’s array element.

The two accesses are supposed to be protected by a barrier at line 21. So what’s wrong?
The detection report lists the thread as well as an OpenMP task identifier for each access. If we now look for these tasks in the OpenMP Task Spawn Tree – Figure 17 below, we see that they belong to different OpenMP teams. Since barriers only affect the innermost parallel region, the accesses made from different teams are not protected.

The problem is that compute() has been called with the same argument by two different threads.

If we now look at the new_phase3() code we added, we see that we call compute() inside a parallel for loop with data[0] instead of data[i] – Figure 18 below.
From this small example you have seen how powerful the thread data sharing filters can be. The focus filter could be helpful if the location (code or data) is known but not the reason. It will allow you to use all the additional information the debugger can provide while concentrating on the data sharing event in the focus region.

**Use case 2: Reproducing Inspector findings**

In this example, we start with a good idea of where to look. We just don’t know exactly what to look for.
Inspector reports (Figure 19 above) a data race in compute(), the function that we call in new_phase3().

The write at line 22 races with reads at line 18 and 19 as well as with another write at line 22. Those accesses are supposed to be synchronized with the barrier at line 21 and the per-thread array location, respectively. So what’s wrong?

To get more information about the context in which the race accesses occur, we will try to reproduce it in the debugger.

We start by opening the Data Sharing Filter window. We select the focus filter set and add a filter for the accesses we’re interested in. We set a single address range filter covering lines 18 to 22. See dialog in Figure 15 [From: barrier.cpp:18 ; To: barrier.cpp:22].

We know there is a write/write race as well as read/write races. To speed up the analysis, we look for the write/write race first. We open the Data Sharing Event window and use the context menu to select Ignore Read Accesses.

In this example could have set the filter on line 22 instead of on lines 18 to 22.

Before we start debugging, we again check that all relevant files are compiled with Parallel Debug support enabled by using option –debug parallel; we recompile, if necessary.
When using Inspector we recommend disabling the parallel debug instrumentation. The parallel debug run-time library uses synchronization constructs that are not known to Inspector, so it may report races in libpdbx.so.

It is good practice to stop at main (or even earlier) and check the filter window. Filters that cannot be evaluated will be displayed in red. Make sure all relevant filters are blue before you continue debugging.

With all the filters in place and read accesses ignored the debugger will stop at the write\write data race.

Figure 20 - Data Sharing Events Report and Task Spawn Tree

As in the first use case, the problem becomes obvious when we look at the OpenMP Task Spawn Tree. OpenMP Task 19 and 24 belong to two different teams – Figure 20 above.

Re-entrant Function Call Detection

From the Intel® Debugger Parallel pull down menu (Figure 21) [select ‘Break on Re-entrant Call.’] you can access a dialog that turns on reentrant function call detection. The debugger will stop execution when two or more (different) threads call the function at the same time.
Thus it permits to debug and monitor accesses closely so you can decide what measures may need to be taken to ensure consistent execution behavior of your application. Prerequisite for the re-entrant function call detection again is compilation of your code with the Intel® C++ Compiler using the --debug parallel option. To enable detection you need to specify a memory address that can be resolved as belonging to a specific function in the dialog mentioned above. Any access timing conflicts between multiple threads to that function will then cause the execution to break.

**Debugging SIMD Code**

The Intel® Debugger provides access to MMX and SSE registers. Each register used to store a set of values which can be operated on by a single instruction. The set of values, known as a *vector*, are displayed as a table in the Vector Register Window (Figure 22).

Each register in the Vector Register Window works is displayed as a series values. The size of the values is user-configurable. The default data size is int32.
Take for example two 128 bit registers \_reg1 and \_reg2 (see Figure 23). They can be displayed as a vector of long integers. The row label shows the register name, the column label indicate the position within the register.

The layout of the rows and columns can be modified to better represent how your data is organized.

The Vector Evaluation Window an also be used to display vectors (Figure 24). Variables can be drag-and-dropped from the source or evaluation window into the vector evaluation window. The layout and value of the variable can be modified by the user.
The improved support for STL (Standard Template Library) in the latest release significantly simplifies debugging of C++ applications. You can now display and assign values element wise to a STL container as shown in Figure 25 below.

This facility is also used to improve the ability to evaluate data variables in applications using Intel®TBB. In addition there are improvements to the execution control of TBB code which makes it easier to come to the user portion of an application using TBB.
MPI Cluster Support

The Intel® IDB supports debugging of Message Passing Interface (MPI) applications launched by

- mpirun, an MPI launcher from mpich, a public domain implementation of MPI
- mpiexec, an MPI launcher from Intel® MPI 3.0 and later
- prun, a parallel launcher of Resource Management System* (RMS) from Quadrics

**NOTE: MPI DEBUG SUPPORT CURRENTLY NOT GUI ASSISTED**

MPI debug support is not directly assisted by the Intel® Debugger Graphical User Interface. Please use either the command line version of IDB called idbc, or initiate the debugger connection to the MPI node from the debugger command window inside the Intel® Debugger GUI.

To start your parallel application under debugger control, you need to have the environment variable `IDB_HOME` set to the directory that your debugger is in.

When debugging an application launched by mpich, issue the following command at the shell:

```
% mpirun -dbg=idb -np N [other mpich options] application [application arguments] [-idb idb options]
```

where `N` represents number of processes and `application` is the name of the MPP program you would like to debug.

The debugger can debug an MPI job that uses the MPICH* or the Intel MPI 3.0 libraries. In addition, it can attach to an existing MPI job that uses MPICH.

**NOTE: USE CORRECT MPIRUN_DBG SCRIPT**

You need to copy the `mpirun_dbg.idb` script that comes with the debugger’s package to the bin directory of your MPICH* installation. Also note that the GDB compatibility mode is now available for debugging the MPI applications.

When debugging an application launched from the Intel® MPI 3.0 and later libraries, issue the following command:

```
% mpiexec -idb -n <number of processes > [other Intel MPI options ] <executable > [arguments to the executable ]
```

where `N` represents number of processes and `application` is the name of the program you would like to debug.

To attach to an existing MPI job that uses mpich, issue the following command:

```
% idbc -pid launcher pid -parallelattach application
```
where the *launcher pid* is the ID of the launcher process that spawned all the processes in the job and the *application* is the name of the launcher executable. To find the ID of this process, use the command "ps -xf". For example:

```
$ ps -ax | grep cpi | grep p4pg
1488 pts/38 S 0:00 /users/user_name/mpich-1.2.7p1/examples/basic/cpi-
p4pg /users/user_name/mpich-1.2.7p1/mpich1_starters/PI1362-
p4wd /users/user_name/mpich-1.2.7p1/mpich1_starters
1489 pts/38 S 0:00 /users/user_name/mpich-1.2.7p1/examples/basic/cpi-
p4pg /users/user_name/mpich-1.2.7p1/mpich1_starters/PI1362-
p4wd /users/user_name/mpich-1.2.7p1/mpich1_starters
```

To attach to the process, use the ID with the lesser value:

```
$ idb -pid 1488 -parallelattach cpi
Intel(R) Debugger for IA-32 -based Applications, Version X, Build XXX
Reading symbolic information from /nfs/isv/home/user_name/mpich-1.2.7p1/examples/basic/cpi...done
Attached to process id 1488 ....
stopped at [<opaque> __nanosleep_nocancel(...) 0x001ba81b]
(idb) [0:3] Intel(R) Debugger for IA-32 -based Applications, Version X, Build XXX
[0:3] Reading symbolic information from /nfs/isv/home/user_name/mpich-1.2.7p1/examples/basic/cpi...done
%1 [0:3] Attached to process id [1488;1544] ....
(idb) [0:3] stopped at [int main(int, char**):29 0x08049d64]
[0:3] 29 sleep (1);
(idb) w
(idb) [0:3] 24 MPI Comm size(MPI COMM WORLD,&numprocs);
[0:3] 25 MPI Comm rank(MPI COMM WORLD,&myid);
[0:3] 26 MPI Get processor name(processor name,&namelen);
[0:3] 27
[0:3] 28 while (!stop) {
[0:3] > 29 sleep (1);
[0:3] 30 }
[0:3] 31
[0:3] 32 fprintf(stderr,"Process %d on %s\n",myid, processor name);
(idb) assign stop=1
(idb) n
(idb) [0:3] stopped at [int main(int, char**):28 0x08049d52]
[0:3] 28 while (!stop) {
(idb)
[0:3] stopped at [int main(int, char**):28 0x08049d58]
[0:3] 28 while (!stop) {
(idb)
[0:3] stopped at [int main(int, char**):32 0x08049d69]
[0:3] 32 fprintf(stderr,"Process %d on %s\n",
(idb)
```

For a list of issues with MPI support, see the "Known Limitations" section in the Release Notes.

When the debugger starts your parallel application, it detects and attaches to all of your application's processes. At this point, your application stops before executing any user code and the debugger displays a prompt.
You can now set any necessary breakpoints and use the `continue` command to continue the execution of your application.

The debugger can be used to debug applications launched by prun using the command below. $N$ is the number of nodes and $n$ the number of processes.

```
$ $IDB_HOME/idb [idb options] -parallel `which prun` -n n -N N [other prun options] application [application arguments]
```

To start your parallel application under debugger control, you need to have the environment variable `IDB_HOME` set to the directory that your debugger is in. When debugging an application launched by mpich, issue the following command at the shell. $N$ represents number of processes and `application` is the name of the MPP program you would like to debug:

```
mpirun -dbg=idb -np N [other mpich options] application [application arguments] [-idb idb options]
```

Make sure there is a file called `mpirun_dbg.idb` in the directory in which `mpirun` is located. Also note the Intel IDB option `-gdb` is not yet supported under parallel debugging session.

When the debugger starts your parallel application, it detects and attaches to all of your application's processes. At this point, your application stops before executing any user code and the debugger displays a prompt.

You can now set any necessary breakpoints and use the `continue` command to continue the execution of your application.

## How to get the debugger

Intel Debugger is a component of the Intel® Composer XE 2011 for Linux* which is part of Intel® Parallel studio XE 2011. The product can be purchased, or an evaluation copy can be downloaded, from the Intel® Software Network ([http://software.intel.com](http://software.intel.com)).

On the same page you can find software product information, user forums, blogs, knowledge base articles and information how to get product support.
Optimization Notice

Intel® compilers, associated libraries and associated development tools may include or utilize options that optimize for instruction sets that are available in both Intel® and non-Intel microprocessors (for example SIMD instruction sets), but do not optimize equally for non-Intel microprocessors. In addition, certain compiler options for Intel compilers, including some that are not specific to Intel micro-architecture, are reserved for Intel microprocessors. For a detailed description of Intel compiler options, including the instruction sets and specific microprocessors they implicate, please refer to the “Intel® Compiler User and Reference Guides” under “Compiler Options.” Many library routines that are part of Intel® compiler products are more highly optimized for Intel microprocessors than for other microprocessors. While the compilers and libraries in Intel® compiler products offer optimizations for both Intel and Intel-compatible microprocessors, depending on the options you select, your code and other factors, you likely will get extra performance on Intel microprocessors.

Intel® compilers, associated libraries and associated development tools may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include Intel® Streaming SIMD Extensions 2 (Intel® SSE2), Intel® Streaming SIMD Extensions 3 (Intel® SSE3), and Supplemental Streaming SIMD Extensions 3 (Intel® SSSE3) instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors.

While Intel believes our compilers and libraries are excellent choices to assist in obtaining the best performance on Intel® and non-Intel microprocessors, Intel recommends that you evaluate other compilers and libraries to determine which best meet your requirements. We hope to win your business by striving to offer the best performance of any compiler or library; please let us know if you find we do not.
Appendix A – The Graphical User Interface

The graphical user interface (Fig. 1) supports a rich user experience with access to all the standard debug features such as stepping, breakpoints, callstack, source code switching, disassembly, variable views, and memory windows. The online help describes these features.

![Figure A.1 - Intel® Debugger Graphical User Interface](image)

In Fig. 2 below we take a closer look at the debugger menu bar illustrating the location and view of some of the buttons used to access key features.
**Run Control:**

- Run/Continue
- Suspend execution
- Kill current process
- Restart
- Step into
- Step over
- Run until caller
- Asm instruction step into
- Asm instruction step over

**Parallelism Support and Parallel Structure Views (OpenMP, Cilk Plus):**

- Enable Detection
- Stop on Event
- Break on Re-entrant Call...
- Thread Data Sharing Events
- Thread Data Sharing Filters
- Serialize Parallel Regions
- Tasks
- Barriers
- Taskwaits
- Locks
- Teams
- Task Spawn Tree

*Figure A.2 - Intel® Debugger menu bar and buttons for key features*