



Quick Reference Guide to Optimization with Intel® Parallel Composer

For IA-32 processors and Intel® 64 processors

Intel® Software Development Products

Application Performance

A Step-by-Step Approach to Application Tuning with Intel® Parallel Composer

In this product version, all optimization levels assume support for the SSE2 instruction set by default. On older IA-32 processors such as the Intel® Pentium® III processor, the option `/arch:IA32` must be added.

1. Start with "Release" build configuration in the Microsoft Visual Studio* IDE, which defaults to `/O2` optimization for performance tuning. Choose the `/O3` Optimization setting for maximum optimization.
2. Optimize your application for simultaneous multithreading, multicore, and multi-processor systems using the parallel performance options as described in the sections "Multithreading Support," "Commonly Used OpenMP* Directives," "Language Extensions for Parallelism," "Intel® Threading Building Blocks Constructs (Intel® TBB)," and Intel® Integrated Performance Primitives (Intel® IPP)."
3. See the table for "Intel® Parallel Debugger Extension for Microsoft Visual Studio*" for details on debugging parallelism issues related to OpenMP use.
4. Use Intel® Parallel Inspector to help you diagnose hard-to-find threading and memory errors in your application to reduce your time to market.
5. Fine-tune performance to target IA-32 and Intel® 64-based systems with processor-specific options such as `/QxSSE4.2` for Intel® Core™ i7 processor. Alternatively, you can use `/QxHOST`, which will optimize for and use the most advanced instruction set for the processor on which you compiled. Processor-specific optimization options can be set via the Microsoft Visual Studio* Code Generation property page value for "Use Intel® Processor Extensions." See the table "Recommended Processor-Specific Optimization Options for IA-32 and Intel® 64 Processors."
6. For applications using floating-point calculations, select the appropriate `/fp` option from the table "Floating-Point Arithmetic Consistency and Precision."
7. Use the Intel® Parallel Amplifier to help you identify performance hotspots, locks, waits, and concurrency analysis for your application that could benefit from further tuning.
8. Once you have identified performance tuning opportunities, you may want to provide Intel® Parallel Composer with more information to fine-tune specific functions that use `#pragma`. The optimization and vectorization reports may indicate places where loops could not be optimized fully due to pointer aliasing or memory-access overlaps, for example. See the table "Fine-Tuning (All Processors)" for details on using the optimization and vectorization reports.
9. To identify additional parallelism opportunities in your application, you can download the Intel® Parallel Advisor Lite from whatif.intel.com.*

Please consult the Intel Parallel Composer Documentation and the "Optimizing Applications with the Intel® Parallel Composer" white paper at the link <http://software.intel.com/en-us/articles/optimizing-applications-with-intel-parallel-composer-c-compiler/> for more details.

Please refer to the following article for more information on Intel Parallel Composer's parallelization features:

<http://software.intel.com/en-us/articles/intel-parallel-composer-parallelization-guide/>

General Optimization Options

Before you begin performance tuning, you may want to check correctness of your application by building it without optimization using `/Od`. Begin performance tuning with `/O1`, `/O2`, or `/O3`. These are general optimization options that should be at the heart of any application tuning for all 32-bit and 64-bit Intel® processors. Measure your performance before proceeding with more advanced options.

C++ Property Page and Field	Option	Comment
Optimization, Optimization	<code>/Od</code>	No optimization. Used during the early stages of application development and debugging. Use a higher setting when the application is working correctly.
Optimization, Optimization	<code>/O1</code>	Optimize for size. Omits optimizations that tend to increase object size. Creates the smallest optimized code in most cases. This option is useful in many large server/database applications where memory paging due to larger code size is an issue.
Optimization, Optimization	<code>/O2</code>	Maximize speed. Default setting. Enables many optimizations, including vectorization. Creates faster code than <code>/O1</code> in most cases.
Optimization, Optimization	<code>/O3</code>	Enables <code>/O2</code> optimizations plus more aggressive loop and memory-access optimizations, such as scalar replacement, loop unrolling, code replication to eliminate branches, loop blocking to allow more efficient use of cache, and additional data prefetching. The <code>/O3</code> option is particularly recommended for applications that have loops that do many floating-point calculations or process large data sets. It may occasionally slow down other types of applications compared to <code>/O2</code> .
General, Debug Information Format	<code>/ZI</code>	Generates debug information for use with any of the common platform debuggers. This option turns off <code>/O2</code> and makes <code>/Od</code> the default unless <code>/O2</code> (or another option) is specified.
Command Line, Additional Options	<code>/debug:full</code>	Produces full debugging information including symbol table information needed for full symbolic debugging of unoptimized code and global symbol information needed for linking. If this option is used with optimized code, full symbol information will be generated including the local symbol table information. This may result in minor performance degradation.

Multithreading Support (/Qparallel and /Qopenmp)

C++ Property Page and Field	Option	Comment
Language, OpenMP Support	/Qopenmp	Enables the parallelizer to generate multithreaded code based on the OpenMP* directives and Intel C++ Compiler language extensions. May require an increased stack size.
Command Line, Additional Options	/Qopenmp-report {0 1 2}	Controls the OpenMP* diagnostic levels as follows: 0 – Displays no diagnostic information (default) 1 – Displays diagnostic messages indicating loops, regions, and sections successfully parallelized 2 – Adds information on successful handling of MASTER constructs, SINGLE constructs, CRITICAL constructs, ORDERED constructs, and ATOMIC directives Must be used in conjunction with /Qopenmp.
Optimization, Parallelization	/Qparallel	Detects simply structured loops capable of being executed safely in parallel, and automatically generates multithreaded code for these loops.
Command Line, Additional Options	/Qpar-report {0 1 2 3}	Controls the auto-parallelizer's diagnostic levels as follows. Must set /Qparallel. 0 – Displays no diagnostic information (default) 1 – Indicates loops successfully parallelized 2 – Adds information on loops that were not parallelized 3 – Adds information about any proven or assumed dependencies inhibiting auto-parallelization (reasons for not parallelizing)

Language Extensions for Parallelism

Intel® Parallel Composer uses simple C and C++ language extensions to make parallel programming easier. This is accomplished through the use of keywords that are used as statement prefixes.

Keyword	Description	Compatible OpenMP directives
__par	Specifies that all iterations of the "for" statements are independent of each other and may be executed in parallel.	#pragma omp parallel for
__critical	Ensures that concurrent access to shared variables does not leave data in an inconsistent state.	#pragma omp critical
__taskcomplete S1	Creates an environment for parallel execution of tasks within a specified statement. Sequential execution of the program resumes after the taskcomplete statement completes.	#pragma omp parallel #pragma omp single { S1 }
__task S2	Executes the S2 statement as a separate task. Execution of the task that encountered the __task statement may proceed past the statement without waiting for its completion.	#pragma omp task { S2 }

For the application to benefit from the parallelism afforded by the above keywords, the switch /Qopenmp must be used during compilation. These parallelism extensions utilize the OpenMP* runtime library, but abstract out the use of the OpenMP pragmas, keeping the code more naturally written in C or C++. The mapping between the parallelism extensions and the OpenMP constructs are defined in the column "Compatible OpenMP directives" in the table above.

Commonly Used OpenMP* Directives

Intel® Parallel Composer supports OpenMP* 3.0 and is compatible with the OpenMP support in Microsoft Visual C++. By adding OpenMP pragmas and "#include <omp.h>" to the source code and compiling the application with the `/Qopenmp` option, you have a parallelized application. You can control the multithreaded code execution by setting the environment variables `OMP_NUM_THREADS`, `OMP_SCHEDULE`, `OMP_DYNAMIC`, etc.

Directive	Description
<code>#pragma omp parallel for [clause] ... for - loop</code>	Parallelizes the loop that immediately follows the parallel for directive. Execution of the parallel loop is controlled through optional clauses such as "private," "shared," "schedule," and so on.
<code>#pragma omp parallel sections [clause] ...</code> { <code>[#pragma omp section structured-block] ...</code> }	Distributes the execution of the different sections among the threads in the parallel team. Each section is executed once, and each thread executes zero or more sections.
<code>#pragma omp master structured-block</code>	The code contained within the master construct is executed by the master thread in the thread team.
<code>#pragma omp critical [(name)] structured-block</code>	Provides mutual exclusion access to the structured-block. An optional <i>name</i> may be used to identify the critical construct. All critical constructs without a name are considered to have the same unspecified name. A thread waits at the beginning of a critical region until no thread is executing a critical region with the same name. The critical construct enforces exclusive access with respect to all critical constructs with the same name in all threads, not just those threads in the current team.
<code>#pragma omp barrier</code>	Used to synchronize the execution of multiple threads within a parallel region. Ensures all the code occurring before the barrier has been completed by all the threads, before any thread can execute any of the code past the barrier directive.
<code>#pragma omp atomic expression-statement</code>	Provides mutual exclusion via hardware synchronization primitives. While a critical section provides mutual exclusion access to a block of code, the atomic directive provides mutual access to a single assignment statement.
<code>#pragma omp threadprivate (list)</code>	Specifies a list of variables that are replicated, with each thread getting its own private copy.

Please refer to the following document for more information on OpenMP directives and their use:

OpenMP Application Program Interface, Version 3.0 May 2008 at <http://openmp.org/wp/openmp-specifications/>

Intel® Threading Building Blocks Constructs (Intel® TBB)

The Intel® Threading Building Blocks in the Intel® Parallel Composer offers a rich methodology to express parallelism in C++ programs. Intel® TBB is a C++ template library that represents a high-level task-based parallelism that abstracts platform details and threading mechanism for performance and scalability. Please refer to the following link for more information on Intel Threading Building Blocks: <http://www.intel.com/cd/software/products/asm-na/eng/340571.htm>

Use the "Select Build Components" dialog from the Intel Parallel Composer within Visual Studio* IDE to set the configuration to use Intel TBB.

Construct	Description
<code>parallel_for<Range,Body,Partitioner></code>	A <code>parallel_for(range, body, partitioner)</code> represents parallel execution of <code>body</code> over each value in <code>range</code> . The optional <code>partitioner</code> specifies a <code>partitioning</code> strategy.
<code>parallel_reduce<Range,Body></code> <code>parallel_reduce<Range,Value,RealBody,Reduction></code>	A <code>parallel_reduce(range, body)</code> executes the function <code>body</code> over the objects specified in <code>range</code> , and with the aid of an auxiliary <code>join</code> function combines them together to form a final result. Alternately, <code>value parallel_reduce(range, value, real body, reduction)</code> is a syntactic variant that allows <code>real body</code> and <code>reduction</code> to be specified as <code>lambda</code> constructs. The parameter <code>value</code> specifies how to initialize the reduction temporaries; the final accumulation is returned by the function.
<code>parallel_scan<Range,Body></code>	A <code>parallel_scan(range, body)</code> computes a parallel prefix, also known as parallel scan.
<code>parallel_do<InputIterator,Body></code>	A <code>parallel_do(first, last, body)</code> applies a function object <code>body</code> over the half-open interval <code>[first, last)</code> . Items may be processed in parallel. Additional work items can be added by <code>body</code> if it has a second argument of type <code>parallel_do_feeder<T></code> . The function terminates when <code>body(x)</code> returns for all items <code>x</code> that were in the input sequence.
pipeline Class	Performs pipelined execution, with support for parallel or serial filters in pipelines.
<code>parallel_sort<RandomAccessIterator,Compare></code>	Template function that performs an unstable sort of a sequence defined by random access iterators <code>[begin, end]</code> . An unstable sort might not preserve the relative ordering of elements with equal keys. The sort is deterministic; sorting the same sequence will produce the same result each time. See Intel® TBB documentation for complete requirements for the specification of the <code>RandomAccessIterator</code> .
<code>concurrent_queue<T,Allocator></code>	A bounded first-in, first-out data structure that permits multiple threads to concurrently push and pop items.
<code>concurrent_hash_map<Key,T,HashCompare,Allocator></code>	Maps keys to values in a way that permits multiple threads to access and manipulate key-value pairs concurrently.
<code>concurrent_vector<T,Allocator></code>	An enhanced version of the standard vector class that supports concurrent growth and access from multiple threads.
<code>tbb_allocator<T></code>	A convenient means to specify the Intel® TBB scalable allocator for those constructors that take allocator arguments; reverts to the standard allocator if a scalable allocator is not available.
<code>cache_aligned_allocator<T></code>	Allocates memory on cache line boundaries, in order to avoid false sharing.
<code>scalable_allocator<T></code>	Allocates and frees memory in a way that scales with the number of processors.
<code>atomic<T></code>	Supports atomic read, write, fetch-and-add, fetch-and-store, and compare-and-swap.
mutex Class	Models Mutex Concept using underlying OS locks. It is a wrapper around OS calls that provide mutual exclusion. There are six different variants of the mutex class for performance or capability requirements.

Intel® Integrated Performance Primitives (Intel® IPP)

The following table contains a list of Intel® IPP function domains and their main function operations or algorithms supported. Listed under each function domain, for example "(ippi*.*)", is the corresponding header, static and dynamic library file names (for example, header: ippi.h; static: ippi.lib; dynamic: ippiemerged.lib and ippiemerged.lib). The Intel IPP libraries come with extensive samples in each domain.

Use the "Select Build Components" dialog from the Intel® Parallel Composer within Visual Studio* IDE to set the configuration to use Intel IPP or Intel IPP Cryptography.

Domain	Functions
1. Image processing (ippi*.*)	<ul style="list-style-type: none">• Geometry transformations, such as resize/rotate• Linear and nonlinear filtering operation on an image for edge detection, blurring, noise removal, and so on for filter effect• 2-D Linear transforms FFT, DFT, DCT• Image statistics and analysis
2. Color conversion (ippcc*.*)	<ul style="list-style-type: none">• Converting image/video color space formats: RGB, HSV, YUV, YCbCr• Up/Down sampling• Brightness and contrast adjustments
3. JPEG Coding (ippj*.*)	<ul style="list-style-type: none">• High-level JPEG and JPEG2000 compression and decompression functions• JPEG/JPEG2000 support functions: DCT, Wavelet transforms, color conversion, downsampling
4. Video Coding (ippvc*.*)	<ul style="list-style-type: none">• VC-1, H.264, AVS, MPEG-2, MPEG-4, H.261, H.263; and DV codec support functions
5. Computer Vision (ippcv*.*)	<ul style="list-style-type: none">• Background differencing, Feature Detection (Corner Detection, Canny Edge detection), Distance Transforms, Image Gradients, Flood fill, Motion analysis and Object Tracking, Pyramids, Pattern recognition, Camera Calibration
6. Realistic Rendering (ipp*.*)	<ul style="list-style-type: none">• Acceleration Structures, Ray-Scene Intersection, and Ray Tracing• Surface properties, shader support, tone mapping
7. Signal Processing (ipps*.*)	<ul style="list-style-type: none">• Transforms: DCT, DFT, MDCT, Wavelet (both Haar and user-defined filter banks), Hilbert• Convolution, Cross-Correlation, Auto-Correlation, Conjugate• Filtering: IIR/FIR/Median Filtering, Single/Multirate FIR LMS filters• Other: Windowing, Jaehne/Tone/Triangle signal generation, Thresholding
8. Audio Coding (ippac*.*)	<ul style="list-style-type: none">• MP3, AAC, HE-AAC, AC3
9. Speech Coding (ippsc*.*)	<ul style="list-style-type: none">• Adaptive/Fixed Codebook functions, Autocorrelation, Convolution, Levinson-Durbin Recursion, Linear Prediction Analysis and Quantization, Echo Cancellation, Companding
10. Speech Recognition (ippsr*.*)	<ul style="list-style-type: none">• Feature Processing, Model Evaluation/Estimation/Adaptation, Vector Quantization, Polyphase Resampling, Advanced Aurora, Ephraim-Malah Noise Suppression, AEC, Voice Detection
11. Data Compression (ippdc*.*)	<ul style="list-style-type: none">• Entropy-coding compression: Huffman, VLC• Dictionary-based compression: LZSS, LZ77• Burrows-Wheeler Transform, MoveToFront, RLE, Generalized Interval Transformation• Compatible feature support for zlib, bzip2, gzip, and lzo
12. Cryptography (ippcp*.*)	<ul style="list-style-type: none">• Big-number Arithmetic/Rijndael, DES, TDES, SHA1, MD5, RSA, DSA, Montgomery, prime number generation, and pseudo-random number generation (PRNG) functions
13. String Processing (ippch*.*)	<ul style="list-style-type: none">• Compare, Insert, Change Case, Trim, Find, Regexp, Hash
14. Small Matrix (ippm*.*)	<ul style="list-style-type: none">• Addition, Multiplication, Decomposition, Eigenvalues, Cross-product, Transposition
15. Vector Math (ippvm*.*)	<ul style="list-style-type: none">• Logical, Shift, Conversion, Power, Root, Exponential, Logarithmic, Trigonometric, Hyperbolic, Erf
16. Data Integrity (ippdi*.*)	<ul style="list-style-type: none">• GF(2^m) Arithmetic Functions• Arithmetic Functions for Polynomials over GF(2^m)• Reed-Solomon Code Functions

Visit <http://software.intel.com/en-us/articles/developing-ipp-multicore-applications/> for more details on how to develop Intel IPP Applications for multicore processors.

Recommended Processor-Specific Optimization Options for IA-32 and Intel® 64 Architectures

The `/Qx` switches optimize for, and are recommended for best performance on, the corresponding or later Intel processors. For example, use the `/QxSSE4.2` switch for Intel® Core™ i7 processor. The `/arch` switches are recommended for excellent performance on all processors that support the corresponding instruction set, including compatible non-Intel processors. (e.g., `/arch:SSE3` for processors that support SSE3). The `/Qax` options will produce a binary optimized for the corresponding Intel processor that contains a second default code path optimized for any processor, including compatible non-Intel processors that support SSE2 instructions.

C++ Property Page and Field	Option	Comment
Code Generation, Intel Processor-Specific Optimization	<code>/Qx {SSE4.2 SSE4.1 SSSE3 SSE3 SSE3_ATOM SSE2 HOST}</code>	<p>Generates specialized code for the indicated Intel processor. The executable should only be run on the targeted or later Intel processors.</p> <p>SSE4.2 – May generate SSE4, SSSE3, SSE3, SSE2, and SSE instructions for Intel processors, including SSE4 Efficient Accelerated String and Text Processing. Optimizes for the Intel® Core™ processor family (e.g., the Intel® Core™ i7 processor).</p> <p>SSE4.1 – May generate SSE4 Vectorizing Compiler and Media Accelerators, SSSE3, SSE3, SSE2, and SSE instructions for Intel® processors. Optimizes for Intel® 45nm Hi-k next-generation Intel Core™ microarchitecture.</p> <p>SSSE3 – May generate SSSE3, SSE3, SSE2, and SSE instructions for Intel processors. Optimizes for the Intel® Core™2 processor family.</p> <p>SSE3 – May optimize and generate SSE3, SSE2, and SSE instructions for Intel processors. Performs optimizations not enabled with <code>/arch:SSE3</code>.</p> <p>SSE3_ATOM – May generate SSE3, SSE2, and SSE instructions for Intel processors. Optimizes for the Intel® Atom™ processor and Intel® Centrino® processor technology.</p> <p>SSE2 – May optimize and generate SSE2 and SSE instructions for Intel processors. Performs optimizations not enabled with <code>/arch:SSE2</code>.</p> <p>HOST – May optimize and generate any instructions that are supported by the compilation host. On Intel processors, this may correspond to the most suitable <code>/Qx</code> option; on compatible non-Intel processors, this may correspond to the most suitable <code>/arch</code> option. The resulting executable may not run on processors that do not support all the instruction sets supported by the compilation host.</p>
Code Generation, Add Processor-Optimized Code Path	<code>/Qax {SSE4.2 SSE4.1 SSSE3 SSE3 SSE3_ATOM SSE2}</code>	<p>Generates specialized code for the corresponding Intel processors while also generating a default code path corresponding to <code>/arch:SSE2</code>. This default code path may be modified by using, in addition, a <code>/Qx</code> or <code>/arch</code> switch. For example, for best performance on the Intel® Core™ i7 processor and also good performance on compatible non-Intel processors that support SSE3, use <code>/QaxSSE4.2 /arch:SSE3</code>.</p> <p>This will produce binaries with one code path that takes full advantage of the Intel® Core™ i7 processor; the other code path will run well on both Intel and compatible non-Intel processors that support SSE3 but not SSE4. At runtime, the application automatically selects the code path that is appropriate for the processor on which it is running.</p>
Code Generation, Enable Enhanced Instruction Set	<code>/arch: {SSE3 SSE2 IA32}</code>	<p>Generates optimized code that may make use of the specified instruction sets. The executable should only be run on processors supporting the specified instruction sets.</p> <p>SSE3 – May generate SSE3, SSE2, and SSE instructions. Code path may execute on Intel and compatible non-Intel processors that support SSE3.</p> <p>SSE2 – May generate SSE2 and SSE instructions. Code path may execute on Intel and compatible non-Intel processors that support SSE2. (default)</p> <p>IA32 – Generates code without any extended instruction sets that will run on any Intel® Pentium® or later Intel processor or compatible non-Intel processor. [IA-32 architecture only].</p>

Source Code Analysis

Static Code Analysis (also called "Parallel Lint") helps identify some potential application errors, such as misuses of OpenMP* directives, threadprivate variables, C++ exceptions and potential deadlocks or data races.

C++ Property Page and Field	Option	Comment
Diagnostics, Level of Source Code Parallelization Analysis	/Qdiag-enable:sc-parallel [1 2 3]	Specifies diagnostic messages issued by the Source Code Analyzer. 1 - Produces the diagnostics with severity level set to all critical errors 2 - Produces the diagnostics with severity level set to all errors. This is the default if n is not specified. 3 - Produces the diagnostics with severity level set to all errors and warnings
Diagnostics, Analyze of Include Files	/Qdiag-enable:sc-include	Tells the Source Code Analyzer to analyze include files and source files when issuing diagnostic messages.

Interprocedural Optimization (IPO)

IPO includes more aggressive function-inlining to reduce function call overhead and expose more optimization opportunities, such as constant and routine attribute propagation, dead code elimination, and forward substitution. However, IPO can increase code size and compile time. Be sure to measure your application performance, compile time, and code size tradeoffs when using these options.

C++ Property Page and Field	Option	Comment
Optimization, Interprocedural Optimization	/Qipo[value]	Permits inlining and other interprocedural optimizations among multiple source files. The optional value argument controls the maximum number of link-time compilations (or number of object files) spawned. Default for value is 0 (the compiler chooses). Caution: This option can in some cases significantly increase compile time and code size.
Command Line, Additional Options	/Qipo-jobs[n]	Specifies the number of commands (jobs) to be executed simultaneously during the link phase of Interprocedural Optimization (IPO). The default is 1 job.

Floating-Point Arithmetic Consistency and Precision

Intel® Parallel Composer provides options for enhancing the consistency or precision of floating-point results on all Intel® architectures, at some cost in performance. Refer to the Compiler Options section of the Intel Parallel Composer Documentation for detailed information on floating-point options.

C++ Property Page and Field	Option	Comment
Code Generation, Floating Point Model	<code>/fp:name</code>	<p>Controls the consistency and precision of floating-point results by restricting certain optimizations. The possible values of name are:</p> <ul style="list-style-type: none">fast=[1]2 – Allows more aggressive optimizations at a slight cost in accuracy or consistency. (fast=1 is the default)precise – Enables only value-safe optimizations on floating-point codeexcept – Enforces floating-point exception semanticsstrict – Strictest mode of operation, enables both the precise and except options and disables fma contractionssource – Rounds intermediate results to source-defined precision and enables value-safe optimizationsdouble – Rounds intermediates in 53-bit (double) precisionextended – Rounds intermediates in 64-bit (extended) precision <p>Recommendation: <code>/fp:precise /fp:source</code> is the recommended form for the majority of situations where enhanced floating-point consistency and reproducibility are needed.</p> <p>For complete information on this option, refer to the Intel® Parallel Composer Documentation.</p>

Fine-Tuning (All Processors)

Once you have identified performance “hotspots,” locks, waits, and concurrency analysis, you may need to provide Intel® Parallel Composer with more information to fine-tune specific functions. The optimization and vectorization reports may show places where loops could not be optimized fully due to pointer aliasing or memory-access overlaps, for example. The Intel Parallel Composer Documentation includes details on other `#pragmas`, directives, and intrinsics that can be used to control loop unrolling, vectorization, and prefetching for further fine-tuning within your application code.

C++ Property Page and Field	Option	Comment
Language, Recognize the Restrict Keyword	<code>/Qrestrict[-]</code>	Enables [disables] pointer disambiguation with the restrict keyword. Off by default.
Diagnostics, Optimization Diagnostic Level	<code>/Qopt-report:[n]</code>	Generates an optimization report directed to stderr. n specifies the level of detail, from 0 (no report) to 3 (maximum detail). Default is 2.
Diagnostics, Optimization Diagnostic Phase	<code>/Qopt-report-phase:name</code>	Optimization reports are generated for phase name . The option can be used multiple times in the same compilation to get output from multiple phases. Some commonly used name arguments are as follows: <ul style="list-style-type: none">all – All possible optimization reports for all phases (default)ipo_inl – Inlining report from the Interprocedural Optimizerhlo – High-Level Optimizer (includes loop and memory optimizations)hpo – High-Performance Optimizer (includes vectorizer and parallelizer)
Command Line, Additional Options	<code>/Qvec-report [n]</code>	Controls the vectorizer’s diagnostic levels as follows: <ul style="list-style-type: none">n = 0: no information (default)n = 1: indicates vectorized loopsn = 2: indicates vectorized and nonvectorized loopsn = 3: indicates vectorized loops and explains why other loops were not vectorizedn = 4: indicates nonvectorized loopsn = 5: indicates nonvectorized loops and prohibiting data dependence information

Intel® Parallel Debugger Extension for Microsoft Visual Studio*

The Intel® Parallel Debugger Extension focuses on the detection of parallelism-related coding issues with the use of OpenMP*. To use the Intel Parallel Debugger Extension, please build your application with the following option settings:

C++ Property Page and Field	Option	Comment
Debug, Enable Parallel Debug Checks	/debug:parallel	Generates parallel debug code instrumentations needed for the thread data sharing and reentrant call detection of the Intel® Parallel Debugger Extension.
Language, OpenMP Support	/Qopenmp	Enables the parallelizer to generate multithreaded code based on the OpenMP* directives. May require an increased stack size.

This guarantees that symbol information is available for the application and has been instrumented to permit the type of debug exceptions to be triggered so the Intel® Parallel Debugger Extension can detect data sharing events and function reentrancy. The table below lists key capabilities and options for the Intel Parallel Debugger Extension and their benefits for identifying runtime issues associated with concurrent coding models:

Feature	Description
Shared Data Event Detection	Thread data sharing read or write events will be logged, can be displayed in a dedicated thread data sharing events window, and followed back to the source line that caused them. It is also possible to choose between only logging these events and having the program execution halted if such an event occurs. This can be a powerful tool for hunting down runtime errors caused by possible data-sharing violations.
Reentrant Call Detection	The program execution will be halted if a reentrant function call occurs and the Intel® Parallel Debugger Extension will indicate the thread IDs of the threads entering the function and source location of the reentrant call. This can be a powerful tool to identify runtime problems caused by not properly protecting function calls in a multithreaded environment.
Serialize Parallel Regions	To clarify whether a runtime issue is caused by a serial algorithmic problem or has been introduced when OpenMP* directives were added, it is possible to force serial single-threaded execution of OpenMP parallel regions. The code segment of the parallel region associated with the current program counter memory location will be put into serial single-threaded execution mode.
OpenMP Windows* (Task, Task Spawn Tree, Locks, Barriers, Teams)	Displays and allows monitoring of the currently active OpenMP* threads, teams, barriers, locks, spawn trees, and taskwails in the debugged application, along with the exact current execution state of all OpenMP threads.
SSE Register Window	Provides access to MMX, SSE, SSE2, and SSE3 registers commonly used for data vectorization and single instruction multiple data (SIMD) handling. It displays vectors as rows (or columns) in a table display. Therefore, the SSE Registers Window displays expressions not as single values, but as a vector of values. It thus provides you with ways to change the representation of the data in those registers that makes their use in the debug application executable as well as the actual data mapping to the underlying hardware more transparent.

Please refer to the following articles for more information on the Intel® Parallel Debugger Extension features:
<http://software.intel.com/en-us/articles/parallel-debugger-extension/>
<http://software.intel.com/en-us/blogs/2008/12/17/debugging-heavily-threaded-parallel-code/>



For product and purchase information, visit the
Intel® Software Development Products site at:
www.intel.com/software/parallelstudio

© 2009-2010, Intel Corporation. All rights reserved. Intel, the Intel logo, Intel Atom, Centrino, Core, and Pentium are trademarks of Intel Corporation in the U.S. and other countries.

* Other names and brands may be claimed as the property of others.

