Intel® Parallel Composer

Product Brief

Intel® Parallel Composer

Build Serial and Parallel C and C++ Applications for Multicore Systems

Intel® Parallel Composer is a comprehensive set of Intel® C++ compilers, libraries, and debugging capabilities for developers bringing parallelism to their Windows*-based client applications. It integrates with Microsoft Visual Studio*, is compatible with Visual C++, and supports the way developers work, protecting IDE investments while delivering an unprecedented breadth of parallelism development capabilities, including parallel debugging. Intel Parallel Composer is a stand-alone product or can be purchased as part of Intel® Parallel Studio, which includes Intel® Parallel Inspector to analyze threading and memory errors, and Intel® Parallel Amplifier for performance analysis of parallel applications.

Intel Parallel Composer Components

- Intel C++ Compilers for 32-bit processors, a cross-compiler to create 64-bit applications on 32-bit systems, and a native 64-bit compiler
- Intel® Parallel Debugger Extension, which integrates with the Microsoft Visual Studio debugger
- Intel® Threading Building Blocks (Intel® TBB), an award winning C++ template library that abstracts threads to tasks to create reliable, portable, and scalable parallel applications. It can also be used with Visual C++.
- Intel® Integrated Performance Primitives (Intel® IPP) is an extensive library of multicore-ready, highly optimized software functions for multimedia, data processing, and communications applications. Intel IPP includes both hand-optimized primitive-level functions and high-level threaded solutions such as codecs. It can be used for both Visual C++ and .NET development.
- Sample code and a great Getting Started Guide to get you going quickly

Intel® C++ Compiler

Microsoft Visual Studio integration and compatibility

All features in Intel Parallel Studio are seamlessly integrated into Microsoft Visual Studio 2005 and 2008. Intel Parallel Composer is one of three main functional groups in Intel Parallel Studio. It includes the Intel C++ Compiler, Intel Parallel Debugger Extension, Intel Threading Building Blocks, and Intel Integrated Performance Primitives.

The compiler offers native 32-bit development, a cross-compilation environment (32-bit host to develop 64-bit applications), and native 64-bit development. You have the option of installing only the 32-bit capability, only the 64-bit capabilities, or both.

The Intel C++ Compiler and the associated Intel Parallel Debugger Extension offer C and C++ developers a number of advantages, but it is not required for the use of other components in Intel Parallel Composer or the full Intel Parallel Studio. This means you can use Intel Threading Building Blocks and Intel Integrated Performance Primitives with the Visual C++ compiler. You can also use the Intel Parallel Studio memory leak or concurrency checking capabilities on applications built with Visual C++. In short, there are plenty of reasons for you to be interested in using the full Intel Parallel Studio, including a powerful, easy-to-use, compatible Intel C++ compiler.

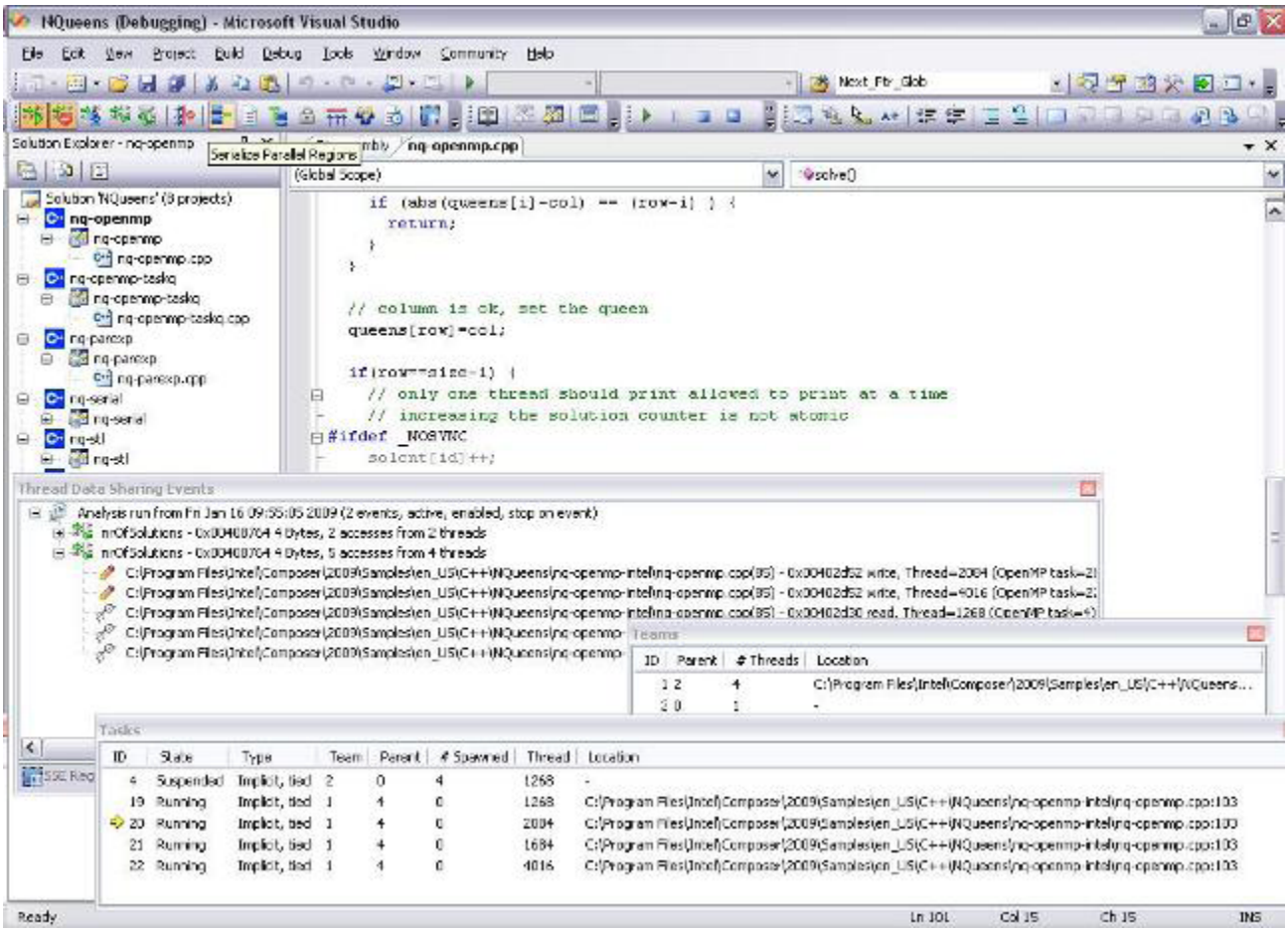

Figure 1: Intel® Parallel Composer integrates into Visual Studio*. The solution on display uses Intel C++. You can easily switch to Visual C++ from the Project menu or by right-clicking over the solution or project name.

Easy to get started and stay connected with a growing parallelism community

Intel Parallel Composer includes an easy-to-use Getting Started Guide that offers a quick tour of functionality and code samples used to convey how-to instructions. It even includes links to short videos, so there is no doubt how to use the parallelism features in Intel Parallel Composer. Users of Intel Parallel Composer found the guide to be worth the few minutes it takes to go through it. They found the sample code useful in introducing parallelism concepts and techniques, which led to productive use of the tools.

The Getting Started Guide is available from several places. For example, you can find it on our Web pages, at several points during installation, from the Visual Studio Help menu (along with in-depth documentation), and from the Intel Parallel Studio or Intel Parallel Composer tree structure available from the Windows "Start" button. There is even a prompt for it upon completion of installation. Whether you are a beginner, a parallelism pro, or somewhere in between, it's worth a few minutes to go through the Getting Started Guide.

Once you get going, you will find it useful to join the growing community of developers taking advantage of systems based on Intel multicore processors. Intel provides a dynamic forum for developers to exchange ideas, post comments and questions, and earn points to become an Intel® Black Belt Software Developer. We also provide a large and growing knowledge base presenting a variety of topics to developers interested in parallelism. Join the community today. Visit the parallel programming and multicore community at http://software.intel.com/en-us/articles/intel-parallel-studio/. From there, you can tap into all of its resources, including blogs, knowledge bases, downloads, and more. Feel free to explore, and don't forget to save it to your favorites list.

Support for lambda functions

The Intel Compiler is the first C++ compiler to implement lambda functions in support of the working draft of the next C++ standard C++0x. A lambda construct is almost the same as a function object in C++ or a function pointer in C. Together with closures, they represent a powerful concept, because they combine code with a scope. A closure is a function that can refer to and alter the values of bindings established by binding forms that textually include the function definition. In short, a lambda function, together with a closure, can be seen as syntactic sugar around function objects and function pointers that offers a convenient way to write function objects, or lambdas, right at the point of use.

The source code in Figure 2 below is an example of a function object created by a lambda expression. Tighter C++ and Intel TBB integration allows the simplification of the functor operator() concept by using lambda functions and closures to pass code as parameters.

```
void solve() {
  parallel_for(blocked_range<size_t>(0, size, 1),
  [](const blocked_range<int> &r){
    for (int i = r.begin(); i != r.end(); ++i)
    setQueen(new int[size], 0, (int)i);
    };
  });
}
```

Figure 2: Source code example of a lambda function

Simple concurrency functions

The Intel® C++ Compiler in Intel Parallel Studio offers four new keywords to help make parallel programming easier: __taskcomplete, __task, __par, and __critical. In order for your application to benefit from the parallelism made possible by these keywords, you specify the /Qopenmp compiler option and then recompile, which links in the appropriate runtime support libraries, which manage the actual degree of parallelism. These new keywords use the OpenMP 3.0* runtime library to deliver the parallelism, but free you from actually expressing it with OpenMP* pragma and directive syntax. This keeps your code more naturally written in C or C++.

The keywords mentioned above are used as statement prefixes. For example, we can parallelize the function, solve(), using __par. Assuming that there is no overlap among the arguments, the solve() function is modified with the addition of the __par keyword. With no change to the way the function is called, the computation is parallelized. An example is presented in Figure 3:

```
void solve() {
  __par for(int i=0; i<size; i++) {
  // try all positions in first row
  // create separate array for each recursion
  // started here
  setQueen(new int[size], 0, i);
  }
}
```

Figure 3: Example of __par, one of 4 simple concurrency functions, new in the Intel C++ Compiler in Intel® Parallel Studio

OpenMP 3.0

OpenMP is an industry standard for portable multithreaded application development. It is effective at fine-grain (loop-level) and large-grain (function-level) threading. OpenMP 3.0 supports both data and now task parallelism using a directives approach, which provides an easy and powerful way to convert serial applications into parallel applications, enabling potentially big performance gains from parallel execution on multicore and symmetric multiprocessor systems.

When an application that has been written and built using OpenMP is run on a system with just one processor, the results are the same as unmodified source code. Stated differently, the results are the same as those you would get from unmodified, serial-execution code. This makes it easier for you to make incremental code changes while maintaining serial consistency. Because only directives are inserted into the code, it is possible to make incremental code changes and still maintain a common code base for your software as it runs on systems that still have only one processor.

OpenMP is a single source code solution that supports multiple platforms and operating systems. There is also no need to "hard-code" the number of cores into your application because the OpenMP runtime chooses the right number for you.

OpenMP 3.0 Task Queuing

Sometimes programs with irregular patterns of dynamic data or complicated control structures, like recursion, are hard to parallelize efficiently. The work queuing model in OpenMP 3.0 allows you to exploit irregular parallelism, beyond that possible with OpenMP 2.0 or 2.5.

The task pragma specifies the environment within which the enclosed units of work (tasks) are to be executed. When a task pragma is encountered, the code inside the task block is conceptually queued into the queue associated with the task. To preserve sequential semantics, there is an implicit barrier at the completion of the task. The developer is responsible for ensuring that no dependencies exist or that dependencies are appropriately synchronized, either between the task blocks, or between code in a task block and code in the task block outside of the task blocks. An example is presented above in Figure 4.

```
#pragma omp parallel
#pragma omp single
{
  for(int i=0; i<size; i++) {
    // try all positions in first row
    // create separate array for each recursion
    // started here
#pragma omp task
    setQueen(new int[size], 0, i);
  }
}
```

Figure 4: An example of OpenMP3 3.0 task queuing

In the example in Figure 4 above, we need only one task queue. Therefore, we need to set up the queue by invoking only one thread (omp single). The setQueens calls are independent of each other's, and therefore they fit nicely into the task concept. You might want to also read about the Intel Parallel Debugger Extension, directly below, which makes it easy to inspect the state of tasks, teams, locks, barriers, or taskwaits in your OpenMP program in dedicated windows.

Intel Parallel Debugger Extension

Intel Parallel Composer includes the Intel Parallel Debugger Extension which, after installation, can be accessed through the Visual Studio Debug pull-down menu (see Figure 5 below).

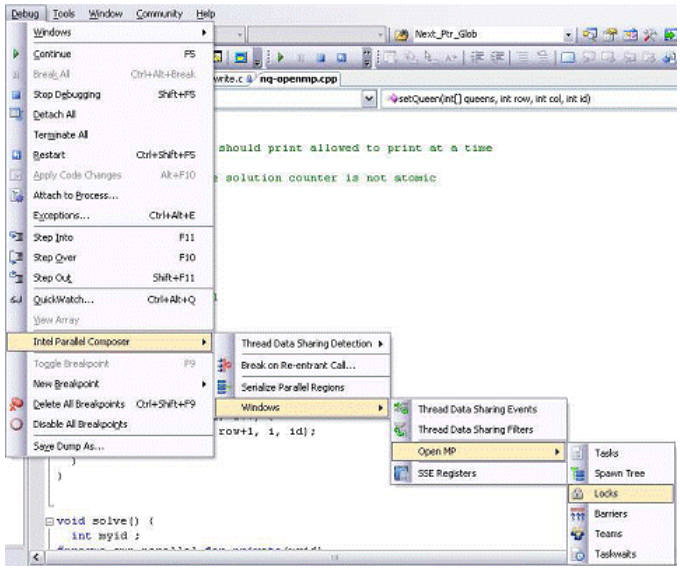

Figure 5: Intel Parallel Debugger Extension is accessible from the Debug pull-down menu in Microsoft Visual Studio

The Intel Parallel Debugger Extension provides you with additional insight and access to shared data and data dependencies in your parallel application. This facilitates faster development cycles and early detection of potential data access conflicts that can lead to serious runtime issues. After installing the Intel Parallel Composer and starting Visual Studio, you can use the Intel Parallel Debug Extension whenever your applications are taking advantage of Single Instruction Multiple Data (SIMD) execution and get additional insight into the execution flow and possible runtime conflicts if your parallelized application uses OpenMP threading.

To take advantage of the advanced features of the Intel Parallel Debugger Extension like shared data event detection, function re-entrancy detection, and OpenMP awareness including serialized execution of parallelized code, compile your code with the Intel Compiler using the /debug:parallel option for debug info instrumentation.

For more information, see the Intel Parallel Debugger Extension white paper at http://software.intel.com/en-us/articles/parallel-debugger-extension/. This paper goes into many more details and benefits the debugger extension can bring to you, and how to best take advantage of them.

If you are evaluating debugging products for your parallel applications, consider the larger Intel Parallel Studio product suite. It includes Intel Parallel Inspector, which features memory leak analysis and thread checking tools. It also includes Intel Parallel Amplifier, which provides hotspot (performance bottleneck) analysis and concurrency checking tools to debug for code correctness with added awareness of parallelized code and data. Intel Parallel Studio provides all of these capabilities, including the Intel Parallel Debugger Extensions.

Optimize embarrassingly parallel loops

Algorithms that display data parallelism with iteration independence lend themselves to loops that exhibit "embarrassingly parallel" code. Intel Parallel Composer supports three techniques to maximize the performance of such loops with minimal effort: auto-vectorization, use of Intel-optimized valarray containers, and auto-parallelization. Intel Parallel Composer can automatically detect loops that lend themselves to auto-vectorization. This includes explicit *for* loops with static or dynamic arrays, vector and valarray containers, or user-defined C++ classes with explicit loops. As a special case, implicit valarray loops can either be auto-vectorized or directed to invoke optimized Intel Integrated Performance Primitives (IPP) library primitives. Auto-vectorization and use of optimized valarray headers optimize the performance of your application to take full advantage of processors that support the Streaming SIMD Extensions.

In a moment, we'll look at how to enable Intel optimized valarray headers. But first, let's look at Figure 6, which shows an example of an explicit valarray, vector loops, and an implicit valarray loop.

```
valarray<float> vf(size), vfr(size);
vector<float> vecf(size), vecfr(size);


//log function, vector, explicit loop
for (int j = 0; j < size-1; j++) {
  vecfr[j]=log(vecf[j]);

}

//log function, valarray, explicit loop
for (int j = 0; j < size-1; j++) {
  vfr[j]=log(vf[j]);
}

//log function, valarray, implicit loop
vfr=log(vf);
```

Figure 6: The source code above shows examples of explicit valarray, vector loops, and an implicit valarray loop

To use optimized valarray headers, you need to specify the use of Intel Integrated Performance Primitives as a Build Component Selection and set a command line option. To do this, first load your project into Visual Studio and bring up the project properties pop-up window. In the "Additional Options" box, simply add "/Quse-intel-optimized-headers" and click "OK." Figure 7 presents a picture of how to do this.

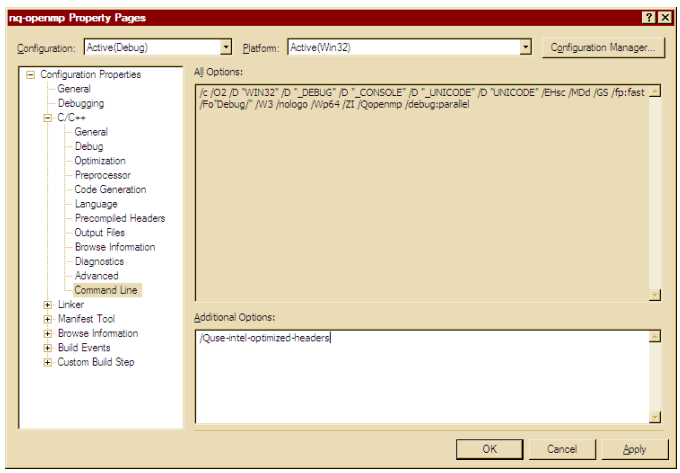

Figure 7: Adding the command to use optimized header files to a command line in Visual C++

Next, from the Project menu, open the Build Component Selection pop-up. In the box to the right of "Intel Integrated Performance Primitives," select "Common" and click "OK." Figure 8 presents a picture of this. With this done, you can rebuild your application and check it for performance and behavior as you would when you make any change to your application.

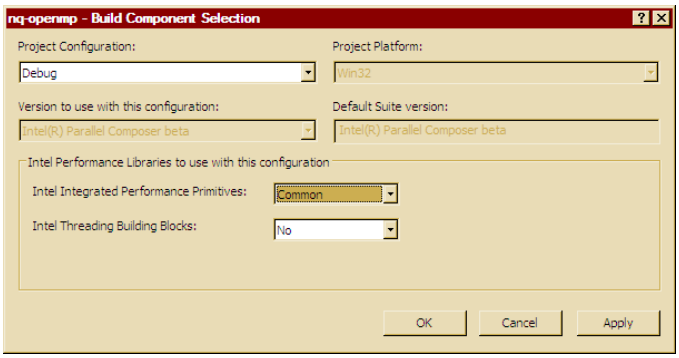

Figure 8: Directing Visual Studio to use Intel IPP

Auto-parallelization improves application performance by finding parallel loops capable of being executed safely in parallel and automatically generating multithreaded code, allowing you to take advantage of multicore processors. Automatic parallelization relieves the user from having to deal with the low-level details of iteration partitioning, data sharing, thread scheduling, and synchronizations.

Auto-parallelization complements auto-vectorization and use of optimized valarray headers, giving you optimal performance on multicore systems that support SSE. For more information on multithreaded application support, see the user guide (http://software.intel.com/en-us/intel-parallel-composer/, then click the documentation link).

Intel Threading Building Blocks

Intel Threading Building Blocks (Intel TBB) is an award-winning C++ template library that abstracts threads to tasks to create reliable, portable, and scalable parallel applications. Included in Intel Parallel Composer, Intel TBB is a standard template library (STL) that can be used with the Intel C++ Compiler or with Microsoft Visual C++.

Intel TBB solves *three key challenges* for parallel programming:

- **Productivity**: Simplifies the implementation of parallelism
- **Correctness**: Helps eliminate parallel synchronization issues
- **Maintenance**: Aids in the creation of applications ready for tomorrow, not just today

Advantages of using Intel TBB:

- **Future-proof applications**: As the number of cores (and threads) increase, application speedup increases using Intel TBB's sophisticated task scheduler
- **Portability**: Implement parallelism once to execute threaded code on multiple platforms
- **Interoperability**: Commitment to work with a variety of threading methods, hardware, and operating systems
- **Active open source community**: Intel TBB is also available in an open source version. opentbb.org is an active site with forums, blogs, code samples, and much more

Intel TBB offers comprehensive, abstracted templates, containers, and classes for parallelism. Figure 9 highlights the major functional groups within Intel TBB.

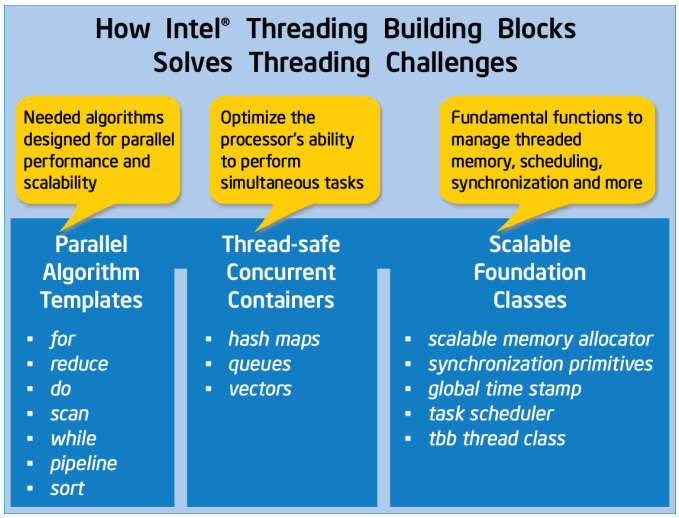

Figure 9: Major function groups within Intel® TBB

Problem	Solution
How to add parallelism easily	Intel TBB parallel_for command Straightforward replacement of for/next loops to get advantages of parallelism Load-balanced parallel execution of fixed number of independent loop iterations
Management of threads to get best scalability	Intel TBB Task Scheduler Manages thread pool and hides complexity of native threads Designed to address common performance issues of parallel programming - Oversubscription: One scheduler thread per hardware thread - High overhead: Programmer specifies tasks, not threads - Load imbalance: Work-stealing balances load
Memory allocation is a bottleneck in concurrent environment	Intel TBB provides tested, tuned, and scalable memory allocator based on per-thread memory management algorithm - As an allocator argument to STL template classes - As a replacement for malloc/realloc/free calls (C programs) - As a replacement for global new and delete operators (C++ programs)

Figure 10: Intel® TBB addresses three major parallelism issues

Intel Integrated Performance Primitives (Intel IPP)

Intel Parallel Composer includes the Intel Integrated Performance Primitives. Intel IPP is an extensive library of multicore-ready, highly optimized software functions for multimedia, data processing, and communications applications. It offers thousands of optimized functions covering frequently used fundamental algorithms in video coding, signal processing, audio coding, image processing, speech coding, JPEG coding, speech recognition, computer vision, data compression, image color conversion, cryptography, string processing/ regular expressions, and vector/matrix mathematics.

Intel IPP includes both hand-optimized primitive-level functions and high-level threaded samples, such as codecs, and can be used for both Visual C++ and .NET development. All of these functions and samples are fully thread-safe, and many are internally threaded, to help you get the most out of today's multicore processors and scale to future manycore processors.

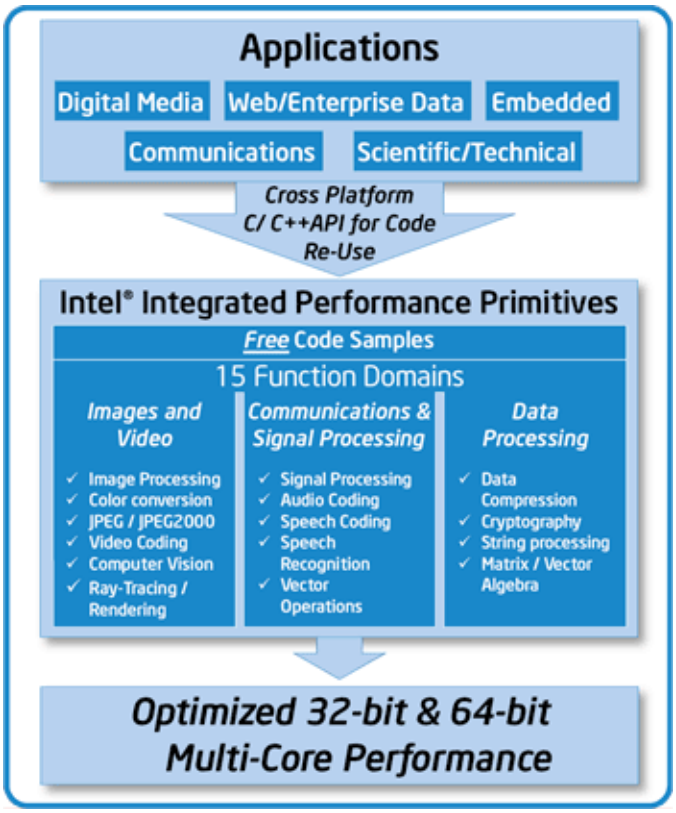

Figure 11: Intel® Integrated Performance Primitives is included in Intel® Parallel Composer, a part of Intel® Parallel Studio, and features threaded and thread-safe library functions over a wide variety of domains listed above

Intel IPP Performance

Depending on the application and workload, Intel IPP functions can perform many times faster than the equivalent compiled C code. In the image resize example below, the same operation that required 338 microseconds to execute in compiled C++ code required only 111 microseconds when Intel IPP image processing functions were used. That is a 300% performance improvement.

Figure 12: In this image resizing example (from 256x256 bits to 460x332 bits), the Intel® IPP-powered application ran in 111 msec vs. 338 msec for compiled C++ code (system configuration: Intel® Xeon, 2.9 GHz, 2 processors, 4 cores/processor, 2 threads/processor)

Using Intel IPP in Visual Studio

It's easy to add Intel IPP support to a Microsoft Visual Studio project. Intel Parallel Composer includes menus and dialogs to add Intel IPP library names and paths to a Visual Studio project. Simply click on the project name in the Solution Explorer, select the Intel Build Components Selection menu item, and use the Build Components dialog to add Intel IPP. Then just add Intel IPP code to your project including the header and functional code. You'll notice that the Build Selection dialog automatically adds the library names to the linker for IPP and adds a path to the Intel IPP libraries.

In addition to C++ projects, Intel IPP can also be used in C# projects using the included wrapper classes to support calls from C# to Intel IPP functions in the string processing, image processing, signal processing, color conversion, cryptography, data compression, JPEG, matrix, and vector math domains.

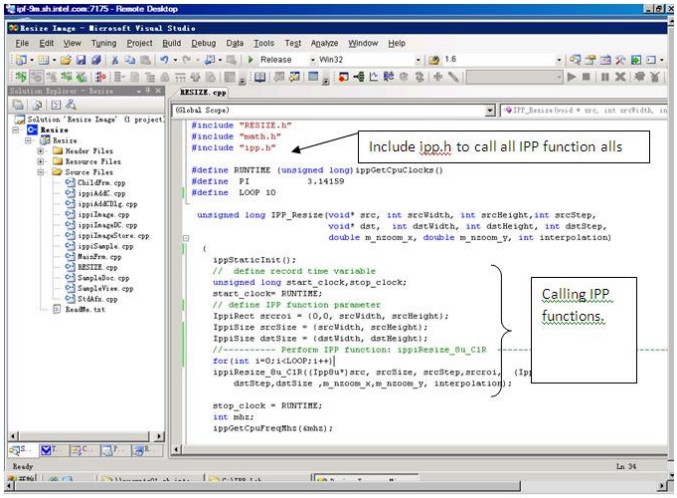

Figure 13: It's easy to incorporate Intel® IPP library calls into your Visual Studio* code

Features

- Seamlessly upgrades Microsoft* Visual Studio for C/C++ parallelism. It integrates into Visual Studio and preserves your IDE investment, while adding parallelism capabilities.
- Intel Parallel Debugger Extension integrates with the Microsoft debugger, enhancing Visual Studio to help find and address parallelism issues. Saves time in getting applications ready to be used.
- Includes simple concurrency functions, data parallel arrays, and thousands of threaded library functions, which simplify threading tasks and speed application development
- Auto-parallelization and auto-vectorization options, which simplify development and save time
- Integrated array notation, data-parallel Intel IPP functions speed audio, video, signal analysis, and other application classes
- Includes Intel TBB, the most efficient way to implement parallel applications and unleash multicore platform
- Extensive documentation, including code examples, for getting started with parallelism. Also includes a short Getting Started Guide to get you going in just a few minutes.
- Community support. You're not alone out there. Join the growing community of developers adding parallelism to their code. Draw on the experience of others and contribute your own knowledge and experience, and win prizes while doing it.

System Requirements

- Microsoft Visual Studio
- For the latest system requirements, go to: www.intel.com/software/products/systemrequirements/

Support

Intel Parallel Studio products include access to community forums and a knowledge base for all your technical support needs, including technical notes, application notes, documentation, and all product updates.

For more information, go to
http://software.intel.com/en-us/articles/intel-parallel-studio/

Beta Versions Available Now

Start adding parallelism to your applications and take advantage of the growing installed-base of multicore systems in the market today, and future-proof your applications now for the manycore systems coming soon.

Download and register for the user forums at:
www.intel.com/software/ParallelStudioBeta/

Intel® Parallel Studio

Designed for today's serial applications and tomorrow's software innovators.

Intel brings simplified parallelism to Microsoft Visual Studio* C++ developers with a complete productivity solution designed to optimize serial and new parallel applications for multicore and scale for manycore.

Intel® Parallel Studio: Create optimized serial and parallel applications with the ultimate all-in-one parallelism toolkit

Intel® Parallel Composer: Develop effective applications with a C/C++ compiler and advanced threaded libraries

Intel® Parallel Inspector: Ensure application reliability with proactive parallel memory and threading error checking

Intel® Parallel Amplifier: Quickly find bottlenecks and tune parallel applications for scalable multicore performance

0209/BLA/CMD/PDF 321554-001