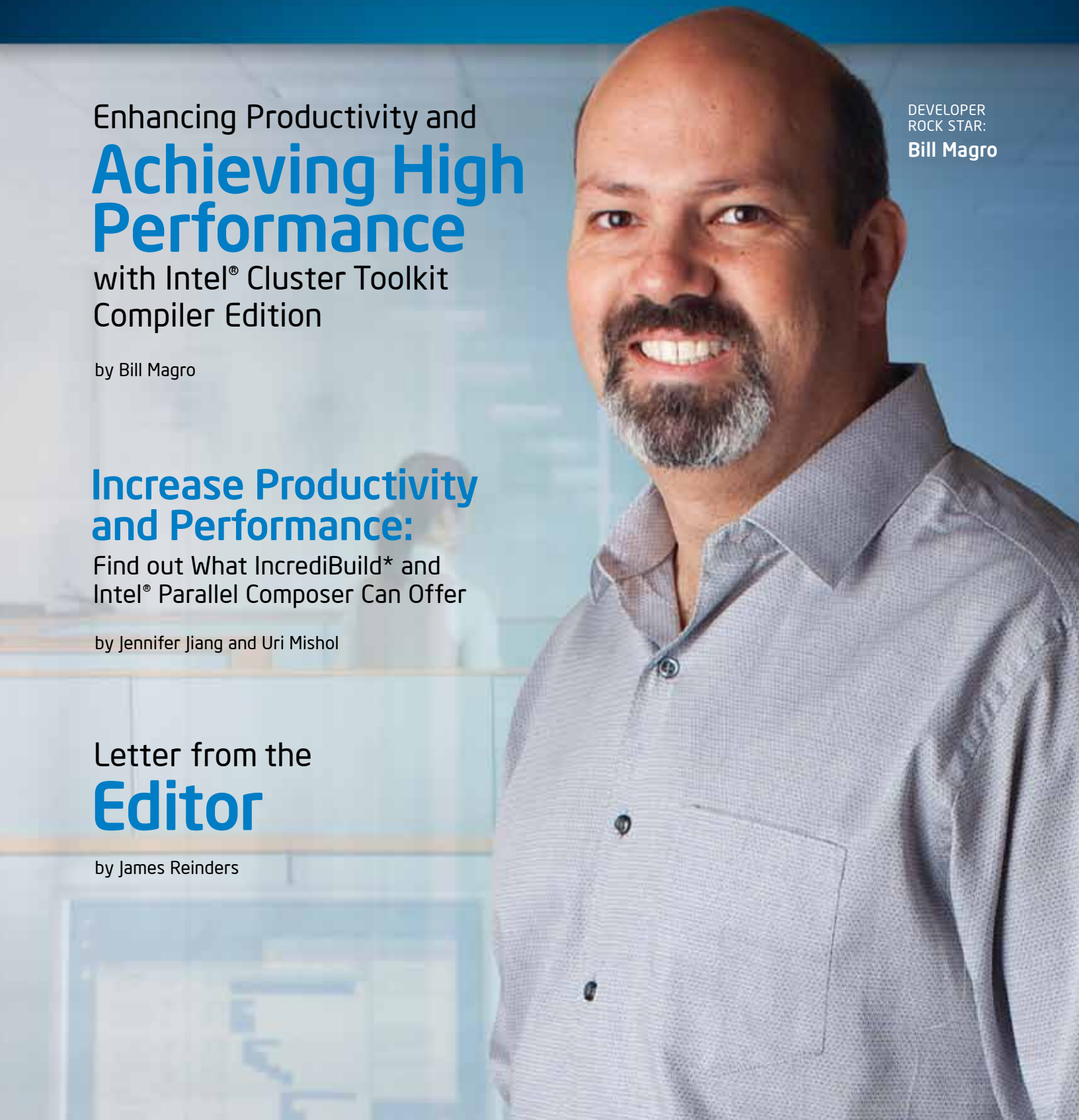
THE PARALLEL UNIVERSE

Issue 3
June 2010



DEVELOPER ROCK STAR:
Bill Magro

Enhancing Productivity and Achieving High Performance with Intel® Cluster Toolkit Compiler Edition

by Bill Magro

Increase Productivity and Performance:

Find out What IncrediBuild* and Intel® Parallel Composer Can Offer

by Jennifer Jiang and Uri Mishol

Letter from the Editor

by James Reinders

CONTENTS

Parallelism Full Steam Ahead!

James Reinders, lead evangelist and director of Intel® Software Development Products, explains how recent industry and product developments have positioned parallelism to take off full throttle.

We have had five years of multicore processors and four years of Intel® Threading Building Blocks. Time flies. Now, in 2010, we have Intel® Threading Building Blocks 3.0 (Intel® TBB) and Microsoft* Visual Studio* 2010. Later this year we'll have Intel's second generation of Intel® Parallel Studio (no charge to those who purchased the original).

I mentioned legacy, education, and tools in the prior issue as primary concerns for developers we have interviewed. Our focus on delivering solutions to these challenges continues. We have solutions for real-world existing (legacy) programs that are also the right tools for new applications. We have webinars and extensive online training (much easier to find now using the new Intel® Learning Lab). We also have great tools for developers, some of which you'll learn more about in this issue, and may more of which are explored at the Intel® Learning Lab.

I recently talked with Microsoft's Herb Sutter over breakfast before doing a webinar together (watch it on demand and join us for future events). Herb reminded me of a graph he showed me a few years ago about technology adoption. His curves showed a slow start and then a big takeoff, followed by mass acceptance and adoption. He showed a graph for object-oriented programming. Similarly, graphical interfaces (monochrome to VGA to ...) and the Internet followed his curve. It's a valid observation, even if not completely quantified. The bottom line being that mass acceptance doesn't happen overnight. But you can see signs, and you can enumerate forces.

Forces today include quad-core and eight-core processors. Four hardware threads will yield speedups for parallel programs more universally than dual-core. This is because the overhead of a tasking or threading model is more easily acceptable comparing one to four hardware threads instead of one to two. Quad-core processors are mainstream now, and eight-core processors are easy to find, too. This changes everything in terms of what developers can do and what application users will utilize.

Another force at work is good software development solutions. The Intel Threading Building Blocks project recently introduced Intel® Threading Building Blocks 3.0. This is a very mature solution with unequaled adoption worldwide. The new features in Intel TBB 3.0 really represent refinements that come primarily from dedicated users giving feedback on what would help them use Intel TBB more effectively in real-world applications (learn more about Intel TBB 3.0).

I'm a developer, not a researcher, so the stage of maturity that Intel TBB 3.0 represents is what really excites me. I'm delighted because of its here-and-now usefulness and large community of users. Adobe's adoption of Intel TBB for the Creative Suite 5* is another instance of what really makes me excited. I've been involved with Intel TBB since v1.0 in 2006, so I can count it among a handful of exceptionally successful products I've worked on in my career. Success to me comes only one way: customers who value your product. We have success with Intel TBB.

Intel Parallel Studio is another rising star. I'm pleased by Intel's decision to create the next major version and not charge current customers for the upgrade. This means that the new features, including Intel® Parallel Advisor, which are now in beta, will be available to all Intel Parallel Studio customers as a free upgrade in the fall when it is released. The new version will be known as Intel® Parallel Studio 2011, but we should have it out a few months before the end of 2010.

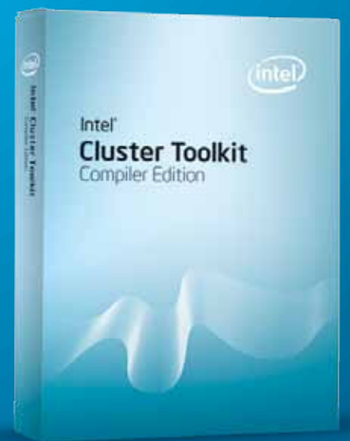
How do I gauge the success of Intel Parallel Studio? The same way I judge the success of Intel Threading Building Blocks: by what our customers are doing with it. What I see after only one year of Intel Parallel Studio suggests the same ramp of adoption that Herb, Intel TBB, and others have led me to expect. Early adopters have a job to do, and they find that Intel Parallel Studio makes that job possible both in terms of speed to solution and greater confidence in the results.

Finally, released in April, Microsoft Visual Studio 2010 represents a milestone as well with the first introduction of that product's support for parallelism. Developers working with .NET will find a task stealing option (called TPL), which is similar to Intel TBB but intended for .NET. Microsoft TPL works great for .NET parts of applications and in conjunction with Intel TBB (for C/C++) because of a layer called the Microsoft* Concurrency Runtime (ConcRT) a new addition in Microsoft Visual Studio 2010. We already have Intel TBB 3.0 and Intel's OpenMP* (in the forthcoming Intel® C++ Compiler 12.0) using ConcRT because of its ability to coordinate multiple models used in a single application to avoid oversubscription. This is another real sign of maturation of solutions for software developers.

Full steam ahead indeed. The emergence of quad-core and eight-core processors, along with the maturity of Intel TBB 3.0, Adobe's adoption of Intel TBB, the maturing of Intel Parallel Studio, and the arrival of Microsoft Visual Studio 2010, show parallelism is definitely really to go full throttle.

JAMES REINDERS
Portland, Oregon
June 2010

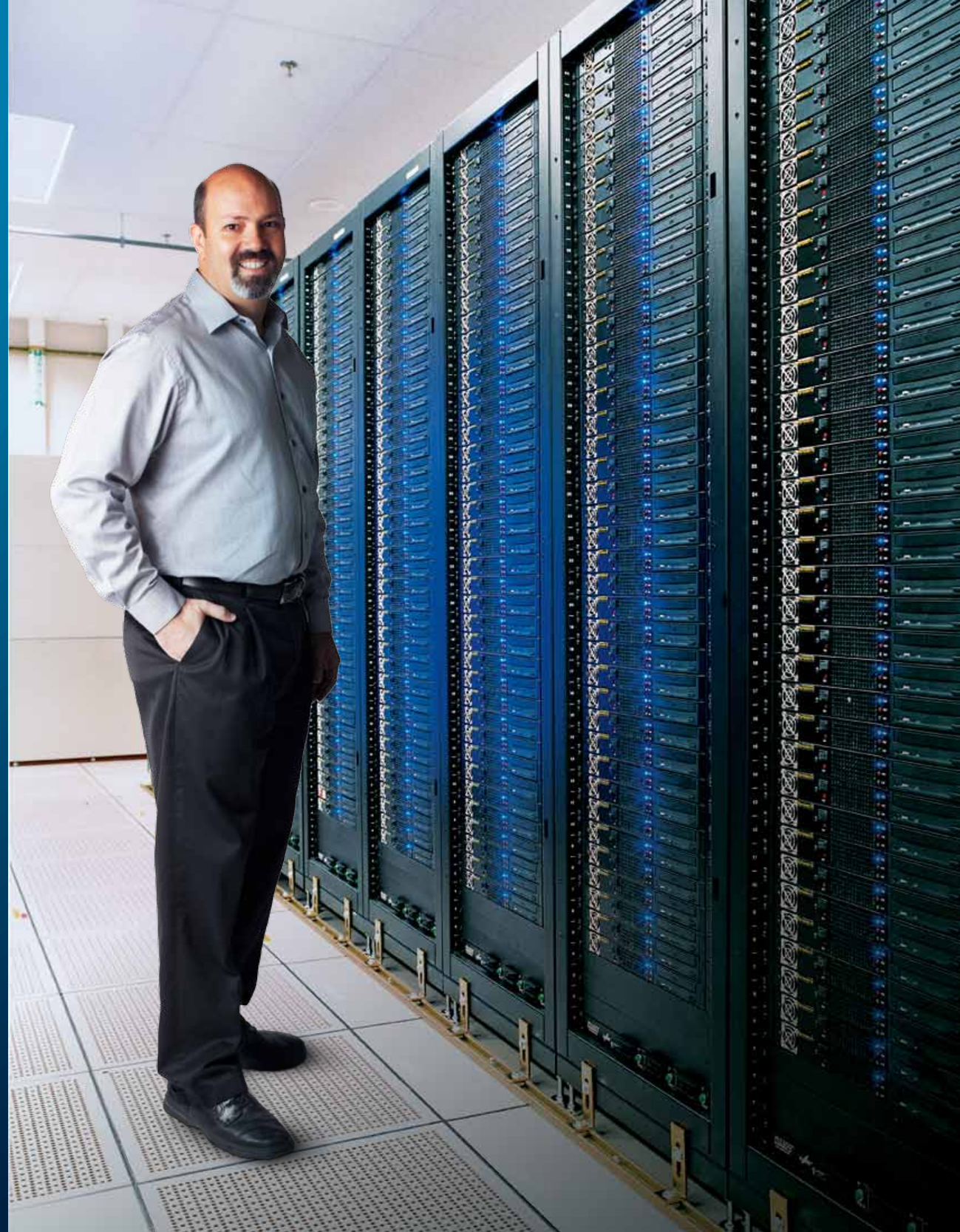
James Reinders is chief software evangelist and director of Software Development Products at Intel Corporation. His articles and books on parallelism include *Intel Threading Building Blocks: Outfitting C++ for Multicore Processor Parallelism.*



By Bill Magro

Message-passing interface applications port seamlessly from dual-core desktops to multithousand-server clusters, a key advantage of distributed memory parallelism.

Enhancing Productivity and Achieving High Performance with Intel® Cluster Toolkit Compiler Edition



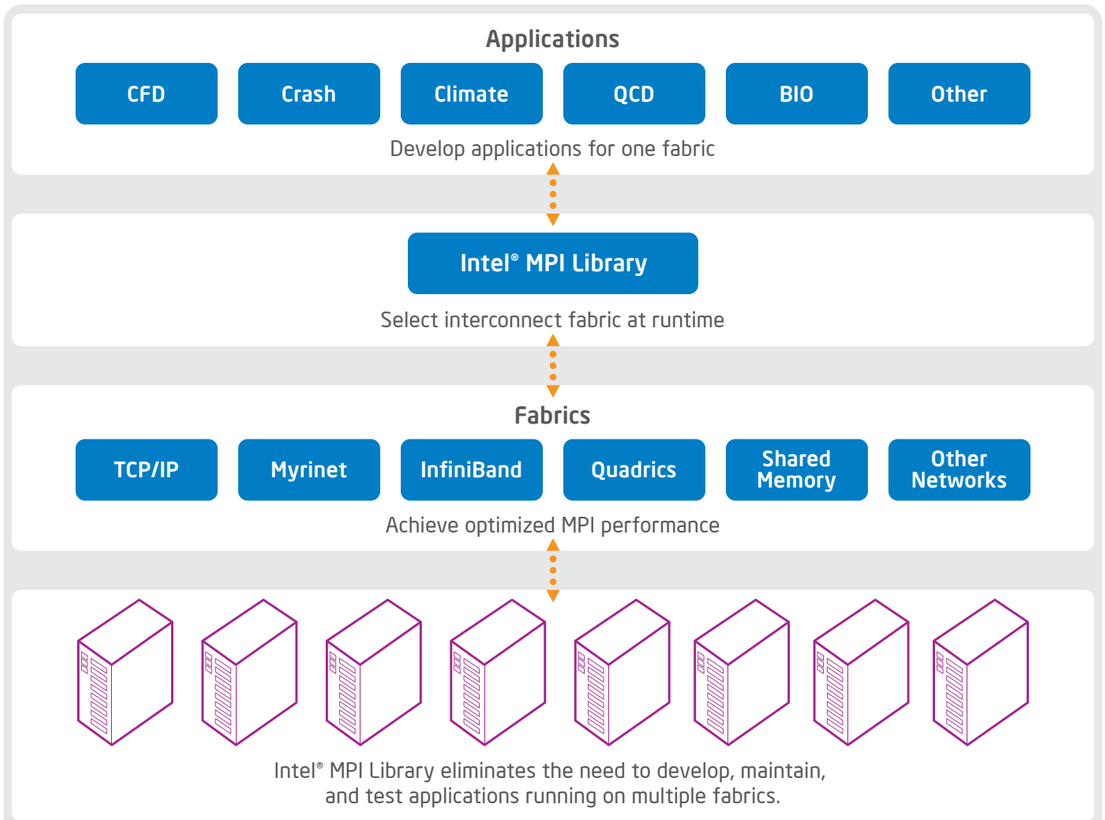


Figure 1: Intel MPI's unique interconnect independent architecture allows a single application binary to efficiently run over a wide range of networking fabrics.

Clusters of networked servers are the most popular form of high-performance computers today. Developers of cluster applications most commonly use the message-passing interface, or MPI, to implement parallelism. MPI is a mature and well-established industry standard for implementing distributed memory parallelism. MPI allows anywhere from a few to tens of thousands of processes to exchange information and work together to collectively solve the world's hardest problems.

The Intel® Cluster Toolkit Compiler Edition is the premier toolkit for developers writing MPI applications. The suite of products includes the essential tools for MPI developers.

The power of MPI has also reached the workstation, where MPI applications run well and are frequently used. That highlights a key advantage of distributed memory parallelism over the shared memory techniques you'll most often read about: MPI applications port seamlessly from dual-core desktops to multithousand-server clusters. The Intel® Cluster Toolkit Compiler Edition is the premier toolkit for developers writing MPI applications. The suite of products includes the essential tools for MPI developers: C++ and Fortran compilers, performance libraries, analysis tools and benchmarks, and, of course, a high-performance MPI implementation. The newly released toolkit version 4.0 brings a host of new usability, productivity, and performance features.

Intel® MPI is a good place to start. Version 4.0 is the biggest step forward in the product since its launch. But before we explore its advances, let's take a moment to revisit its origins. Intel MPI was created to bring the plug-and-play simplicity of desktop operating systems to the world of high-performance computing (HPC). On the desktop, we take for granted that we can freely select applications that will work with a wide range of printers and network interfaces.

But in the world of high-performance computing, high-performance networks—critical to achieving good scaling on clusters—typically arrived with a custom implementation of MPI. As a result, software vendors were typically burdened with creating and validating a separate application version for each flavor of network or "interconnect." This approach was expensive, and it left Ethernet as the least-common denominator. Beyond those who developed and used their own applications, high-performance MPI applications were limited, and innovative network vendors found it hard to penetrate the commercial market.

Intel MPI addressed this issue via an interconnect-independent architecture (**Figure 1**). By moving from an approach that supported one or more specific interconnects to one focused on a small number of stable binary interfaces—sockets, direct access programming libraries (DAPLs), and shared memory—Intel MPI decoupled application development and innovation from network development and innovation. This simple but powerful approach has allowed software vendors to write a single application binary that is forward and backward compatible with a wide range of interconnects. Fewer application versions means lower development and validation costs, while improved compatibility through standard interfaces brings lower support costs.

At the same time, interconnect vendors now have immediate access to a wide range of commercial applications simply by implementing a driver for one of Intel MPI's supported interfaces. Today, a variety of commercial MPI applications utilize a wide range of advanced interconnection networks via Intel MPI.

Initially targeted at commercial software developers, Intel MPI's architecture focused on compatibility, usability, and performance in the sweet spot of commercial software: two- to 256-core parallelism. Despite this focus, the product has proven popular with MPI developers targeting 1,024 cores and above. As a result, Intel MPI 4.0 introduces a new architecture designed to scale to 10,000 cores and beyond.

You might wonder what prevents the previous architecture from scaling to these levels. To start with, you need to know that to reach even 1,000 cores, an advanced interconnect is typically essential. Usually, one uses an interconnect based on remote direct memory access, or RDMA. InfiniBand* is the most common such interconnect in HPC. To achieve performance on RDMA networks, Intel MPI takes a familiar tack: It trades memory consumption for performance. Small message performance is critical to scaling MPI applications, and Intel MPI optimizes these transfers by creating and using pinned memory buffers. This avoids paying the repeated (and high) cost of pinning and unpinning a memory region for a single transfer.

It's an effective approach, but these buffers are needed for every communication target—and the memory consumption really adds up when using thousands of processes. Intel MPI was already smart about managing memory—buffers were only created for active targets—but version 4.0 goes further. It introduces a connectionless protocol that avoids memory registration. By using the RDMA network's unreliable datagrams and moving the reliability protocol into MPI itself, Intel MPI can now send small messages to an arbitrary number of endpoints, all from a single set of memory buffers.

As a result, local memory usage no longer grows with the number of endpoints, and MPI jobs can scale into the thousands—or even tens of thousands—of processes. The default connection-oriented approach still delivers the highest performance for smaller jobs, but those needing extreme scalability can enable the connectionless approach by setting I_MPI_DAPL_UD=enable in the environment. Intel MPI 4.0 provides binary compatibility with previous versions, so existing binaries can take advantage of these new capabilities without recompiling or relinking.

THE PARALLEL UNIVERSE

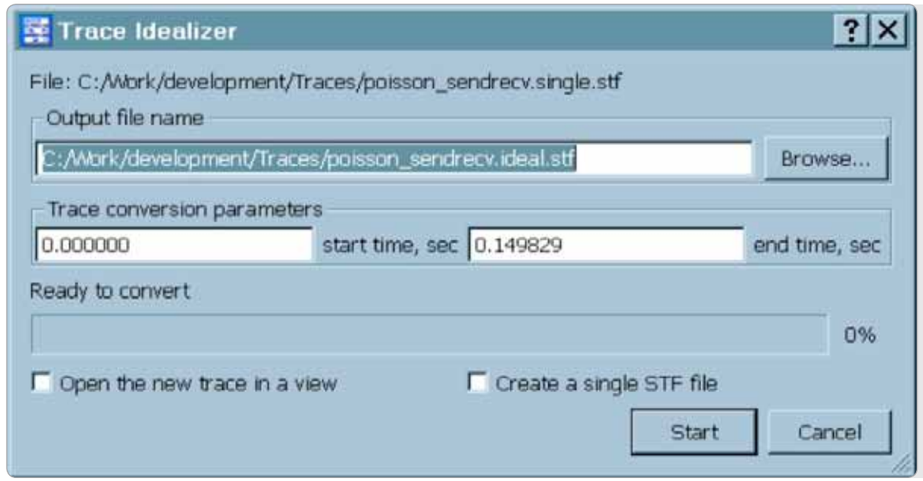


Figure 2: This dialog box makes it easy to create an idealized message trace.

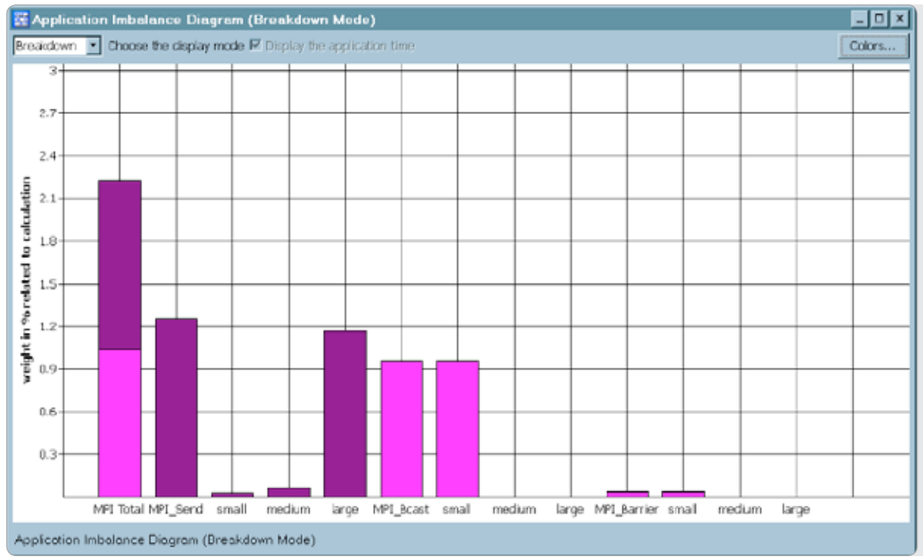


Figure 3: The application imbalance view helps you see the portion of time spent in messaging network transfers and application load imbalance. The former will decrease with faster networks, while the latter requires application tuning.

Intel® MPI 4.0 is now capable of efficiently running over interconnects that more directly support MPI semantics.

As noted, the focus is usually on shared memory programming models, such as threading and tasking. Since MPI is a distributed memory model, it might be a surprise that its shared memory performance inside a single system is rather important. It is especially important when MPI applications are deployed on workstations. Intel MPI's new architecture significantly improves messaging speed over shared memory. As an example, on a modern Intel® Xeon® platform, Intel MPI achieves about twice the messaging performance of a popular open-source MPI on messages up to 16kB in size. For larger messages, the advantage is still more than 1.5x.

In addition to cluster applications that also run on workstations, some developers choose MPI to create parallel applications specifically for desktop and workstations, finding that the high-level data decomposition it encourages is a straightforward way to design for scalability.

Another key component of the Intel Cluster Toolkit is its MPI analysis tool, the Intel® Trace Analyzer and Collector. There aren't a lot of things more frustrating in software development than investing a lot of time and energy recoding for performance, only to find the performance doesn't change. Intel provides a number of tools to help you find the critical path and ensure local performance gains will translate into application gains.

A key question is often whether the performance limits observed are due to the hardware or the software. Usually, one doesn't have access to higher performance hardware, so one focuses on the software. In the case of MPI programs, the limiting hardware is often the interconnect, but one also wonders whether the software is itself limiting the scalability. Intel Trace Analyzer and Collector 8.0 provides tools you need to easily visualize the messaging patterns in your application and identify potential hotspots.

However, it's still natural to ask, "Is my application's scalability limited more by my code or by the network?" To help answer this question, Intel Trace Analyzer and Collector 8.0 adds an "ideal interconnect simulator." An "ideal" interconnect is defined, in this case, as one that instantaneously transports any amount of data to a ready receiver. By simulating the effects of an ideal interconnect, one can quickly separate messaging overheads due to transfer time from those originating in the application itself.

Using the feature is pretty straightforward. To get started, a real message trace is collected using the Intel® Trace Collector. With this trace loaded in the Intel Trace Analyzer, select "Advanced > Idealization" and a dialog box appears (**Figure 2**). From here, you can generate a transformed trace file, representing the run on an ideal interconnect. With the new trace file loaded in the tool, it can be compared with the original run. The performance increases achieved from the ideal interconnect are immediately apparent; any remaining inefficiencies are in the code.

This feature is also helpful for people wanting to estimate the gains, if any, they'd see from moving up from, say, 1Gb Ethernet to an InfiniBand interconnect. Intel Trace Analyzer includes a new display—the application imbalance diagram—that breaks down the remaining overheads, helping you quickly identify the MPI function calls and specific message sizes that need attention (**Figure3**).

While the ideal interconnect is one useful trace transform, you can write and apply your own through the custom plug-in framework—another new feature in 8.0. Do you want to understand how much further your code will scale if the latency of the interconnect is halved for small messages? Write a simple transform function, and Intel Trace Analyzer will generate and display the updated trace file.

We've focused on RDMA interconnects here—and for good reason. Standard support for RDMA in InfiniBand and iWarp* interconnects has been a boon for HPC users. It has allowed an unprecedented number of systems vendors and even do-it-yourself computer centers to build world-class supercomputer clusters from affordable and readily available components. One look at the growth of clusters in the world's Top 500 computer systems illustrates the impact of RDMA and Intel® Architecture servers. Despite RDMA's popularity in HPC, its semantics are not a perfect match to those of MPI. For example, every MPI message carries a tag that is matched at the destination, while RDMA lacks such a notion. Such differences mean every MPI implementation must do a certain amount of "impedance matching" to an RDMA fabric. And the extra code brings overheads.

Intel MPI 4.0 is now capable of efficiently running over interconnects that more directly support MPI semantics. Myricom's Myrinet*, with its MX interface, and Qlogic's InfiniBand adapter, with its PSM interface, are good examples. In the past, Intel MPI supported these interconnects only via the DAPL RDMA interface, since no other common binary interface existed. Note that the overheads of impedance matching to these interconnects occurs twice—first from fabric to DAPL and then from DAPL to MPI.

In version 4.0, Intel MPI introduces a new interface, the "Tag-matching Interface," or TMI, tailored to this class of interconnects. The TMI interface represents a substantially thinner—and more efficient—interface for fabrics that natively support semantics that closely approximate those of MPI. The result is simpler drivers and better performance. For example, the latency of small messages over Qlogic's PSM* decreased by a factor of three from Intel MPI 3.2.1 to Intel MPI 4.0.

The Intel Cluster Toolkit Compiler Edition is Intel's premier tool suite for developers of MPI-based cluster applications. We've only scratched the surface here regarding what's new in version 4.0. Check out the release notes to discover many more new features to enhance your productivity as you achieve high performance. □

THE PARALLEL UNIVERSE

Increase Productivity and Performance:

Find out What IncrediBuild* and Intel® Parallel Composer Can Offer

By Jennifer Jiang
and Uri Mishol

Software companies use many software development methodologies, but none eliminate the need for building, testing, and tuning individual components or the whole application.

Is your application taking a long time to build?

Why is it not running as fast as you would like?

IncrediBuild*, a distributed computing tool, can reduce the application build time significantly by distributing the different parts of the build process or compilation across computers in a local network. As a result, the build can run up to 20 times faster.

Intel® Parallel Composer, an Intel® C++ Compiler with performance libraries, as well as the Intel® C++ Compiler Professional Edition, brings much needed runtime performance with its advanced optimization techniques, including auto-vectorization, auto-parallelism, and high-performance optimization.

Build-Time Comparison

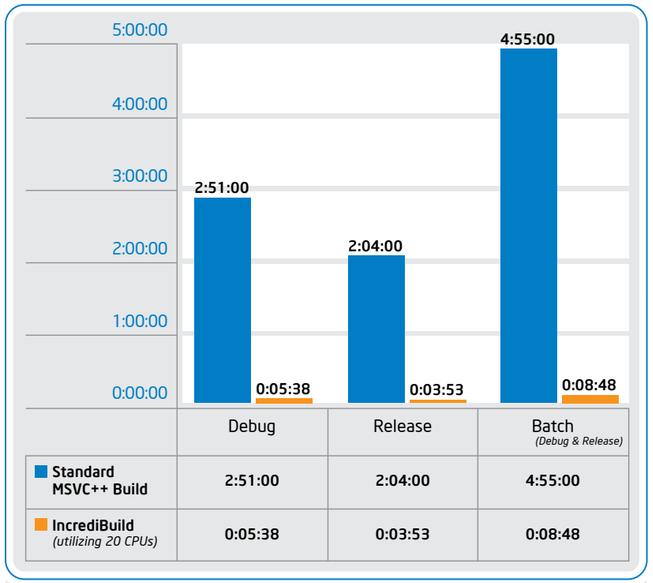


Figure 1: Possible build-time speedup using IncrediBuild

Application performance tuning is an art. It can be a very time-consuming process involving a great deal of testing, data reconstruction, etc.

Introduction

There are many software development methodologies being used by software companies for application development, but none eliminate the need for building, testing, and tuning individual components or the whole application. For some applications, building time may not be an issue. For other applications, it can take hours or dozens of hours to perform a complete build. In such cases, software engineers may be less motivated to change source code and improve readability or maintainability just to avoid the pain of a full rebuild. If faced with this situation, it may be time to consider how you can speed up your application build time so that more time can be spent on designing, coding, debugging, and testing. This is where IncrediBuild* can help.

Application performance tuning is an art. It can be a very time-consuming process involving a great deal of testing, data reconstruction, etc. Other times it may require large-scale surgery, such as a redesign, for better performance. But sometimes, it might only require a few small code changes or a data structure change, or just a change of the compiler. Intel® Parallel Composer can help with the state-of-the-art C++ compiler and a number of fine-tuned performance libraries.

Solutions

Reducing Compile Time: IncrediBuild* by Xoreax Ltd.

IncrediBuild is a distributed computing tool for Windows* software developers. It utilizes a technology called "Grid Computing," a form of distributed computing, in which different parts of one or more processes are executed in parallel across computers connected to a network. This way, all idle CPU cycles on the network can be put to use. As a result of the distributed parallel execution, the process is considerably accelerated. The above chart shows the actual reduction in compilation time for a Microsoft* Visual Studio* C++ project based on the number of agents used (**Figure 1**).

How IncrediBuild Works

IncrediBuild consists of two major components: the Coordinator (running on a server) and Agents (running on all clients). IncrediBuild Agents are client components responsible both for initiating jobs as well as for participating in executing jobs initiated by other Agents. The most basic functionality of an IncrediBuild Agent is to act as a "Helper," executing tasks initiated by other Agents. Throughout the distributed job execution, the Coordinator assigns remote Agents to the executing jobs and balances the jobs among Agents. Relevant input or output files are transferred on demand between remote Agents and the local file system.

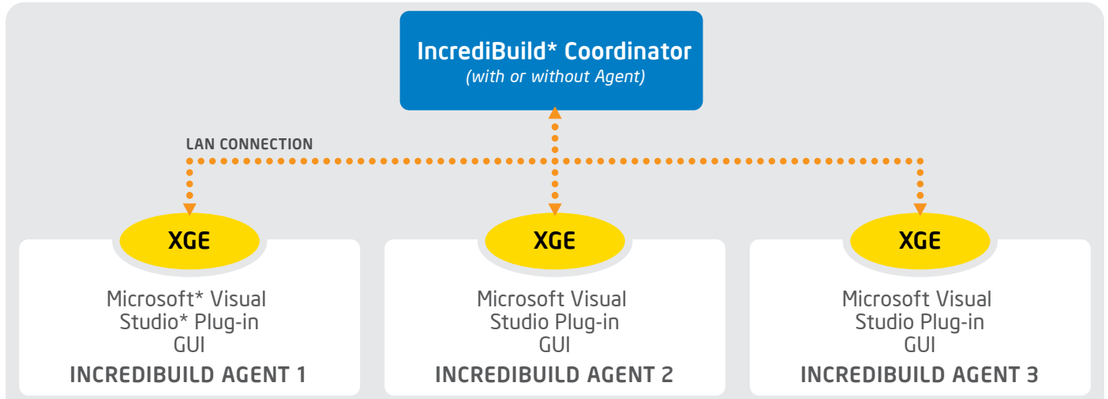


Figure 2: LAN connected computers with IncrediBuild Coordinator and/or Agent installed

Code and Debug with Intel® Parallel Composer

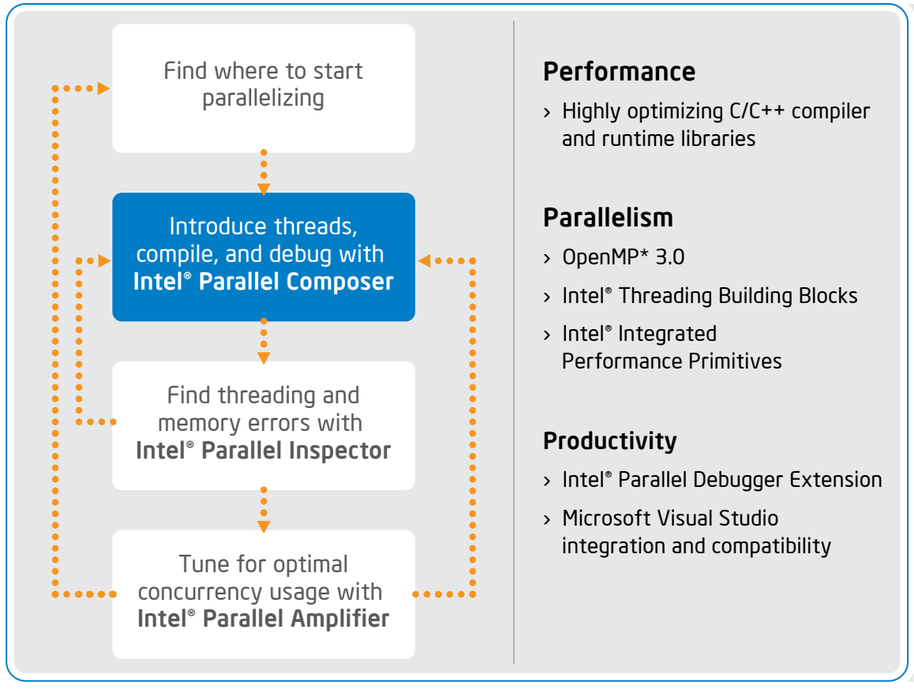


Figure 3: Intel Parallel Composer features

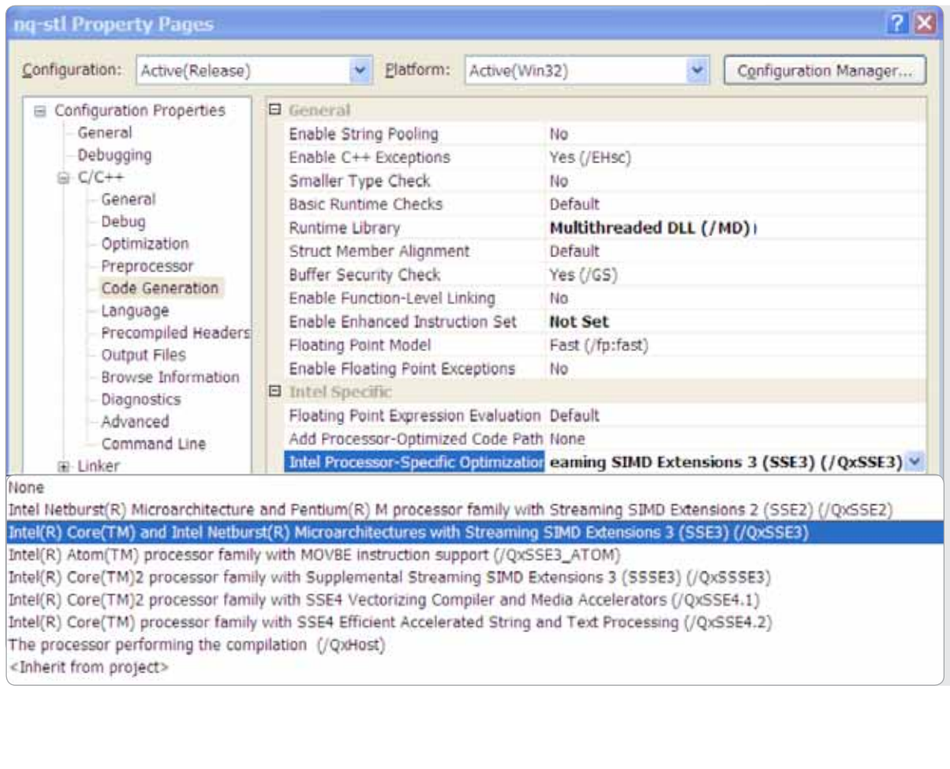


Figure 4: Project property field for /Qx option

The Agents utilize the Virtual Environment (effectively the "brain" of IncrediBuild) of the Xoreax Grid Engine* (XGE*) to ensure that a task runs on remote machines exactly as if it were being executed on the computer that initiated the job—regardless of the remote machine's file system, installation base, and environment. The XGE dynamically adjusts its operation according to the participating machines' status and availability, handling various disconnect and recovery scenarios (**Figure 2**).

Using IncrediBuild

IncrediBuild is easy to install and use. Installation should always begin by installing the Coordinator (the server component), followed by the Agents (client machines), which are typically developer workstations or relatively idle machines that can contribute processing power to running builds. During Agent installation, the setup program will automatically test the connection to Coordinator and set the appropriate settings such as port# and so on. The entire installation process takes only minutes.

A system tray icon will be installed on both Agent and Coordinator after installations; it offers a convenient way to manage everything from a single place.

Version updates can be automatically pushed by the Coordinator to all Agents to further simplify ongoing maintenance.

IncrediBuild is integrated into the following Visual Studio IDEs:**

- Microsoft* Visual Studio* 2008 standard or above
- Visual* Studio* 2005 standard or above
- Visual* Studio* .NET 2003 standard or above
- Visual* Studio* 6.0 standard or above

***Note:** Visual Studio* 2010 is to be supported in Q2 of 2010.*

```
Example: auto-v.cpp

    1 void work( float* a, float *b, float *c, int MAX)
    2 {
    3     for (int I=0;I<=MAX;I++)
    4        c[I]=a[I]+b[I];
    5 }
```

Figure 5

Auto-vectorization

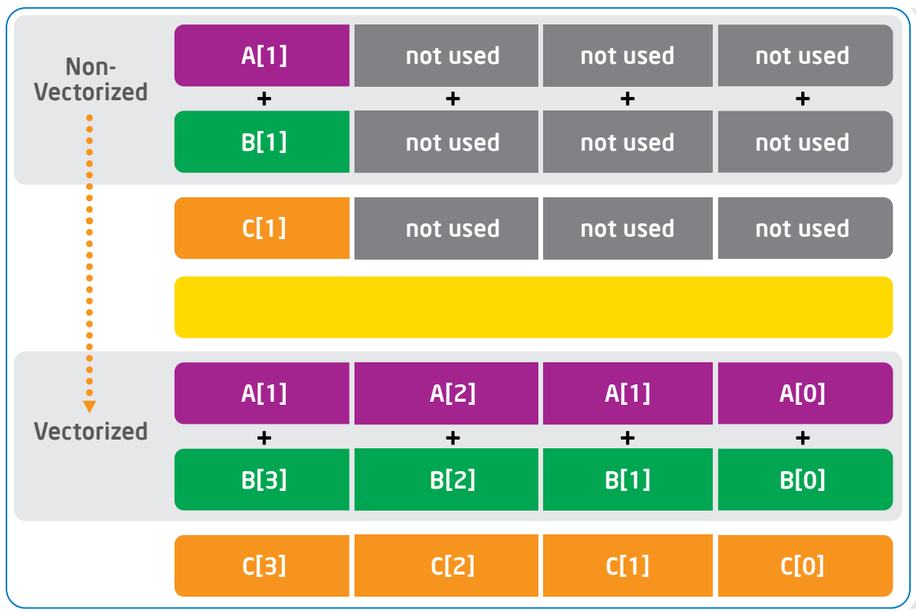


Figure 6: Auto-vectorization translates the single-element operation to four element operation per iteration, hence the better performance.

Both the IncrediBuild Agent and Coordinator modules can only run on Windows OS*.

After having installed an Agent, you will find everything you need in the new "IncrediBuild" menu.

If you are a command line user, IncrediBuild supports command line builds from various scripting languages like Perl, DOS command files, and so on. There's also an extension to IncrediBuild called XGE Interfaces, which allows you to use XGE to speed up other processes such as make-based builds, scripts, data builds, and more.

IncrediBuild also supports the following additional compilers within the IDE or from the command line:

- Intel® C++ Compiler 7.x ~ 11.x Professional Edition for Windows
- Intel® Parallel Composer
- GNU* C++ Compiler variants

Required Software

Both the IncrediBuild Agent and Coordinator modules can only run on Windows OS. However, it is not required that every Agent system have Visual Studio installed; only the Agent that initiates the "Build" job requires Visual Studio, just like any software developer's system. Agents that contribute processing power to builds initiated by other Agents do not require any software apart from the IncrediBuild Agent as the entire build environment is virtualized via XGE's virtualization mechanism.

The latest IncrediBuild version 3.51 supports the following Windows OS versions:

- Windows NT*, Windows* 2000, Windows XP*, Windows Server* 2003, Windows Server 2008*, Vista*, and Windows* 7
- Please visit the following pages for more information about IncrediBuild:
  - FAQs about IncrediBuild
  - What others have to say about this tool

Improving Performance: Intel Parallel Composer

Intel Parallel Composer is one of the four components included in Intel Parallel Studio. The Intel Parallel Composer component contains the Intel C++ Compiler for Windows with OpenMP* 3.0 support, the Intel® Threading Building Blocks (Intel® TBB), and the Intel® Integrated Performance Primitives (Intel® IPP). And best of all, the optimizations provided by the Intel® C++ Compiler, such as auto-vectorization, auto-parallelism, inter-procedural optimization, and high-performance optimization, do not require any code changes or very minimal code changes. You only need to pass the right compiler options; the Intel C++ Compiler will do the magic and generate the binary with the best performance for the processors you would like to target (**Figure 3**).

So, just by switching to Intel C++ Compiler, your application can benefit from those optimizations for better performance. How could those optimizations bring better performance?

Auto-vectorization: controlled by options /Qx[SSE2| SSE3| SSE4.1| SSE4.2| SSE3_ATOM| AVX], /arch:[SSE|SSE2|SSE3|SSSE3|SSE4.1], or /Qax{SSE2, SSE3, SSSE3, SSE4.1, SSE4.2, AVX} under the project property "C/C++ -> Code Generation" fields "Intel Processor-Specific Optimization" or "Add Processor-Optimized Code Path" (**Figure 4**).

Auto-vectorization enables processor-specific optimizations targeting any processor with Intel® Streaming SIMD Extensions (Intel® SSEx) support, including processors from Intel and AMD*. It exploits instruction-level parallelism (ILP) for loops that do not have loop dependencies and no data aliasing. By using one SIMD instruction, it can process two, four, eight, or up to 16 data elements in parallel, depending on the data type, speeding up the application runtime performance (**Figure 5**).

The Intel C++ Compiler will use the Intel SSEx instructions to accomplish the operation illustrated in figure 6, resulting in a nice performance improvement for the application (**Figure 6**).

There are some requirements in order for loops to be vectorized. Option "/Qvec-report3" is very helpful in determining why a critical loop is not vectorized. There are pragmas provided to help the compiler better understand the loops for vectorization, for example:

- #pragma ivdep
- #pragma loop_count min(N), max(M), avg(K)

Auto-vectorization optimization can only be done for the inner-most loops. Consult the reference links at the end of this article for more information about auto-vectorization.

Auto-parallelization: controlled by the option /Qparallel under project property "C/C++ -> Optimization" field "Parallelization."

Auto-parallelization is another optimization for loops. The auto-parallelizer analyzes the code and dataflow to determine if the loops are good worksharing candidates and partitions the data for threaded code. It then translates serial portions of the program into equivalent multithreaded code. Auto-parallelized applications can usually run faster on multiprocessor and multicore systems.

Auto-parallelization optimization is usually applied for the outer loops.

High-level Optimization (HLO): controlled by /O3 under project property "C/C++ -> Optimization" field "Optimization."

HLO exploits the source code constructs (loops and arrays) and does more aggressive optimizations including prefetching, scalar replacement, cache blocking, and loop- and memory-access transformations (loop unroll and jam, loop distribution, data prefetching, etc.). HLO is recommended for loop-intensive applications that perform substantial floating-point calculations or process large data sets.

Interprocedural Optimization: controlled by /Qip and /Qipo under project property "C/C++ -> Optimization" field "Interprocedural Optimization."

/Qip is the interprocedural optimization within one compilation unit.

/Qipo is the interprocedural optimization among multiple files. This is also called "whole-program optimization."

When /Qipo is used, the compiler has the knowledge of the whole program, including all global variables, functions, parameters, etc. this enables it to do a better job for all the possible optimizations, including:

- Inlining
- Constant propagation
- Alias analysis
- Dead code elimination
- C++ class hierarchy analysis
- Indirect call conversion

Interprocedural optimization can be very beneficial to application performance, but because of the aggressive inlining, the application binary size may increase significantly.

The auto-parallelizer analyzes the code and dataflow to determine if the loops are good worksharing candidates and partitions the data for threaded code.

Figure 7: LAN-connected systems used for demo

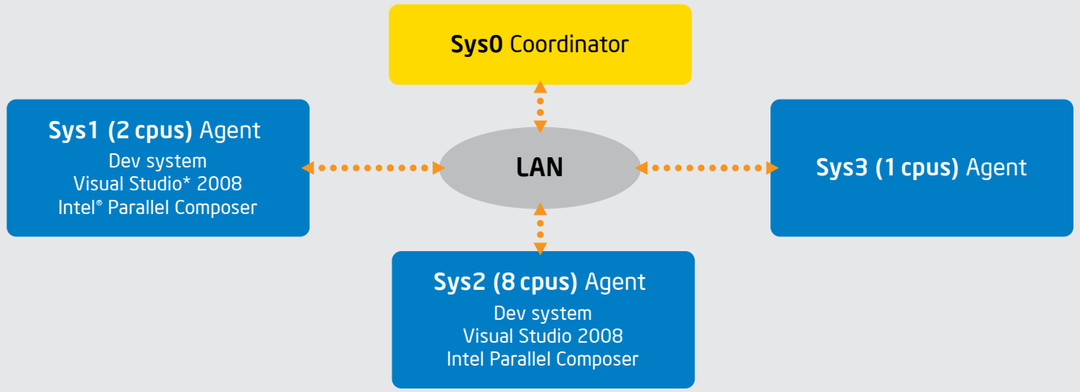
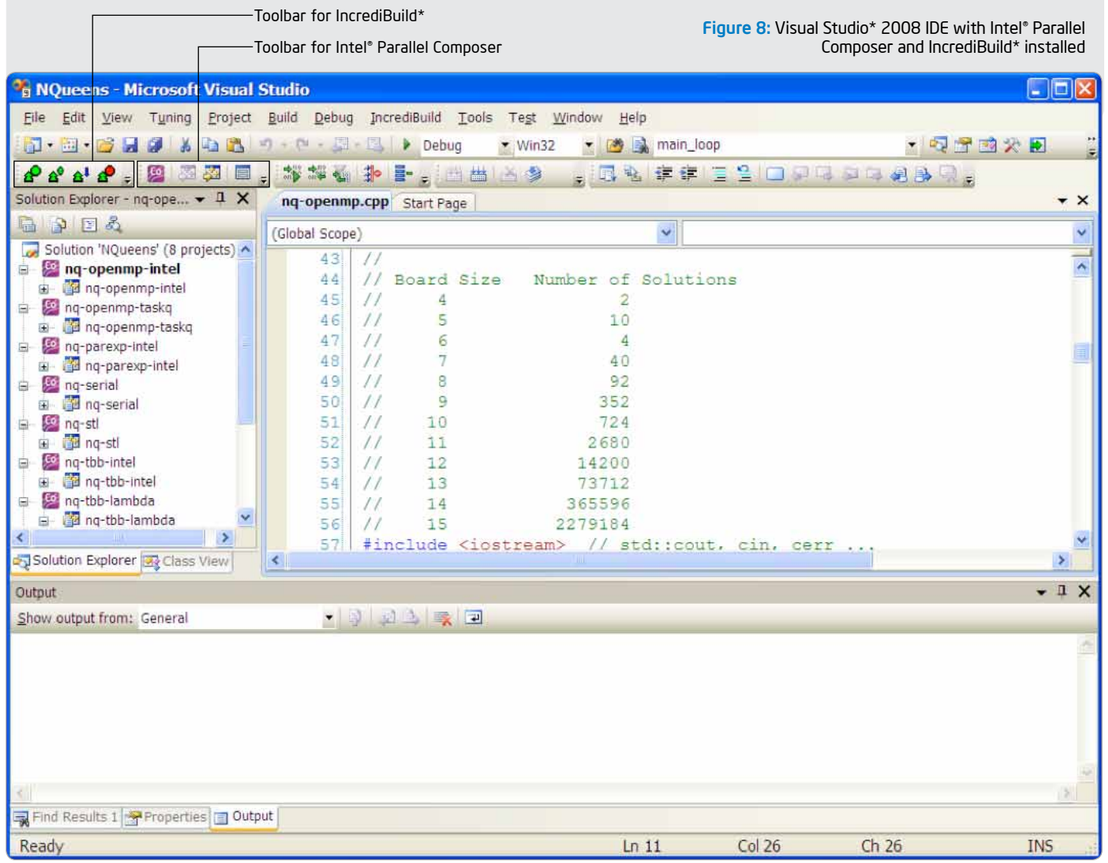


Figure 8: Visual Studio* 2008 IDE with Intel® Parallel Composer and IncrediBuild* installed



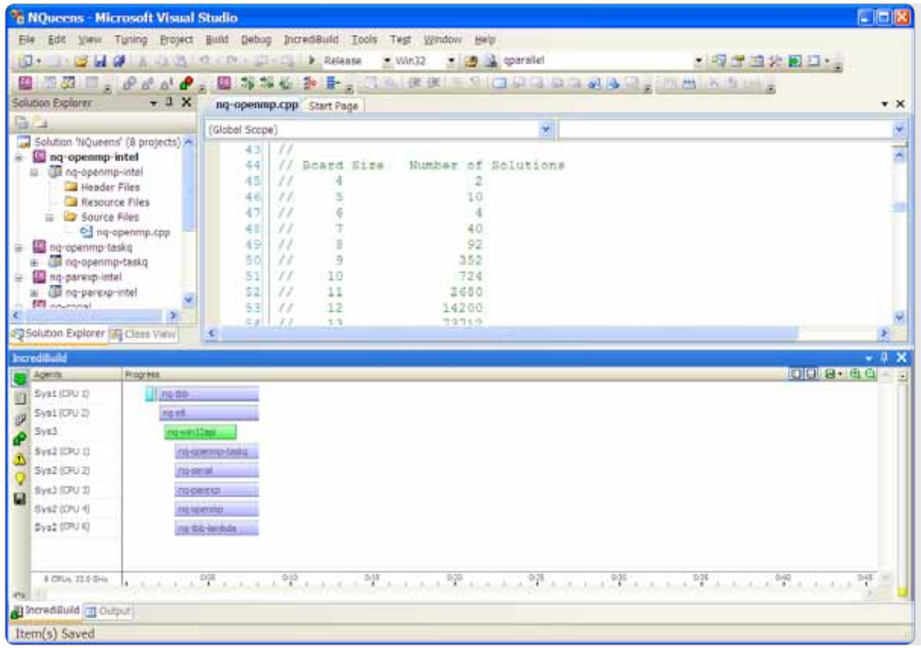


Figure 9: Rebuilding solution in progress

Compatibility with Microsoft* Visual* C++ 2005 or 2008:

Intel Parallel Composer is fully compatible with Microsoft Visual C++ 2005 and 2008. What does this mean?

It means that you can continue to build everything with Microsoft Visual C++, but only build the performance-critical code (files or projects) with Intel Parallel Composer. In other words, you can mix and match.

It is necessary to tune the performance by trying out different compiler options. But with IncrediBuild saving you time on your builds, you can take more time to concentrate on tuning and trying out different compiler optimization options for best performance.

Trying It Out

Let's get started with IncrediBuild 3.51 and Intel® Parallel Composer update 5.

We will use four computers that are connected through LAN: Sys0, Sys1, Sys2, and Sys3

- Sys1 and Sys2 are development machines with Microsoft* Visual Studio 2008 standard (or above) installed.
- Sys0 and Sys3 are used for nondevelopment without Microsoft Visual Studio 2008.

Installing the Tools

First, download the fully functional 30-day trial version of IncrediBuild. Follow the instructions to install the Coordinator and Agents. The assumption is that you have installed the Coordinator on Sys0, and the Agents on Sys1, Sys2, and Sys3 (note that it's also possible to install an Agent on the same machine running the Coordinator).

Next, download and install the evaluation of Intel Parallel Composer on Sys1 and Sys2.

You should now have a farm of machines as illustrated in **Figure 7**.

Examining Microsoft Visual Studio 2008 on Sys1

Launch Microsoft Visual Studio 2008 from Sys1 and open your solution. As an example, we'll use the NQueen sample that comes with Intel Parallel Composer (located in the "[Program files]\ Intel\ Parallel Studio\Composer\Samples\en_US\C++\Nqueens\" folder). You should see the screen illustrated in **Figure 8**.

Projects shown with the Intel Parallel Composer icon have been set to use the Intel C++ Compiler. To reset back to use the Microsoft Visual C++ Compiler, click the third icon on the Intel Parallel Composer's toolbar.

Building the NQueen Sample Solution from Sys1

Let's use IncrediBuild to build the solution so we can optimize the compilation time. If we choose to, we can always build using Microsoft Visual Studio's normal build method.

To build using IncrediBuild, click on the first icon on the IncrediBuild toolbar. You should see all the available systems building the program for you as illustrated in **Figure 9**.

Explanation of Figure 9:

- Sys1 is the agent who initiated the build job.
- Three Agents (machines) are helping the build job.
- Each CPU is building files from a separate project in this test.
- A total of eight projects are being rebuilt.
- Each project is rebuilt using a dedicated CPU: two CPUs from Sys1, five CPUs from Sys2, one CPU from Sys3.
- Almost eight projects are rebuilding at the same time.

Once the build has finished, you will see the screen illustrated in **Figure 10**.

Now, let's build the whole solution with Microsoft Visual Studio; Note that the "Build Time" for all eight projects is as follows:

1. >Build Time: 0:00:10
2. >Build Time: 0:00:11
3. >Build Time: 0:00:06
4. >Build Time: 0:00:04
5. >Build Time: 0:00:06
6. >Build Time: 0:00:06
7. >Build Time: 0:00:05
8. >Build Time: 0:00:06

The total time used is: 54 seconds, 2.5x slower than building the solution with IncrediBuild. One thing to note here is that build improvement will be substantially more impressive with larger projects that have more source files; the NQueen sample is a rather small solution, hence the modest 2.5x improvement.

Testing NQueen Performance

The test is conducted on a laptop. You will need the following:

- Intel® Core™ 2 Duo T7300 @2.00GHz
- DDR2 2GB RAM
- Windows XP* SP3
- Visual Studio* 2008 SP1
- Intel® Parallel Composer Update5

BLOG highlights

Testing for Scalability in Four Easy Steps

BY STEPHEN BLAIR-CHAPPELL

The other day I wrote a parallel program that worked very well on my two-core laptop. The question is how well would it scale? When I run the same program on a four- or eight-core machine will it go proportionally faster?

Not many of us have an eight- or 16-core machine sitting under our desk, so testing programs for scalability sounds out of reach for most of us. Intel has come to the rescue by making available for public use a multicore lab that is accessed through the Intel® Parallel Universe Portal.

In a nutshell, to test a program's scalability all that has to be done is log on to the portal, upload your executable as a Zip file, fill in the command-line options, and wait for the results to be available. The jobs you submit are queued, so time taken to get the results will depend on how busy the portal is. When I did my tests I had to wait about 40 minutes for the results. There are more details in this FAQ.

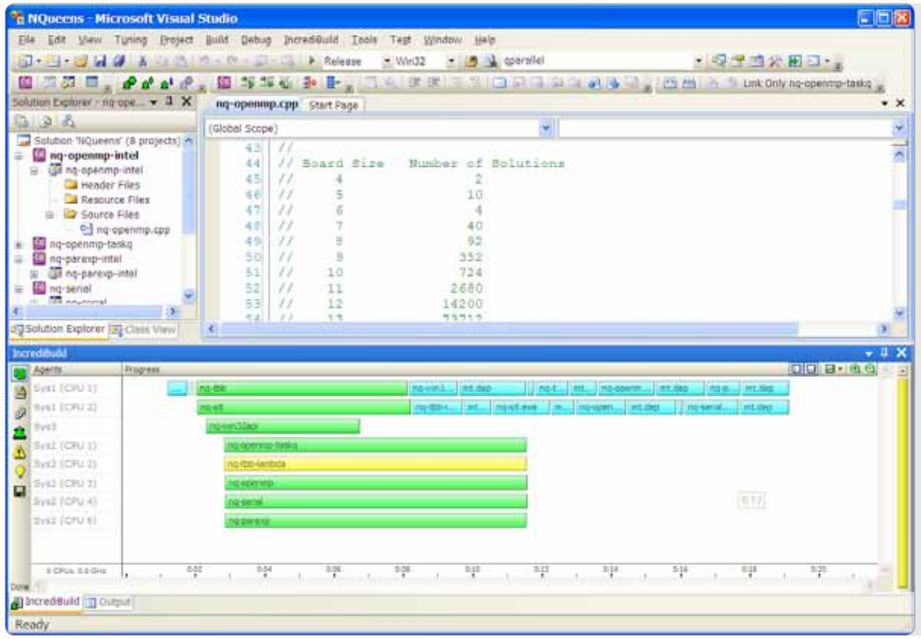


Figure 10: Finished rebuilding solution: time bar shows less than 20 sec used

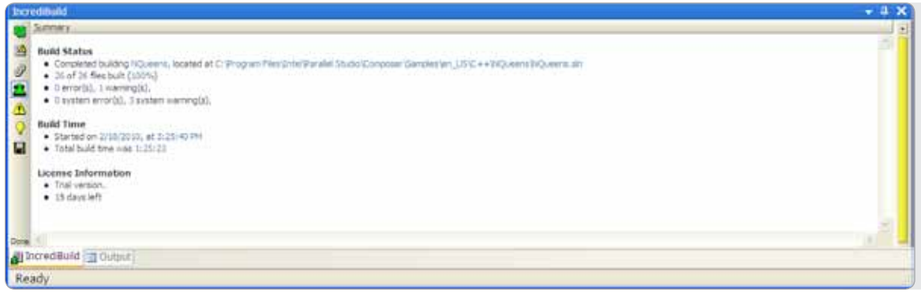
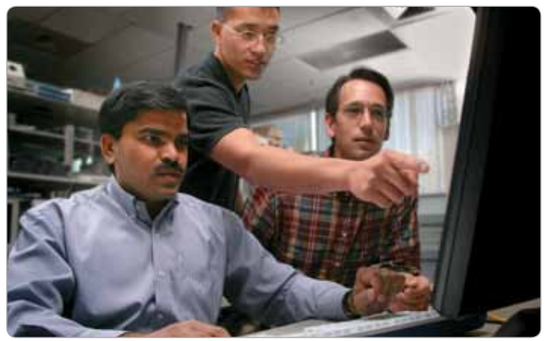


Figure 11: IncrediBuild* summary page: It shows how many files built, total time used, etc.



One thing to note here is that build improvement will be substantially more impressive with larger projects that have more source files.

Only two projects—nq-serial and nq-stl—are used from the NQueen solution, because the other five projects can only be built with Intel Parallel Composer.

Preparation:

- Start with default "Release" configuration.
- Set the "Command Arguments" property to "12" under the project property [Configuration Properties -> Debugging] so it is measurable.
- When building with Microsoft Visual C++, set the following optimizations: /O2 /Ob2 /GL /arch:SSE2
  - Optimization: Maximize Speed (/O2)
  - Inline Function Expansion: Any Suitable (/Ob2)
  - Whole Program Optimization: Use Link Time Code Generation (/GL)
  - Enable Enhanced Instruction Set: SSE2
- When building with Intel Parallel Composer, set following optimizations: /O3 /Qipo /QxSSE3
  - Optimization: Maximize Speed plus High-Level Optimization (/O3)
  - Interprocedural Optimization: Multi-file (/Qipo)
  - Intel Processor-Specific Optimization: /QxSSE3

NQueen Test Result

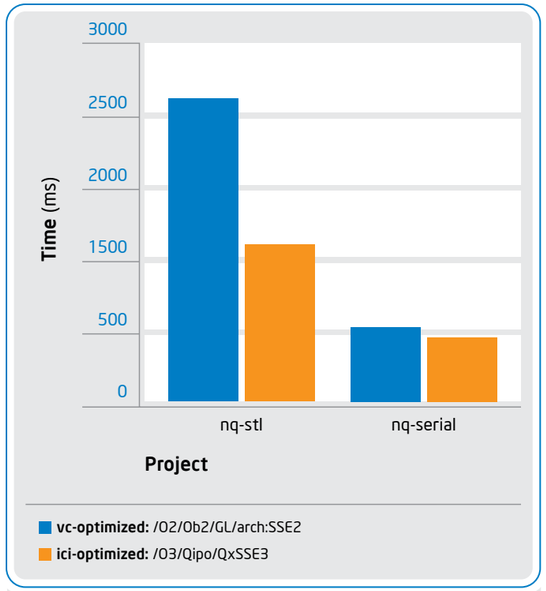


Figure 12: Test Results

The NQueen sample is a rather small solution, hence the modest 2.5x improvement.

Test results:

See **Figure 12** for the results. Note the 1.5x speedup for the nq-stl project and the 1.12x speedup for the nq-serial project.

Summary

We have shown you how IncrediBuild can help your application build time and how Intel Parallel Composer can improve your application's runtime performance. By simply combining those two tools you will get the best of both worlds: quicker compile time, and faster application performance. □

Reference Material

Learn more about Intel Parallel Composer and or Intel Parallel Studio:

Intel® Parallel Studio Homepage

Learn more about IncrediBuild:

IncrediBuild Support Center

IncrediBuild Knowledgebase Center

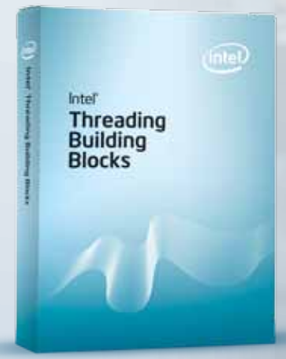
Article: "Distributed compilation with IncrediBuild* and Intel® C++ Compiler on Windows*"

Download a free trial of the tools:

IncrediBuild Download Center

Intel® Software Evaluation Center

Optimizations for MSC.Software SimXpert* Using Intel® Threading Building Blocks



By Kathy Carver, Mark Lubin, and Bonnie Aona

To address increasing customer model sizes and align with the multicore processor roadmaps for hardware vendors, MSC.Software engaged with Intel to thread SimXpert.

Intel® Software College provided training for a group of MSC.Software* engineers on threading for multiprocessor architectures and Intel® threading tools (Intel® Thread Checker, Intel® Thread Profiler, and Intel® Threading Building Blocks (Intel® TBB)). A multiphased, incremental threading approach was defined for the project.

For Phase One, MSC.Software identified 72 engineering operations in the post-processing portion of SimXpert that are responsible for the calculation of various engineering quantities (e.g., von Mises, Principal, Tresca, and Maximum Shear stresses). Intel prototyped the engineering operations and investigated both Intel® Threading Building Blocks (Intel® TBB) and OpenMP* for threading implementation. Intel TBB was selected as the best method due to its compatibility with all supported platforms. Its performance was also slightly faster than OpenMP.

For Phase Two, code responsible for producing graphical primitives was threaded, which improved performance for fringe plots. This article discusses the details of these threading implementation phases, the results achieved, and the plans for additional threading for SimXpert in future phases.

Background/Workloads Measured

Once the finite element model has been analyzed, the results can be accessed by SimXpert for post-processing. It was the Post-processing Component (PPC) of SimXpert that Intel and MSC.Software targeted for threading. This "module" allows the expert analyst to do the following:

- View selected results in a variety of ways, such as fringe, deformation, contour, vector, and tensor plots
- Identify problems
- Redesign areas of a structure

Performance for both threading phases was measured for fringe plots using large simulation models provided by MSC.Software customers. These models represent typical use cases from customers in the aerospace, automotive, and general manufacturing industries. The numerical and graphical loading that occurs is due to several critical factors:

- Free faces **(Figure 1)** are the internal and external faces of the model's finite elements where a fringe plot is rendered.
- The clustering of the finite element IDs for the elements whose free faces are being rendered directly affects the resulting data retrieval time.
- The dimensionality of the data (i.e., scalar, vector, tensor data type) directly affects the number of data values that are retrieved for post-processing.
- Also playing a role is the complexity of the engineering derivation applied to the initial analysis data to transform it from either a vector or tensor data type to a scalar data type for fringe plot rendering.

Graphical Primitives Are Created for the Free Faces of the Element

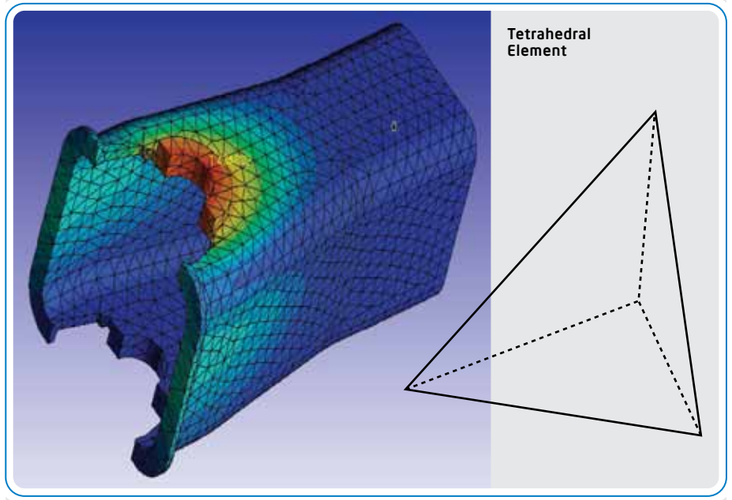


Figure 1: Free face rendering on the model's finite elements

For more information regarding performance and optimization choices in Intel software products, visit http://software.intel.com/en-us/articles/optimization-notice.

THE PARALLEL UNIVERSE

Threading SimXpert: Phase One

The initial targets for threading SimXpert were 72 engineering calculations in the PPC portion of SimXpert. Transformations were required in the original serial code before it could be parallelized with tbb::parallel_for (**Figures 2 and 3**).

After the transformations were completed, tbb::parallel_for was integrated into the application. MSC.Software relied heavily on other threading tools, such as Intel® Thread Checker and Intel® Thread Profiler, to ensure correctness and optimum performance. This code represented only 7.4 percent of the total runtime for SimXpert, but threading resulted in an average 4.9 percent improvement in overall performance. **Table 1** shows the scaling that was achieved on a 2S 3.0GHz Intel® Xeon® processor 5100 series platform/8GB with Red Hat* Linux* 4 update 3.

```
for (size_t i=0; i<Size;++i) {
    deriveFunc(ptr_inArray,ptr_outArray);
    ptr_inArray += inStride);
    ptr_outArray += outStride);
}
```

Figure 2: Original serial code

```
for (size_t i=0;i<Size; ++i) {
    deriveFunc(ptr_inArray[k* inStride],
               ptr_outArray[k * outStride]);
}
```

Figure 3: Transformation to make arrays random access containers

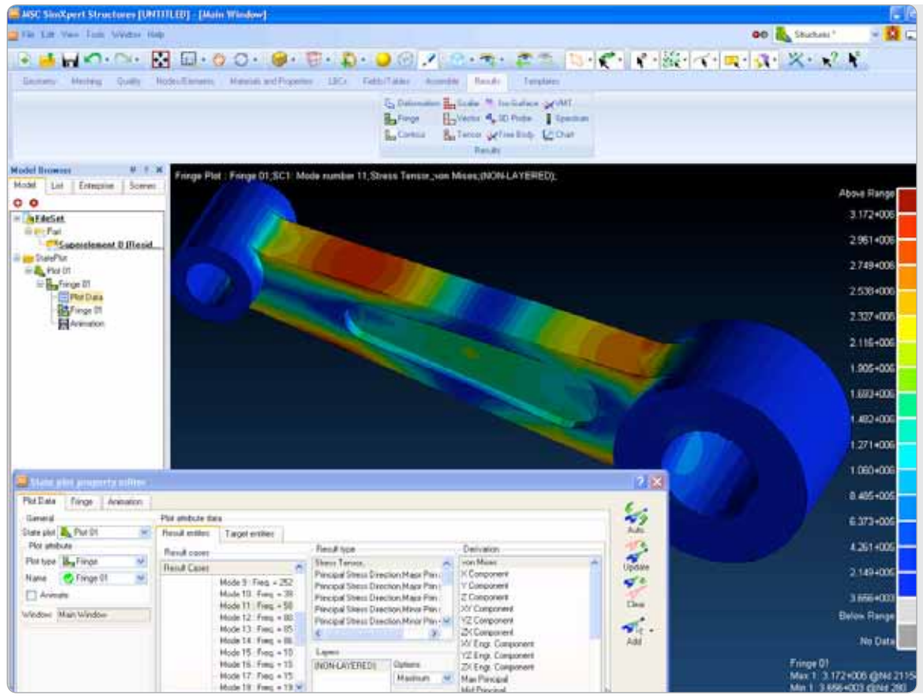


Figure 4: Fringe plot of von Mises stress

Plot	File name/ Entity count	Serial time (sec)	Parallel time (sec)	Speedup factor (Serial Time/ Parallel Time)	Serial process time (sec)	Parallel process time (sec)	Percent process speedup (s-p)/s	Percent time spent in numeric operations
Fringe - Stress, Max Princ Avg Meth=Avg/Derive, Extrap Meth=Avg	xx0kst0.xdb/624924	0.765	0.196	3.903	10.22	9.65	5.579	7.48
Fringe - Stress, Mid Princ Avg Meth=Avg/Derive, Extrap Meth=Avg	xx0kst0.xdb/624924	0.763	0.195	3.904	10.209	9.635	5.623	7.47
Fringe - Stress, Min Princ Avg Meth=Avg/Derive, Extrap Meth=Avg	xx0kst0.xdb/624924	0.762	0.197	3.873	10.208	9.636	5.604	7.46
Fringe - Stress, Tresca Avg Meth=Avg/Derive, Extrap Meth=Avg	xx0kst0.xdb/624924	0.767	0.196	3.905	10.228	9.675	5.410	7.50
NFringe - Stress, Max Princ Avg Meth=Avg/ Derive, Extrap Meth=Avg	xx0ust0.xdb/605288	0.696	0.180	3.874	9.573	9.152	4.401	7.27
Fringe - Stress, Mid Princ Avg Meth=Avg/Derive, Extrap Meth=Avg	xx0ust0.xdb/605288	0.691	0.181	3.820	9.553	9.110	4.641	7.24
Fringe - Stress, Min Princ Avg Meth=Avg/Derive, Extrap Meth=Avg	xx0ust0.xdb/605288	0.693	0.179	3.879	9.556	9.114	4.626	7.25
Fringe - Stress, Tresca Avg Meth=Avg/Derive, Extrap Meth=Avg	xx0ust0.xdb/605288	0.693	0.178	3.886	9.584	9.105	4.998	7.23
Fringe - Stress, Max Shear Avg Meth=Avg/ Derive, Extrap Meth=Avg	xx0ust0.xdb/605288	0.695	0.180	3.861	9.554	9.099	4.766	7.27
Fringe - Stress, Max Princ Avg Meth=Avg/Derive, Extrap Meth=Avg	xx0jst0.xdb/2394421	2.883	0.731	3.942	39.068	37.007	5.275	7.38
Fringe - Stress, Mid Princ Avg Meth=Avg/Derive, Extrap Meth=Avg	xx0jst0.xdb/2394421	2.888	0.730	3.956	39.090	36.945	5.486	7.39
Fringe - Stress, Min Princ Avg Meth=Avg/Derive, Extrap Meth=Avg	xx0jst0.xdb/2394421	2.880	0.730	3.947	39.086	36.816	5.808	7.37
Fringe - Stress, Tresca Avg Meth=Avg/Derive, Extrap Meth=Avg	xx0jst0.xdb/2394421	2.874	0.730	3.937	37.996	36.833	3.061	7.56
Fringe - Stress, Max Shear Avg Meth=Avg/ Derive, Extrap Meth=Avg	xx0jst0.xdb/2394421	2.894	0.732	3.952	39.433	38.277	2.932	7.34
Average				3.90			4.872	7.37
Minimum				3.82			2.932	7.23
Maximum				3.96			5.808	7.56

Table 1: Summary for plots where serial time in numeric operations was greater than 0.5 seconds

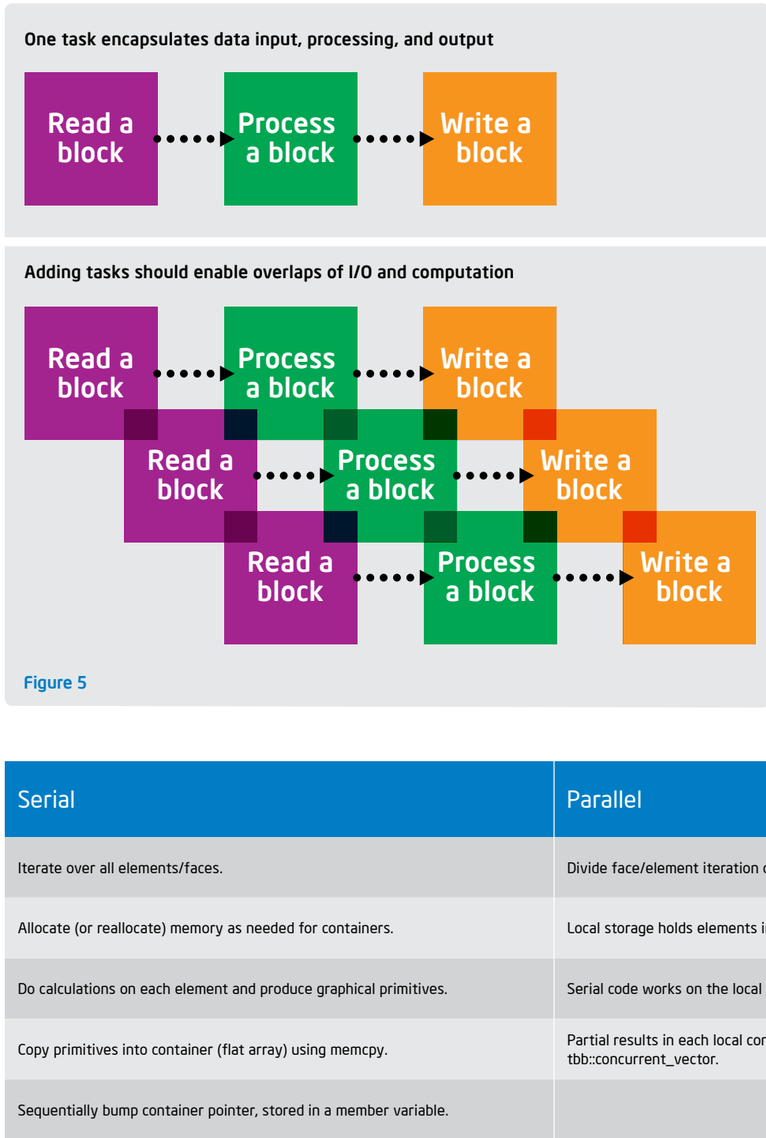


Figure 5

```
Set Input Filter
Tbb:: pipeline inpipe;
NewInPipe("inputfile.name");
Pipeline.add_filter(inpipe);

Set Process Filter
NewComputePipe compipe;
Pipeline.add_filter(compipe);

Set Output Filter
Tbb:: pipeline outpipe;
NewOutPipe("outputfile.name");
Pipeline.add_filter(outpipe);

Execute Pipeline
Pipeline.run(TOKEN_COUNT);
```

Serial	Parallel
Iterate over all elements/faces.	Divide face/element iteration over multiple threads with tbb::parallel_for.
Allocate (or reallocate) memory as needed for containers.	Local storage holds elements in each Intel TBB task.
Do calculations on each element and produce graphical primitives.	Serial code works on the local containers without modification.
Copy primitives into container (flat array) using memcpy.	Partial results in each local container safely get combined into tbb::concurrent_vector.
Sequentially bump container pointer, stored in a member variable.	

Table 2: Serial versus parallel program flow

Threading SimXpert: Phase Two

A key goal for the user experience with SimXpert is quick post-processing of analysis result data. Post-processing analysis involves transforming the initial analysis data to the final numerical form specified by the engineer, and then mapping it to its graphical primitive representation. For example, an engineer may want to direct SimXpert to render color fringe plots of von Mises, Maximum Principal, and Maximum Shear stress to investigate the performance of the simulation model relative to its applied loading. **Figure 4** demonstrates a fringe plot of the von Mises stress distribution across a simple connecting rod model.

Phase Two for SimXpert applied threading to the portion of the code responsible for graphical primitive production for fringe plots. This code accounted for approximately 35 percent of the total plot time. As a proof of concept, MSC.Software and Intel prototyped the threaded code and saw scaling up to 3.2x on four cores. The method used involved the production and packaging of graphics primitives into containers. The program flow was modified as indicated in **Table 2**.

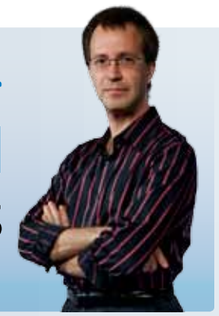
Performance improvements were observed when the models ran on a 2S 2.66GHz Intel Xeon processor 5100 series platform/8GB memory/Windows XP Professional X64 Edition Version 2003 SP2* (**Table 3**).

- A 3-D, solid, finite element simulation model, representing the casting of a V6 engine block (modelsec.xdb) with 98,814 free faces and a 358.7MB file size, achieved a 28 percent performance improvement.
- A 3-D, solid, finite element simulation model, representing a turbine blade (xx0kst0.xdb) with 65,416 free faces and 513.3MB file size, achieved a performance improvement of between 3 percent and 10 percent for various plots.
- A 3-D, solid, finite element simulation model, representing a casting of a kitchen appliance housing (xx0ust0.xdb) with 90,460 free faces and a 281.7MB file size, achieved a performance improvement of between 6 percent and 26 percent for various plots.
- A 2-D and 3-D finite element simulation model, representing a car chassis (xx0o.xdb) with 1,209,323 free faces and a 438.8MB file size, achieved a performance improvement of between 19 percent and 27 percent for various plots.
- A 3-D, solid, finite element simulation model, representing the central hub of an aircraft propeller (xx0fst0.xdb) with 89,935 free faces and a 165.2MB file size, achieved a performance improvement of between 10 percent and 30 percent for various plots.
- A 3-D, solid, finite element simulation model, representing the casting of a straight six-cylinder engine block (xx0jst0.xdb) with 461,808 free faces and a 1028.5MB file size, achieved a performance improvement of between 15 percent and 44 percent for various plots.

Next Steps

In future releases (following SimXpert R4), the remaining plot types will be threaded. The Intel TBB pipeline will also be evaluated for threading overlap processing and buffered I/O. Intel® engineers have prototyped an Intel TBB pipeline that uses the engineering calculations from Phase One. Intel Thread Profiler identified an issue in this initial implementation with buffer thrash. When fixed, the desired scalability was achieved. Matching the pipeline token count to the hardware thread count produced "laminar" scheduling and eliminated buffer thrash, resulting in a 3.9x scaling on four cores and a 7.5x to 7.8x scaling on eight cores (**Figure 5**).

BLOG highlights



Parallel Pattern 10: Scatter

BY MICHAEL MCCOOL

For a while now, I have been working on a blog series describing a set of patterns to use for parallel programming. The concept behind these patterns is that if we can identify a small set of composable deterministic patterns from which most parallel algorithms can be composed, then we can improve the structure and maintainability of parallel programs.

In this post, I want to discuss the most troublesome pattern: scatter. Scatter is required for a fully general parallel programming model, but is also the most difficult to make deterministic. However, there are several variants of scatter, some of which can be tested for determinism, and others that are guaranteed to be deterministic but may require some additional computation to ensure safety.

A scatter operation takes an array of data, an array of locations, and a target array, and updates the elements of the target array with the given data at the given locations.

Workload/Description	modelsec (engine block)	xx0kst0 (turbine blade)	xx0ust0 (housing)	xx0o (car chassis)	xx0fst0 (propeller hub)	xx0jst0 (straigt 6-cyl engine block)
File size/# free faces	358.7 MB/ 98,814	513.3 MB/ 65,416	281.7 MB/ 90,460	438.8 MB/ 1,309,323	165.2 MB/ 83,935	1028.5 MB/ 461,808
Fringe - Eigen Vectors, Translational - percent speedup	28.092					
Fringe - Stress, von Mises Avg Meth=Avg/Derive, Extrap Meth=Avg - percent speedup		3.729	8.422	19.437	10.013	15.061
Fringe - Stress, von Mises Avg Meth=Avg/Derive, Extrap Meth=Avg - percent speedup		8.103	13.03	19.791	10.863	18.856
Fringe - Stress, Tresca Avg Meth=Avg/Derive, Extrap Meth=Avg - percent speedup		8.968	10.733	20.03	10.73	18.988
Fringe - Stress, Octal Avg Meth=Avg/Derive, Extrap Meth=Avg - % speedup		3.373	6.024	19.647	10.472	15.636
Fringe - Stress, Inv 1 Avg Meth=Avg/Derive, Extrap Meth=Avg - percent speedup		2.926	6.143	19.671	10.867	15.463
Fringe - Stress, Max Shear Avg Meth=Avg/Derive, Extrap Meth=Avg - percent speedup		8.199	10.628	19.902	11.007	19.585
Fringe - Disp Trans, Mag - percent speedup		10.334	26.203	27.543	29.824	43.807

Table 3: Speedup for fringe plot optimization (average of three runs)

Conclusion

The MSC.Software project to add threading to SimXpert was successful, resulting in a significant performance improvement in SimXpert and a faster turnaround time for end users, leading to increased productivity. SimXpert was one of the first commercial applications to release with Intel TBB. Intel TBB was an ideal tool for this project since SimXpert is a multiplatform application written in C++ that has many features beyond typical high-performance computing number-crunching applications. In addition, the code of SimXpert was well suited to the incremental threading approach that MSC.Software chose.

For Phase One, measurements for seven very large customer simulation models on a 2S Intel Xeon processor 5100 series platform (four threads) showed scaling between 3.8x to 3.9x for the engineering calculations. For Phase Two, optimizations for fringe plots resulted in a speedup ranging from 3 percent to 44 percent for measured workloads. MSC.Software plans to continue with the incremental threading approach for the remaining plot types, while investigating the Intel TBB pipeline for overlapping processing and I/O.

For more information on SimXpert, visit the MSC.Software website. □

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.

Notice revision #20101101