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

Letter from the Editor
Parallelism Full Steam Ahead!, by JAMES REINDERS ................................. 4
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.

Enhancing Productivity and Achieving High Performance
with Intel® Cluster Toolkit Compiler Edition, by BILL MAGRO ............ 6
Message-passing interface applications port seamlessly from dual-core desktops to multithousand server clusters, a key advantage of distributed memory parallelism.

Increase Productivity and Performance:
Find out What Incredibuild* and Intel® Parallel Composer Can Offer, by JENNIFER JIAng AND URI mISHOL .................. 12
Software companies use many software development methodologies, but none eliminate the need for building, testing, and tuning individual components or the whole application.

Optimizations for MSC.Software SimXpert*
Using Intel® Threading Building Blocks, by KATHY CARVER, MARK LUBIN, AND BONNIE AGNA .......... 24
To address increasing customer model sizes and align with the multicore processor roadmaps for hardware vendors, MSC.Software* engaged with Intel to thread SimXpert*.

DEVELOPER ROCK STAR:
Robert Geva

APP EXPERTISE:
C++ and Fortran Compilers

Robert’s tip to boost performance:
With wider SIMD instructions in Intel® Architecture, expect more from the vectorizer in the Intel® compiler. Use the -Qguide option in Intel® Parallel Composer to get guidance on how simple, local restructuring of your code can get more code vectorized and parallelized by the compiler.

TURN UP YOUR PRODUCTIVITY.

ROCK YOUR CODE.


© 2010, Intel Corporation. All rights reserved. Intel and the Intel logo are trademarks of Intel Corporation in the U.S. and other countries. Other names and brands may be claimed as the property of others.
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 will 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 curve 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 is a valid observation, even if not completely quantitated. 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. 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.

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 3.0, 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 part 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 REINDEERS
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.
Enhancing Productivity and Achieving High Performance with Intel® Cluster Toolkit Compiler Edition

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

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’re 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 mean lower development and validation costs, while improved compatibility with a wide range of interconnects allows vendors to write a single set of memory buffers and pinning code that will work with almost any combination of network devices.

As a result, local memory usage no longer grows with the number of endpoints, all from a single set of memory buffers. This avoids the need for recompiling or relinking. 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.


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 the paying the (red) 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_U=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 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.

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 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.
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 tiling. 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 1048 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 recording 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 BD 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 BD 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 overheads due to latency 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 (Figure 3).

While the ideal interconnect is one useful trace transform, you can write and apply your own through the custom plug-in framework—a 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 Qlic’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 Qlic’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.
Increase Productivity and Performance:

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

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.

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.
Build-Time Comparison

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 1: Possible build-time speedup using IncrediBuild

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

Figure 3: Intel Parallel Composer features

Code and Debug with Intel® Parallel Composer

Find where to start parallelizing

Introduce threads, compile, and debug with Intel® Parallel Composer

Find threading and memory errors with Intel® Parallel Inspector

Tune for optimal concurrency usage with Intel® Parallel Amplifier

Performance

- Highly optimizing C/C++ compiler and runtime libraries
- OpenMP® 3.0
- Intel® Threading Building Blocks
- Intel® Integrated Performance Primitives

Parallelism

Productivity

- Intel® Parallel Debugger Extension
- Microsoft Visual Studio integration and compatibility

```c
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

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 number 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 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

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

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. Using Incredibuild, you can, for example, specify the /Qx option to enable auto-vectorization.

---

**Autovectorization**

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

![Auto-vectorization](image)

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

The latest Incredibuild version 3.51 supports the following Windows OS versions:
- Please visit the following pages for more information about Incredibuild:
  - FAQs about Incredibuild
  - What others have to say about this tool

---

**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.

---

**Example:**

```c
void work(float* a, float* b, float* c, int MAX)
{
    for (int I=0; I<=MAX; I++)
        c[I] = a[I] + b[I];
}
```
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, code 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.”

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

When /Qip 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
- C++ class hierarchy analysis
- Alias analysis
- Indirect call conversion

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

Auto-parallelization: controlled by the option /Parallel 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.

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 Patches (Intel® IPP). And best of all, all of 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 processor 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? (Table 1)

<table>
<thead>
<tr>
<th>Optimization Type</th>
<th>How it Benefits Performance</th>
</tr>
</thead>
<tbody>
<tr>
<td>Auto-vectorization</td>
<td>Improves 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.</td>
</tr>
<tr>
<td>Auto-parallelization</td>
<td>Exploits instruction-level parallelism and data level parallelism. It analyzes the code and dataflow to determine if the loops are good worksharing candidates and partitions the data for threaded code.</td>
</tr>
<tr>
<td>Interprocedural Optimization</td>
<td>Exploits 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, C++ class hierarchy analysis, alias analysis, and indirect call conversion.</td>
</tr>
</tbody>
</table>

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
- C++ class hierarchy analysis
- Alias analysis
- Indirect call conversion

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

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 loop depend
- #pragma loop_count min(n), max(n), avg(n)

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

Auto-parallelization: controlled by the option /Parallel 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.
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 text.
- A total of eight projects are being rebuilt.
- Each project is rebuilt using dedicated CPUs: two CPUs from Sys1, one CPU 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: 0:00:05
7. Build Time: 0:00:06
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

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.
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 "/D2" 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

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.

---

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

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.

*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

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.

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
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

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

<table>
<thead>
<tr>
<th>Plot</th>
<th>File name/Entity count</th>
<th>Serial time (sec)</th>
<th>Parallel time (sec)</th>
<th>Speedup factor</th>
<th>Serial process time (sec)</th>
<th>Parallel process time (sec)</th>
<th>Percent process speedup</th>
<th>Percent time spent in numeric operations</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fringe - Stress, Max Princ</td>
<td>xx0kst0.xdb/624924</td>
<td>0.765</td>
<td>0.196</td>
<td>3.903</td>
<td>10.22</td>
<td>9.65</td>
<td>5.579</td>
<td>7.48</td>
</tr>
<tr>
<td>Fringe - Stress, Mid Princ</td>
<td>xx0kst0.xdb/624924</td>
<td>0.763</td>
<td>0.195</td>
<td>3.904</td>
<td>10.209</td>
<td>9.635</td>
<td>5.623</td>
<td>7.47</td>
</tr>
<tr>
<td>Fringe - Stress, Min Princ</td>
<td>xx0kst0.xdb/624924</td>
<td>0.762</td>
<td>0.197</td>
<td>3.873</td>
<td>10.208</td>
<td>9.636</td>
<td>5.604</td>
<td>7.46</td>
</tr>
<tr>
<td>Fringe - Stress, Tresca</td>
<td>xx0kst0.xdb/624924</td>
<td>0.767</td>
<td>0.196</td>
<td>3.905</td>
<td>10.228</td>
<td>9.675</td>
<td>5.410</td>
<td>7.50</td>
</tr>
<tr>
<td>nunfringe - Stress, Max Princ</td>
<td>xx0ust0.xdb/605288</td>
<td>0.696</td>
<td>0.180</td>
<td>3.874</td>
<td>9.573</td>
<td>9.152</td>
<td>4.401</td>
<td>7.27</td>
</tr>
<tr>
<td>Fringe - Stress, Mid Princ</td>
<td>xx0ust0.xdb/605288</td>
<td>0.691</td>
<td>0.181</td>
<td>3.820</td>
<td>9.553</td>
<td>9.110</td>
<td>4.641</td>
<td>7.24</td>
</tr>
<tr>
<td>Fringe - Stress, Min Princ</td>
<td>xx0ust0.xdb/605288</td>
<td>0.693</td>
<td>0.179</td>
<td>3.879</td>
<td>9.556</td>
<td>9.114</td>
<td>4.626</td>
<td>7.25</td>
</tr>
<tr>
<td>Fringe - Stress, Tresca</td>
<td>xx0ust0.xdb/605288</td>
<td>0.693</td>
<td>0.178</td>
<td>3.886</td>
<td>9.584</td>
<td>9.105</td>
<td>4.998</td>
<td>7.23</td>
</tr>
<tr>
<td>Fringe - Stress, Max Shear</td>
<td>xx0jst0.xdb/2394421</td>
<td>2.883</td>
<td>0.731</td>
<td>3.942</td>
<td>39.068</td>
<td>37.007</td>
<td>5.275</td>
<td>7.38</td>
</tr>
<tr>
<td>Fringe - Stress, Mid Shear</td>
<td>xx0jst0.xdb/2394421</td>
<td>2.888</td>
<td>0.730</td>
<td>3.956</td>
<td>39.090</td>
<td>39.945</td>
<td>5.486</td>
<td>7.39</td>
</tr>
<tr>
<td>Fringe - Stress, Min Shear</td>
<td>xx0jst0.xdb/2394421</td>
<td>2.880</td>
<td>0.730</td>
<td>3.947</td>
<td>39.086</td>
<td>39.816</td>
<td>5.808</td>
<td>7.37</td>
</tr>
<tr>
<td>Fringe - Stress, Tresca</td>
<td>xx0jst0.xdb/2394421</td>
<td>2.874</td>
<td>0.730</td>
<td>3.937</td>
<td>39.996</td>
<td>39.833</td>
<td>3.061</td>
<td>7.56</td>
</tr>
<tr>
<td>Fringe - Stress, Max Shear</td>
<td>xx0jst0.xdb/2394421</td>
<td>2.894</td>
<td>0.732</td>
<td>3.952</td>
<td>39.433</td>
<td>39.277</td>
<td>2.932</td>
<td>7.34</td>
</tr>
<tr>
<td>Average</td>
<td></td>
<td>3.90</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>4.872</td>
<td>7.37</td>
</tr>
<tr>
<td>Minimum</td>
<td></td>
<td>3.82</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>2.932</td>
<td>7.23</td>
</tr>
<tr>
<td>Maximum</td>
<td></td>
<td>3.96</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>5.808</td>
<td>7.56</td>
</tr>
</tbody>
</table>

Table 1: Summary for plots where serial time in numeric operations was greater than 0.5 seconds
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 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 (modelec.xdb) with 1,085,115 free faces and a 3.597MB file size, achieved a 28 percent performance improvement.
- A 3-D, solid, finite element simulation model, representing a turbine blade (codf10x0.dbx) with 65,416 free faces and 513.8MB file size, achieved a performance improvement of between 8 percent and 10 percent for various plots.
- A 3-D, solid, finite element simulation model, representing a casting of a kitchen appliance housing (codf02x0.dbx) with 90,655 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 (codx100.dbx) with 1,209,322 free faces and a 4.885GB 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 (codx9100.dbx) with 493,935 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 (codx110.dbx) with 461,808 free faces and a 1028.6MB file size, achieved a performance improvement of between 15 percent and 44 percent for various plots.

Table 2: Serial versus parallel program flow

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 “faminat” scheduling and eliminated buffer thrash, resulting in a 3.9x scaling on four cores and a 5.7x to 7.8x scaling on eight cores.


Figure 4:
- A 3-D, solid, finite element simulation model, representing the central hub of an aircraft propeller (codf10x0.dbx) with 90,655 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 (codx100.dbx) with 1,209,322 free faces and a 4.885GB 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 (codx9100.dbx) with 493,935 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 (codx110.dbx) with 461,808 free faces and a 1028.6MB file size, achieved a performance improvement of between 15 percent and 44 percent for various plots.

Table 2: Serial versus parallel program flow

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 “faminat” scheduling and eliminated buffer thrash, resulting in a 3.9x scaling on four cores and a 5.7x to 7.8x scaling on eight cores. (Figure 5).

**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.

**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.
Rock your code.

Intel® compilers, libraries, and debugging and tuning tools provide everything you need to roll out reliable apps that scale for today’s multicore innovations. From supercomputers to laptops, and embedded systems to mobile devices, Intel® software tools enable you to optimize legacy serial and threaded code and plug in to multicore.