THE PARALLEL UNIVERSE

Issue 5
November 2010

Letter from the
EDITOR
by James Reinders

INTEL® PARALLEL BUILDING BLOCKS
The Answer(s) to Cracking the Parallelism Puzzle

by David Sekowski

Simplifying High Performance with
INTEL® PARALLEL STUDIO XE AND INTEL® CLUSTER STUDIO TOOL SUITES

by Sanjay Goil and John McHugh

INTEL® PARALLEL STUDIO HAS GONE EXTREME.

FROM THE MAKERS OF Intel® VTune™ Performance Analyzer and Intel® Visual Fortran Compiler comes the ultimate all-in-one performance toolkit:

INTRODUCING INTEL® PARALLEL STUDIO XE

Intel Parallel Studio XE for Linux‡

Purchase the tools individually or get the complete suite and save. See the table for important name change information.

PREVIOUS PRODUCT NAMES	NEW PRODUCT NAMES
	Intel® Parallel Studio XE
	Intel® C++ Studio XE
Intel® Cluster Toolkit Compiler Edition	Intel® Cluster Studio
Intel® Compiler Suite Professional Edition	Intel® Composer XE
Intel® C++ Compiler Professional Edition	Intel® C++ Composer XE
Intel® Visual Fortran Compiler Professional Edition	Intel® Visual Fortran Composer XE
Intel® Fortran Compiler Professional Edition	Intel® Fortran Composer XE
Intel® VTune™ Performance Analyzer *(including Intel® Thread Profiler)*	Intel® VTune™ Amplifier XE
Intel® Thread Checker	Intel® Inspector XE

‡ Intel Parallel Studio XE also supports Windows and Mac OS X.

The ultimate all-in-one performance toolkit

Intel Parallel Studio XE combines Intel's industry-leading C/C++ and Fortran compilers, performance and parallel libraries, correctness analyzers, code-quality tools, and performance profilers with ease-of-use innovations to create an integrated tool suite that helps high-performance computing and enterprise developers boost application performance, reliability, and security.

Advanced compilers and libraries
Intel® Composer XE

Advanced memory, threading, and security analyzer
Intel® Inspector XE

Advanced performance profiler
Intel® VTune™ Amplifier XE

Rock your code. Rock your world.

Get a free 30-day trial of Intel Parallel Studio XE today at http://software.intel.com/en-us/articles/intel-parallel-studio-xe/.

intel®

CONTENTS

Sign up for future issues | Share with a friend

High Performance Options Have Never Been Greater

LETTER FROM THE EDITOR

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.*

This issue of *The Parallel Universe* magazine focuses on the latest Intel® software developer tools that help us tap into the performance offered by today's computers.

From a technology standpoint, the hardware has never been more complex. Even a single feature such as parallelism is present at every level of the computer architecture: superscalar processors with expanding issue rates, SIMD instructions with expanding widths, processors with expanding numbers of cores, and systems with expanding numbers of processors.

As systems become more complex, our tools for software developers have evolved to help software developers. Your feedback has been most helpful. Over the past year we've taken major strides in delivering developer tools for existing and emerging hardware, while simplifying the tools you need for high performance. Our focus has been to offer choice, protect your software investments, and help you scale forward to emerging and future hardware.

On November 9 we introduced the latest installments in our most popular software developer tools. These numerous tools come together to form two comprehensive studios: Intel® Parallel Studio XE and Intel® Cluster Studio. Intel Parallel Studio XE addresses the advanced performance challenges of today's machines for C, C++, and Fortran developers. Intel Cluster Studio offers tools uniquely tailored for distributed computing, specifically helping with programs using MPI, C, C++, and Fortran.

There are a lot of new features to be excited about, including version 12.0 Intel® compilers, a new version of Intel® VTune™ Performance Analyzer, new concurrency-compatible memory checking capabilities, code analysis for security and robustness, advanced MPI support, Intel® Parallel Building Blocks for C/C++, co-array Fortran support, and new threading error detection that handles not only compiled languages, but also .NET code.

Intel Parallel Studio XE, is available for Linux* and Windows* developers. It includes Intel® Composer XE (compilers and libraries), Intel® VTune Amplifier XE, and Intel® Inspector XE.

Intel Cluster Studio, is available for Linux and Windows developers. It includes Intel Composer XE, Intel® MPI Library and MPI Benchmarks, and the Intel® Trace Analyzer and Collector.

Our tools offer a rich selection of parallel programming methods to meet the numerous needs of different applications. They have no equal providing robust ways to express parallelism: OpenMP*, MPI, co-array Fortran, Intel® Math Kernel Library (Intel® MKL), Intel® Integrated Performance Primitives (Intel® IPP), and Intel® Parallel Building Blocks (Intel® PBB) which includes Intel® Threading Building Blocks (Intel® TBB), Intel® Cilk Plus, and Intel® Array Building Blocks (Intel® ArBB).

To explore the advantages of these innovative tools, you'll find articles on Intel Parallel Building Blocks and the parts that make it up, "what's new and exciting in Fortran after all these years," and Intel MKL.

Eliminating defects is an important topic that gets attention in our tools as well, and in a way that's easy to utilize in your build environment. Our tools offer solutions for code quality, security, and application robustness, all applicable to parallel programs. The next-generation correctness analyzers combine memory, threading, and code analysis for security. The article "Intel® Inspector XE: An essential tool during development along with Intel® Composer XE" advocates this tool as an essential and regular part of your development cycle. The case for it seems clear.

Performance profiling is essential for high performance in detecting hotspots, and helping you alleviate them with additional insight into what is actually happening on your system. Our next-generation profiler, the Intel VTune Amplifier XE, provides easy-to-use, yet detailed, insight into the most pressing performance issues.
For the cluster developer, we have our highly scalable implementation of MPI, in the Intel MPI library, architected to scale to the largest systems. The article "On a path to petascale with commodity clusters and Intel MPI" highlights Intel MPI library advancements in our cluster tools for HPC.

Continuous development of software for high performance is a complex undertaking. Intel® software development tools work with existing and emerging Intel architecture, extending Intel leadership in processor technology, and in multicore and manycore processors. As customers, you look for predictability in your software development, and an assurance that the software investments you make today will continue to reap benefits in years to come. Our mission in the software tools group is to simplify the tools—and the way you purchase, install, develop, and support them.

Our Beta customers said very nice things about our new tools prior to release. I believe you will find Intel Parallel Studio XE and Intel Cluster Studio taking significant strides in advancing the innovation bar for programming, productivity, and programmability for high performance.

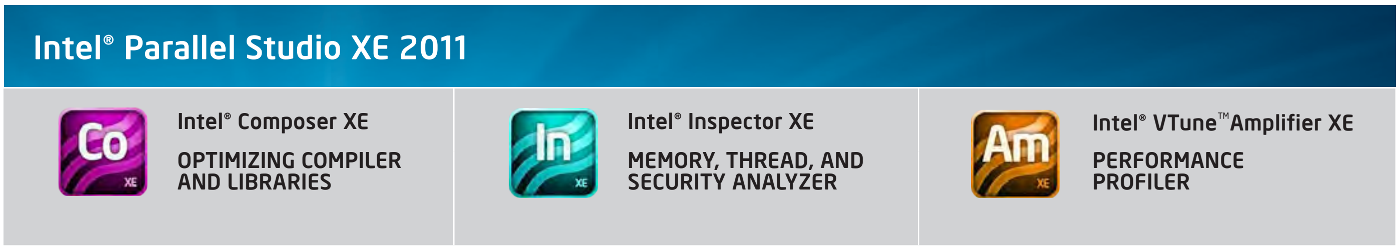
Enjoy!

JAMES REINDERS
Portland, Oregon
November 2010

Simplifying High Performance with

INTEL® PARALLEL STUDIO XE AND INTEL® CLUSTER STUDIO TOOL SUITES

By Sanjay Goil, Product Marketing Manager
John McHugh, Marketing Communication Manager
Intel® Software Development Products



Intel® Parallel Studio XE 2011

Intel® Composer XE — OPTIMIZING COMPILER AND LIBRARIES

Intel® Inspector XE — MEMORY, THREAD, AND SECURITY ANALYZER

Intel® VTune™ Amplifier XE — PERFORMANCE PROFILER

Figure 1

In September, Intel introduced Intel® Parallel Studio 2011, a tool suite for Microsoft* Windows* Visual Studio* C++ developers, with the singular objective of providing the essential performance tools for application development on Intel® Architecture. These tools provide significant innovation, and enable unprecedented developer productivity when building, debugging, and tuning parallel applications for multicore. With the introduction of Intel® Parallel Building Blocks (Intel® PBB), developers have methods to introduce and extend parallelism in C/C++ applications for higher performance and efficiencies.

This month Intel is extending the reach of next-generation Intel tools to developers of applications on both Windows and Linux in C/C++ and Fortran who need advanced performance for multicore today and forward scaling to manycore. Intel Parallel Studio XE 2011 contains C/C++ and Fortran Compilers, Intel® Math Kernel Library (Intel® MKL) and Intel® Integrated Performance Primitives (Intel® IPP) performance libraries, Intel PBB libraries, Intel® Threading Building Blocks (Intel® TBB), Intel® Cilk™ Plus, and Intel® Array Building Blocks (Intel® ArBB), Intel® Inspector XE correctness analyzer, and Intel® VTune™ Amplifier XE performance profiler.

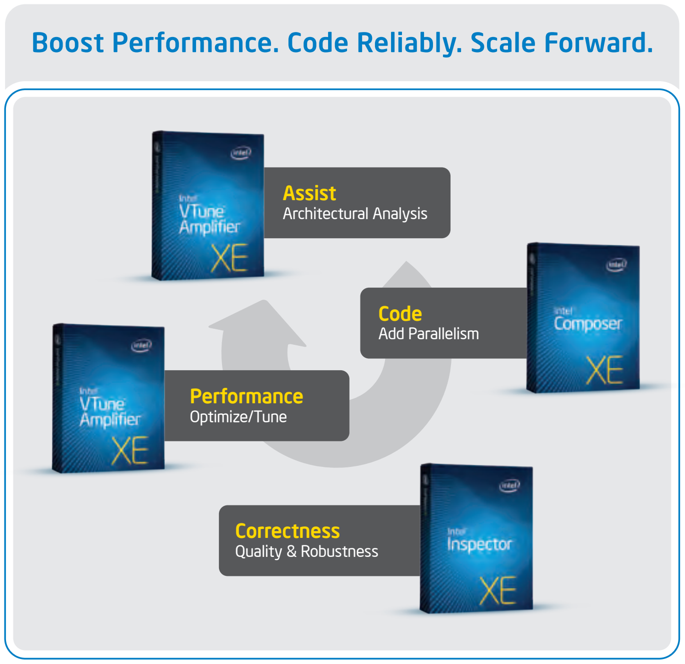
HPC programmers have traditionally been able to use all the compute power made available to them. Even with the performance leaps that Moore's law has allowed Intel architecture to deliver over the past decade, the hunger for additional performance continues to thrive. There are big unsolved problems in science and engineering, physical simulations at higher granularities, and problems where the economically viable compute power provides lower resolution or piecemeal simulation of smaller portions of the larger problem. This is what makes serving the HPC market so exciting for Intel, and it is a significant driver for innovation in both hardware and software methodologies for parallelism and performance.

Intel® Cluster Studio introduces tools for HPC cluster development with MPI, including the scalable Intel® MPI Library and Intel® Trace Analyzer and Collector performance profiler, with the industry-leading C/C++ and Fortran compilers for a complete cluster development tool suite. This is combined with the ease of deployment offered by the Intel® Cluster Ready program, making deployment of cluster applications highly efficient.

Introducing New Tool Suites

Software developers of high performance applications require a complete set of development tools. While traditionally these tools include compilers, debuggers, and performance and parallel libraries, more often the issues in development come in error correctness and performance profiling. The code doesn't run correctly, or exhibits error-prone behavior on some runs, pointing to data races, deadlocks, or performance bottlenecks in locks for synchronization, or exposes security risks at runtime. To this end, Intel's correctness analyzers and performance profilers are a great addition to the development environment for highly robust and secure code development. **Figure 1**.

For advanced and distributed performance, Intel is simplifying the procurement, deployment, and use of HPC tools on IA-32 and Intel® Architecture and compatible platforms, and HPC clusters programmed with the Message Passing Interface (MPI). **Figure 2**.



Phase	Productivity Tool	Feature	Benefit
Advanced Code	Intel® Composer XE	C/C++ and Fortran compilers, performance libraries, and parallel models	> Drives application performance and scalability benefits of multicore and forward scales to manycore. Additionally, provides code robustness and security.
Advanced Correctness	Intel® Inspector XE	Memory and threading error checking tool for higher code reliability and quality	> Increases productivity and lowers cost by catching memory and threading defects early
Advanced Performance	Intel® VTune™ Amplifier XE	Performance profiler to optimize performance and scalability	> Removes guesswork, saves time, and makes it easier to find performance and scalability bottlenecks. Combines ease of use with deeper insights.

Figure 2

Highlights of Intel® Parallel Studio XE 2011

> **Available for Multiple Operating Systems:** Intel® Parallel Studio XE provides the same set of tools to aid development for both Windows* and Linux* platforms. C/C++, Fortran compilers, and performance and parallelism libraries bring advanced optimizations to Mac OS*X.

> **Robustness:** Intel® Inspector XE's memory and thread analyzer finds and pinpoints memory and threading errors before they happen.

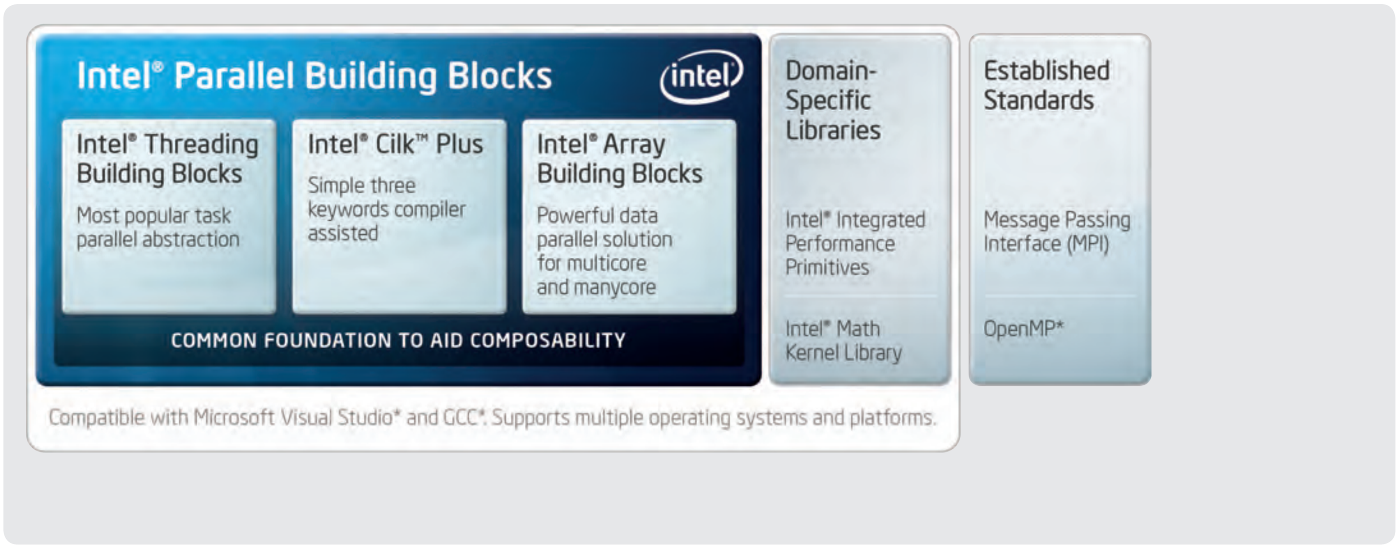
> **Code Quality:** Intel Parallel Studio XE enables developers to effectively find software security vulnerabilities through static security analysis.

> **Advanced Optimization:** The compilers and libraries in Intel® Composer XE offer advanced vectorization support, including support for Intel® AVX. The C/C++ optimizing compiler now includes Intel® PBB library, expanding the types of problems that can be solved more easily in parallelism with increased scalability and reliability. For Fortran developers, it now offers co-array Fortran and additional support for the Fortran 2008 standard.

> **Performance:** Intel® VTune™ Amplifier XE performance profiler finds bottlenecks in serial and parallel code that limit performance. Improvements include a more intuitive interface, fast statistical call graph, and timeline view. Intel® MKL and Intel® IPP performance libraries provide robust multicore performance for commonly used math and data processing routines. A simple linking of the application with these libraries is an easy first step for multicore parallelism.

> **Compatibility and Support:** Intel Parallel Studio XE excels at compatibility with leading development environments and compilers. Intel offers broad support with forums and Intel® Premier Support, which provides fast answers and covers all software updates for one year.

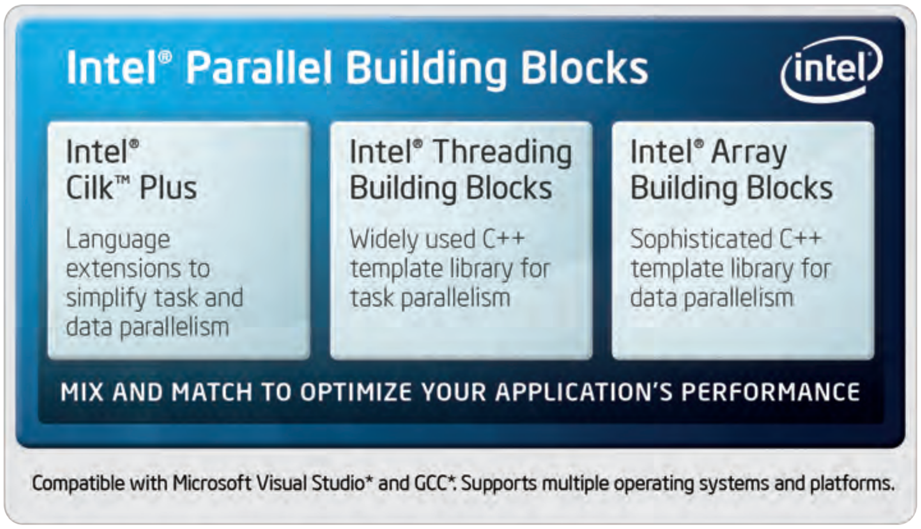
A software development project goes through several steps to get optimal performance on the target platform. Most often, the developer gets a rudimentary performance profile of the application run to show hotspots. Once opportunities for optimization are identified, the coding aspects are handled by the compilers and performance and parallel libraries to add parallelism, presenting task level, data level, and vectorization opportunities. Finally, the correctness tools make robust code possible by checking for threading and memory errors, and identifying security vulnerabilities. This cycle typically repeats itself to find higher application efficiencies.


Intel® Parallel Building Blocks
intel
Intel® Threading Building Blocks
Most popular task parallel abstraction
Intel® Cilk™ Plus
Simple three keywords compiler assisted
Intel® Array Building Blocks
Powerful data parallel solution for multicore and manycore
COMMON FOUNDATION TO AID COMPOSABILITY
Domain-Specific Libraries
Intel® Integrated Performance Primitives
Intel® Math Kernel Library
Established Standards
Message Passing Interface (MPI)
OpenMP*
Compatible with Microsoft Visual Studio* and GCC*. Supports multiple operating systems and platforms.

Old Name	New Name
Intel® Compiler Suite Professional Edition	**Intel® Composer XE**
Intel® C++ Compiler Professional Edition	**Intel® C++ Composer XE**
Intel® Visual Fortran Compiler Professional Edition	**Intel® Visual Fortran Composer XE**
Intel® Visual Fortran Compiler Professional Edition with IMSL	**Intel® Visual Fortran Composer XE with IMSL**
Intel® VTune™ Performance Analyzer (including Intel® Thread Profiler)	**Intel® VTune™ Amplifier XE**
Intel® Thread Checker	**Intel® Inspector XE**
Intel® Cluster Toolkit Compiler Edition	**Intel® Cluster Studio**

Figure 3

The tools introduced in Intel Parallel Studio XE 2011 are next-generation revisions of industry-leading tools for C/C++ and Fortran developers seeking cross-platform capabilities for the latest x86 processors on Windows* and Linux* platforms. Those familiar with Intel's industry-leading tools will see that the product names have transitioned in this new release—in all cases with significant additional capabilities, other names remain the same. **Figure 3**.



Intel® Parallel Building Blocks

Intel® Cilk™ Plus
Language extensions to simplify task and data parallelism

Intel® Threading Building Blocks
Widely used C++ template library for task parallelism

Intel® Array Building Blocks
Sophisticated C++ template library for data parallelism

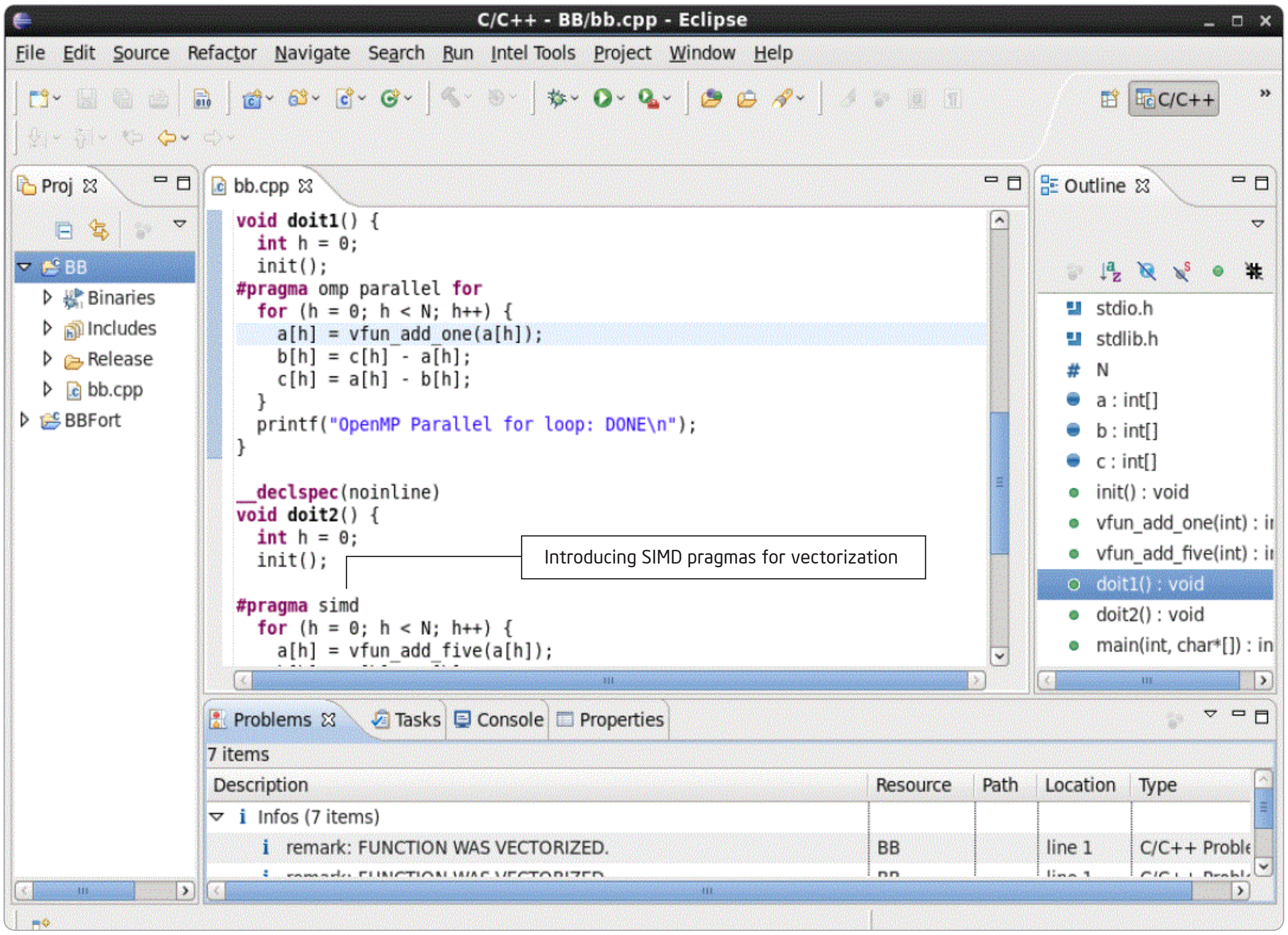
MIX AND MATCH TO OPTIMIZE YOUR APPLICATION'S PERFORMANCE

Compatible with Microsoft Visual Studio* and GCC*. Supports multiple operating systems and platforms.

Figure 4

"Intel® Parallel Studio XE 2011 is a great software development tool for performance-oriented Windows*-based C++ software developers. I achieved an astonishing boost in performance by using Intel® Cilk Plus and Array features in my code. If you need performance, try Intel Parallel Studio XE 2011."

Jorge Martinis
Research and Development Engineer, BR&E Inc.

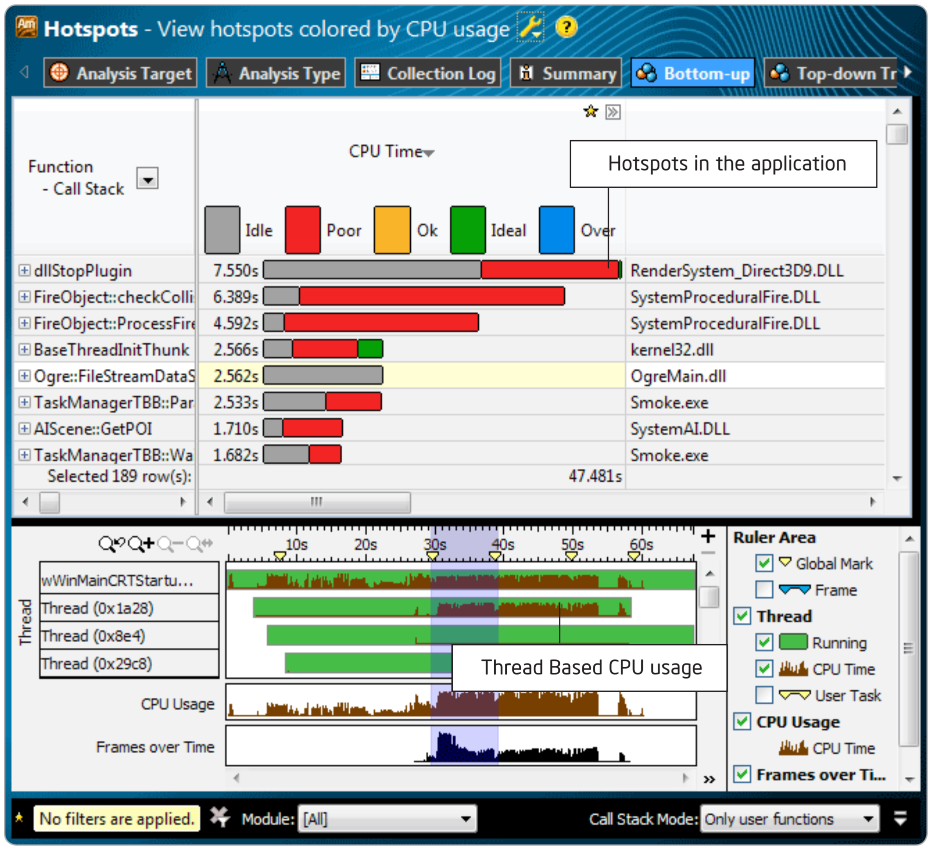


```
void doit1() {
  int h = 0;
  init();
#pragma omp parallel for
  for (h = 0; h < N; h++) {
    a[h] = vfun_add_one(a[h]);
    b[h] = c[h] - a[h];
    c[h] = a[h] - b[h];
  }
  printf("OpenMP Parallel for loop: DONE\n");
}

__declspec(noinline)
void doit2() {
  int h = 0;
  init();

#pragma simd
  for (h = 0; h < N; h++) {
    a[h] = vfun_add_five(a[h]);
```

7 items

Description	Resource	Path	Location	Type
Infos (7 items)				
remark: FUNCTION WAS VECTORIZED.	BB		line 1	C/C++ Proble

Figure 5



Function - Call Stack	CPU Time	
dllStopPlugin	7.550s	RenderSystem_Direct3D9.DLL
FireObject::checkColli	6.389s	SystemProceduralFire.DLL
FireObject::ProcessFire	4.592s	SystemProceduralFire.DLL
BaseThreadInitThunk	2.566s	kernel32.dll
Ogre::FileStreamDataS	2.562s	OgreMain.dll
TaskManagerTBB::Par	2.533s	Smoke.exe
AIScene::GetPOI	1.710s	SystemAI.DLL
TaskManagerTBB::Wa	1.682s	Smoke.exe
Selected 189 row(s):	47.481s	

Figure 6

"BlueJeans Network is working on the next-generation video cloud-processing solution. We process large volumes of audio, video, and data content, and these processes are highly CPU intensive. Intel® IPP 7.0 worked great for us. Its comprehensive set of audio and video processing functionality was the perfect solution for our needs. It was a tremendous timesaver for us—as building these from scratch would have taken us forever! I definitely recommend Intel IPP 7.0."

Emmanuel Weber
Software Architect
BlueJeans Network

What's New in Intel® Composer XE

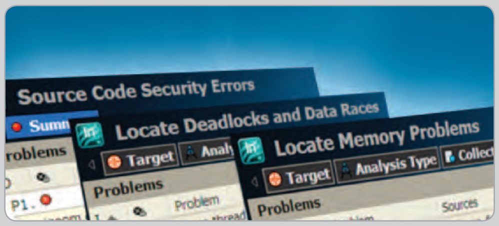
Intel Composer XE contains next-generation C/C++ and Fortran compilers (v12.0) and performance and parallel libraries: Intel MKL 10.3, Intel IPP 7.0, and Intel TBB 3.0. **Figure 2**.

The latest Intel C/C++ compiler, Intel® C++ Compiler XE 12.0, is optimized for the latest Intel architecture processor (code-named Sandy Bridge) with Intel AVX support. The product contains Intel PBB, which includes advances in mixing and matching task, vector, and data parallelism in applications to better map to the multicore optimization opportunities; Intel Cilk Plus; Intel TBB; and Intel ArBB (in beta, available separately). **Figure 4**. There are vector optimizations with Intel AVX with SIMD pragmas, in addition to GAP, an array notation tool to help in auto-parallelization for the highest performance and parallelism on the latest generation of x86 multicore CPUs. For Windows users, support for Visual Studio 2010* is included.

Intel® Fortran Compiler XE 12.0 includes several advances in more complete support for Fortran 2003 standard and some support for Fortran 2008 standards, including Co-array Fortran, vector optimizations with AVX, and help with auto-parallelization for the highest performance and parallelism on the latest x86 multicore CPUs. **Figure 5**.

The performance libraries continue to provide an easy way to include highly optimized and automatically parallel math and scientific functions, and data processing routines for high performance users. The math library, Intel MKL 10.3, includes enhancements such as better Intel AVX support, summary statistics library, and enhanced C language support for LAPACK. The data processing library, Intel IPP 7.0, includes improved data compression and codecs, and support for Intel AVX and AES instructions, continuing to excel at data processing intensive application domains.

Intel® Inspector XE
Memory, Threading, and Security Checker



> Increase application reliability and security

Figure 7

"It was an easy and fast ramp to start using the Intel® Inspector XE 2011 tool. We were able to set the analysis level, obtain a visual interpretation of the collected data, and get helpful information on hidden data races in the code quickly."

Alex Migdalski
CEO and CTO
OTRADA Inc.

Enhanced Developer Productivity with Correctness Analyzers and Performance Profilers

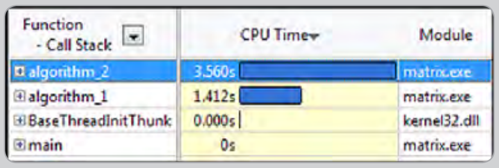
Intel Parallel Studio XE 2011 combines ease-of-use innovations, introduced in Intel Parallel Studio, with advanced functionality for high performance, scalability and code robustness for Linux and Windows. Intel has traditionally offered developer tools on both Windows and Linux, and strives to offer the same functionality across both platforms, especially important for developing applications to run on both operating systems. **Figure 6**.

With the capabilities in the correctness analyzer, Intel Inspector XE, **Figure 7**, the product helps C/C++ and Fortran developer with static and dynamic code analysis through threading and memory analysis tools to develop highly robust, secure, and highly optimized applications.

New capabilities in the Intel® Inspector XE correctness analyzer include:

- Simplified configuration and run analysis
- Finds coding defects quickly, such as:
    - Memory leaks and memory corruption
    - Threading data races and deadlocks
- Supports native threads, understands any parallel model built on top of threads
- Dynamic instrumentation works on standard builds and binaries
- Timeline view to explore context of the respective threads
- Intuitive standalone GUI and command line interface for Windows and Linux
- Advanced command line reporting

Intel® VTune™ Amplifier XE - Hotspot Performance Profiler



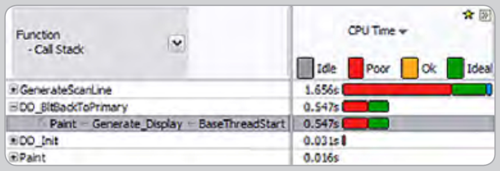
Function - Call Stack	CPU Time	Module
algorithm_2	3.560s	matrix.exe
algorithm_1	1.412s	matrix.exe
BaseThreadInitThunk	0.000s	kernel32.dll
main	0s	matrix.exe

- Find performance bottlenecks
- Functions sorted by amount of CPU time

Figure 8

"Intel VTune Amplifier XE 2011 is the next generation of the Intel VTune Analyzer…"

Intel® VTune™ Amplifier XE - Concurrency Performance Profiler



Function - Call Stack	CPU Time
GenerateScanLine	1.656s
DD_BltBackToPrimary	0.547s
Paint ← Generate_Display ← BaseThreadStart	0.547s
DD_Init	0.031s
Paint	0.016s

- Color shows # of cores utilized
- Click {+} to view call stacks

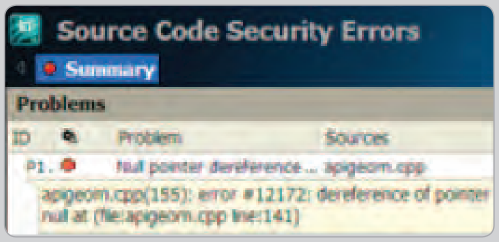
Figure 9

Intel VTune Amplifier XE 2011 is the next generation of the Intel VTune Performance Analyzer, which is a powerful tool to quickly find and provide greater insights into multicore performance bottlenecks. It removes the guesswork and analyzes performance behavior in Windows* and Linux* applications, providing quick access to scalability bottlenecks for faster and improved decision making. **Figures 8,9**.

The next-generation Intel® VTune™ performance profiler has new features, including:

- Easy, predefined analyses
- Fast hotspot analysis (hot functions and call stack)
- Powerful filtering
- Threading timeline
- Frame analysis
- Attach to a running process (Windows)
- Event multiplexing
- Simplified remote collection
- Improved compare results
- Tight Visual Studio* integration
- Non-root Linux* install
- Only EBS driver install needs root

Intel® Inspector XE - Security Checker
Memory, Threading, & Security Checker



P1. Null pointer dereference ... apigeom.cpp

apigeom.cpp(155): error #12172: dereference of pointer null at (file:apigeom.cpp line:141)

When used with Intel® Parallel Studio XE

- Improve code security with static analysis
- Find buffer overruns, unsafe library usage, uninitialized variables, bad pointers

Figure 10

"Intel® static security analysis (SSA) allowed us to easily find lots of potential flaws, thus preventing future bugs or misuse to occur. Ultimately, we expect the SSA to help us not reproduce typical security flaws."

Mikael Le Guerroue
Senior Architecture Engineer
Envivio

Software security starts very early in the development phase, and Intel Parallel Studio XE 2011 makes it faster to identify, locate, and fix software issues prior to software deployment. This helps identify and prevent critical software security vulnerabilities early in the development cycle, where the cost of finding and fixing errors is the lowest. **Figure 10,11**.

Intel's static security analysis (SSA), included in the Intel® Parallel Studio XE bundle, provides unique advantages for robust code development:

- Easier, faster setup and ramp to get static analysis results
- Simple approach to configure and run static analysis
- Discovers and fixes defects at any phase of the development cycle
- Finds more than 250 security errors, such as:
  - Buffer overruns and uninitialized variables
  - Unsafe library usage and arithmetic overflow
  - Unchecked input and heap corruption
- Tracks state associated with issues, even as source evolves and line numbers change
- Displays problem sets and location of source
- Provides filters, assignment of priority, and maintenance of problem set state
- Intuitive standalone GUI and command line interface for Windows and Linux

Feature	Benefit
Support for both Linux* and Windows* platforms	Development capability with the same set of tools on both Windows* and Linux* platforms; enhanced performance, productivity, and programmability
C/C++ Compilers with Intel® Parallel Building Blocks	Breakthrough in providing choice of parallelism for applications— task, data, vector— with mix and match for optimizing application performance. C/C++ standards support
Fortran Compilers with key Fortran 2008 standards support including Co-Array Fortran (CAF)	Advances in the industry-leading Fortran Compilers with new support for scalable parallelism on nodes and clusters (cluster support available separately with Intel® Cluster Studio 2011); Fortran standards support
Memory, threading, and security analysis tools in one package	Enhances developer productivity and efficiencies by simplifying and speeding the process of detecting difficult-to-find coding errors
Updated performance libraries	Multicore performance for common math and data processing tasks, with a simple linking with these automatically parallel libraries
Updated performance profiler	Several ease-of-use enhancements, deeper microarchitectural insights, enhanced GUI, and quicker, more robust performance

Figure 11

Intel® Cluster Studio
Distributed Performance

Contains:

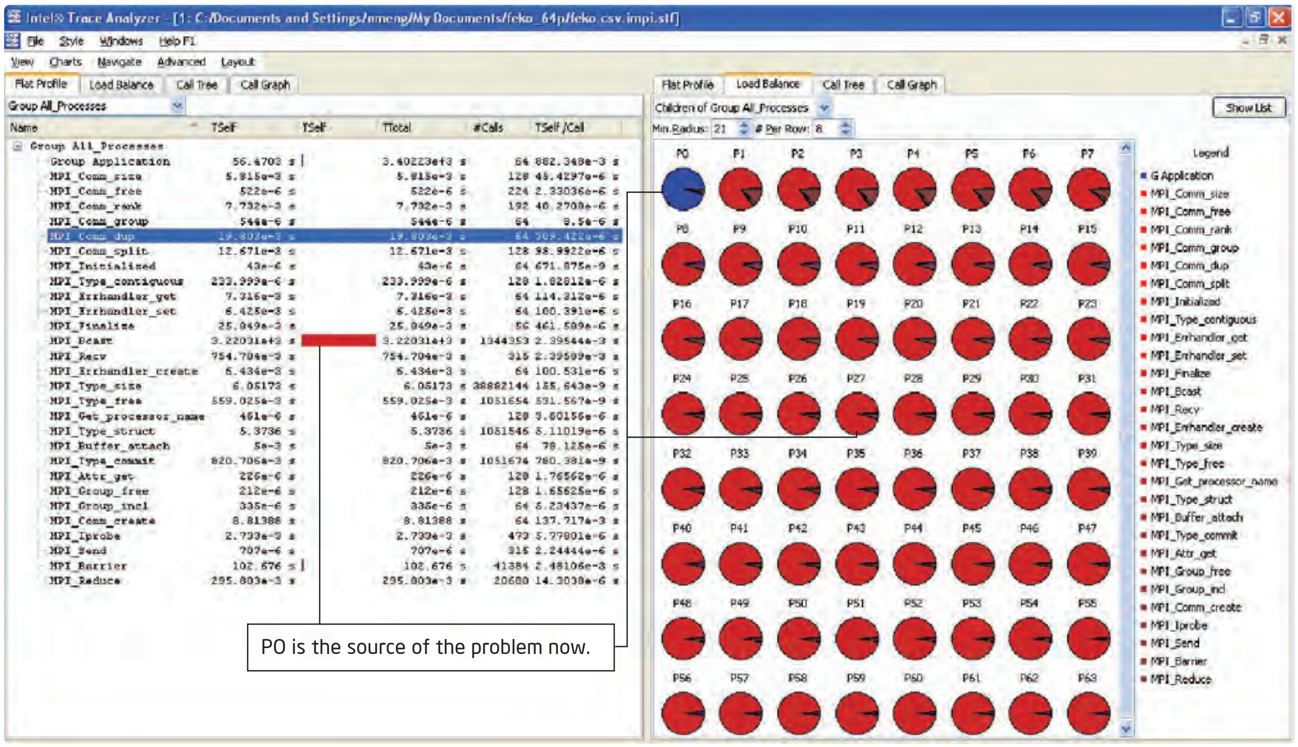
- Intel® Composer XE Compiler and Libraries
- Intel® MPI Library
- Intel® Trace Analyzer and Collector

Increase Performance and Scalability of HPC Cluster Computing

Intel® Cluster Studio 2011 sets a new standard in distributed parallelism on Intel architecture-based clusters. This premier tool suite provides development flexibility for enabling MPI-based application performance for highly parallel shared-memory and cluster systems based on 32 and 64 Intel architectures. The newly architected Intel MPI library 4.0 is key to achieving these advantages by providing new levels of cluster scalability, improved interconnect support across many fabrics, faster on-node messaging, support for hybrid parallelization, and an application tuning capability that adjusts to the cluster and application structure. For the developer, the Intel Trace Analyzer and Collector 8.0 is enhanced with new features that accelerate the analysis and tuning cycle of MPI-based cluster applications. The latest Intel C/C++ and Fortran compiler technology, along with Intel MKL 10.3, Intel IPP 7.0, and Intel PBB (also sold as Intel® Composer XE), complements the suite to further optimize and parallelize application execution on each computing node. Co-array Fortran is supported on clusters in this package.

Along with Intel Cluster Ready (ICR), a program to define cluster architectures for increasing uptime, increasing productivity, and reducing total cost of ownership (TCO) for IA-based HPC clusters, Intel Cluster Studio 2011 makes it easy to code, debug, and optimize to gain higher scalability for MPI-based cluster applications, up to petascale, and also is the premier suite for developing and tuning hybrid-parallel codes that can mix MPI with multithreading paradigms such as OpenMP or Intel PBB.

Intel Cluster Studio 2011 provides an extensive software package containing Intel C/C++ Compilers and Intel® Fortran Compilers for all Intel architectures, plus all the Intel® Cluster Tools that help you develop, analyze, and optimize performance of parallel applications on Linux or Windows. By combining all the compilers and tools into one license package, Intel can provide single installation, interoperability, and support for the best-in-class cluster software tools.


Intel® Trace Analyzer - [1: C:/Documents and Settings/nmeng/My Documents/feko_64p/feko.csv.impi.stf]
File Style Windows Help F1
View Charts Navigate Advanced Layout
Flat Profile Load Balance Call Tree Call Graph
Group All_Processes
Children of Group All_Processes
Min.Radius: 21 # Per Row: 8
Show List
P0 P1 P2 P3 P4 P5 P6 P7 P8 P9 P10 P11 P12 P13 P14 P15 P16 P17 P18 P19 P20 P21 P22 P23 P24 P25 P26 P27 P28 P29 P30 P31 P32 P33 P34 P35 P36 P37 P38 P39 P40 P41 P42 P43 P44 P45 P46 P47 P48 P49 P50 P51 P52 P53 P54 P55 P56 P57 P58 P59 P60 P61 P62 P63
Legend
G Application
MPI_Comm_size
MPI_Comm_free
MPI_Comm_rank
MPI_Comm_group
MPI_Comm_dup
MPI_Comm_split
MPI_Initialized
MPI_Type_contiguous
MPI_Errhandler_get
MPI_Errhandler_set
MPI_Finalize
MPI_Bcast
MPI_Recv
MPI_Errhandler_create
MPI_Type_size
MPI_Type_free
MPI_Get_processor_name
MPI_Type_struct
MPI_Buffer_attach
MPI_Type_commit
MPI_Attr_get
MPI_Group_free
MPI_Group_incl
MPI_Comm_create
MPI_Iprobe
MPI_Send
MPI_Barrier
MPI_Reduce

Name	TSelf	TSelf	TTotal	#Calls	TSelf /Call
Group All_Processes					
Group Application	56.4703 s		3.40223e+3 s	64	882.348e-3 s
MPI_Comm_size	5.815e-3 s		5.815e-3 s	128	45.4297e-6 s
MPI_Comm_free	522e-6 s		522e-6 s	224	2.33036e-6 s
MPI_Comm_rank	7.732e-3 s		7.732e-3 s	192	40.2708e-6 s
MPI_Comm_group	544e-6 s		544e-6 s	64	8.5e-6 s
MPI_Comm_dup	19.803e-3 s		19.803e-3 s	64	309.422e-6 s
MPI_Comm_split	12.671e-3 s		12.671e-3 s	128	98.9922e-6 s
MPI_Initialized	43e-6 s		43e-6 s	64	671.875e-9 s
MPI_Type_contiguous	233.999e-6 s		233.999e-6 s	128	1.82812e-6 s
MPI_Errhandler_get	7.316e-3 s		7.316e-3 s	64	114.312e-6 s
MPI_Errhandler_set	6.425e-3 s		6.425e-3 s	64	100.391e-6 s
MPI_Finalize	25.849e-3 s		25.849e-3 s	56	461.589e-6 s
MPI_Bcast	3.22031e+3 s		3.22031e+3 s	1344353	2.39544e-3 s
MPI_Recv	754.704e-3 s		754.704e-3 s	315	2.39589e-3 s
MPI_Errhandler_create	6.434e-3 s		6.434e-3 s	64	100.531e-6 s
MPI_Type_size	6.05173 s		6.05173 s	38882144	155.643e-9 s
MPI_Type_free	559.025e-3 s		559.025e-3 s	1051654	531.567e-9 s
MPI_Get_processor_name	461e-6 s		461e-6 s	128	3.60156e-6 s
MPI_Type_struct	5.3736 s		5.3736 s	1051546	5.11019e-6 s
MPI_Buffer_attach	5e-3 s		5e-3 s	64	78.125e-6 s
MPI_Type_commit	820.706e-3 s		820.706e-3 s	1051674	780.381e-9 s
MPI_Attr_get	226e-6 s		226e-6 s	128	1.76562e-6 s
MPI_Group_free	212e-6 s		212e-6 s	128	1.65625e-6 s
MPI_Group_incl	335e-6 s		335e-6 s	64	5.23437e-6 s
MPI_Comm_create	8.81388 s		8.81388 s	64	137.717e-3 s
MPI_Iprobe	2.733e-3 s		2.733e-3 s	473	5.77801e-6 s
MPI_Send	707e-6 s		707e-6 s	315	2.24444e-6 s
MPI_Barrier	102.676 s		102.676 s	41384	2.48106e-3 s
MPI_Reduce	295.803e-3 s		295.803e-3 s	20680	14.3038e-6 s

PO is the source of the problem now.

Figure 12

Intel® Trace Analyzer and Collector

- Visualize and understand parallel application behavior
- Evaluate profiling statistics and load balancing
- Analyze performance of subroutines or code blocks
- Identify communication hotspots

Highlights of Intel® Cluster Studio 2011

- Scalability and High Performance: The interconnect-tuned and multicore-optimized Intel® MPI Library delivers application performance on thousands of Intel Architecture and compatible multicore processors.
- Built-in Optimization: Utilize optimizing compilers and libraries in Intel® Composer XE to get the most out of advanced processor technologies. The C/C++ optimizing compiler now includes Intel PBB, which expands the types of problems that can be solved more easily in parallel, and with increased reliability. For Fortran developers, it now offers Co-array Fortran (CAF) and additional support for the Fortran 2008 standard. Intel® compilers also deliver advanced vectorization support with SIMD pragmas.
- Ease of MPI Tuning: Intel® Trace Analyzer and Collector has been enhanced with new features that accelerate the analysis and tuning cycle of MPI-based cluster applications.
- Target Applications to Multiple Operating Systems: Leverage the same source code in Intel® compilers and libraries, which bring advanced optimizations to Windows and Linux.
- Intel® Cluster Ready Qualified: This program defines cluster architectures to increase uptime and productivity and reduce total cost of ownership (TCO) for IA-based HPC clusters.
- Compatibility and Support: Intel Cluster Studio offers excellent compatibility with leading development environments and compilers , while providing optimal support for multiple generations of Intel processors and compatibles. Intel offers broad support through its forums and Intel® Premier Support, which provides fast answers and covers all software updates for one year.

Feature	Benefit
Analysis tools for MPI developers load imbalance diagram; ideal Interconnect simulator	Enhanced developer productivity and efficiencies by simplifying and speeding the detection of errors and offering performance profiling of MPI messages.
Scalable Intel MPI Library with multirail IB support and Application Tuner	Scale to tens of thousands of cores with one of the most scalable and robust commercial MPI libraries in the industry. Ease-of-use with dynamic and configurable support across multiple cluster fabrics and multi-rail IB support
C/C++ Compilers with Intel® Parallel Building Blocks	Breakthrough in providing choice of parallelism for applications— process, task, data, vector— with mix and match for optimizing application performance on clusters of SMP nodes. C/C++ standards support
Fortran compilers with key Fortran 2008 standards support including co-array Fortran (CAF) on clusters (available on Linux now and Windows later)	Advances in the industry-leading Fortran compilers with new support for scalable parallelism on nodes and clusters. Fortran standards supported include key features in Fortran 2008, more complete Fortran 2003 support.
Updated performance libraries, Intel® MKL and Intel IPP	Multicore performance for common math and data processing tasks, with a simple linking with these automatically parallel libraries
Support for both Linux* and Windows* platforms	Development capability with the same set of tools on both Windows and Linux platforms for enhanced performance, productivity, and programmability

Figure 13

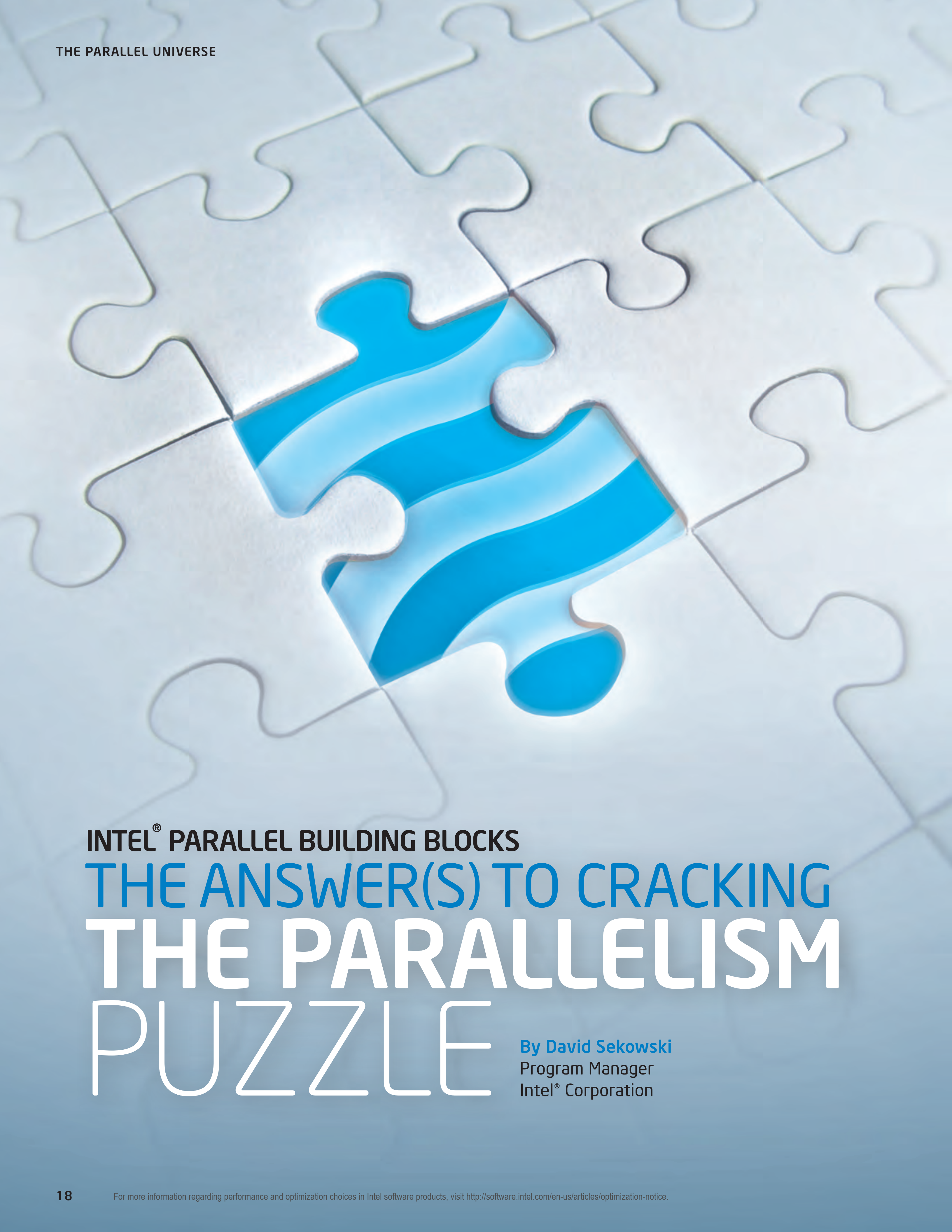
Summary

With the introduction of Intel Parallel Studio XE and Intel Cluster Studio, Intel is extending the reach of the next-generation Intel tools to Windows and Linux C/C++ and Fortran developers needing advanced performance for multicore today and forward scaling to manycore.

The Intel Parallel Studio XE 2011 bundle contains the latest versions of Intel C/C++ and Fortran compilers, Intel MKL and Intel IPP performance libraries, Intel PBB libraries, (Intel TBB, Intel ArBB [in beta], and Intel Cilk Plus), Intel Inspector XE correctness analyzer, and Intel VTune Amplifier XE performance profiler.

Intel Cluster Studio 2011 bundle contains the latest versions of Intel MPI Library, Intel Trace Analyzer and Collector, Intel C/C++ and Fortran compilers, Intel MKL and Intel IPP performance libraries, and Intel PBB libraries (Intel TBB, Intel ArBB [in beta], and Intel Cilk Plus).

For more information, please visit http://www.intel.com/software/products. □



INTEL® PARALLEL BUILDING BLOCKS

THE ANSWER(S) TO CRACKING THE PARALLELISM PUZZLE

By David Sekowski
Program Manager
Intel® Corporation

Editor's note:

Intel® Threading Building Blocks (Intel® TBB) has grown fantastically popular with C++ developers over the past five years. It has been ported to many platforms and used in many applications, including recently in well-known Adobe* products. This article introduces an expanded family of parallel models, with Intel TBB at the very center. The author introduces the complementary models that expand upon what Intel TBB can do in a compatible and complementary manner that makes Intel® Parallel Building Blocks (Intel® PBB) well worth understanding and using.

Motivation

As microprocessors transition from clock speed as the primary vehicle for performance gains to features such as multiple cores, it is increasingly important for developers to optimize their applications to take advantage of these platform capabilities. In the past, the same application would automatically perform better on a newer CPU due to increasingly higher clock speeds. However, when customers buy computers with the latest CPUs, they may not see a corresponding increase in applications that are written in serial or designed to take advantage of only one processing element. Therefore, hardware designers have to find other ways to deliver superior application performance, and they're turning to an old standby from high performance computing: parallel hardware platforms. This represents a new challenge for most software developers who haven't had much experience with writing parallel software.

Since Amdahl's Law was originally coined in 1967, high-performance computing experts have known a thing or two about the answer to this problem. Namely, that by putting more processors on the job we can reduce total application runtime through software parallelism. With the addition of Gustafson's Law in 1988, the upper limits on scalability implied by Amdahl's Law were effectively removed. Taken together these principles opened an alternative path to improving software performance in mainstream applications without increasing clock speed. Welcome to the era of multicore processors.

Today and in the future, cutting-edge applications will turn to parallelism to harness the profound power of dual-, quad-, and even-more-core processors found in most common mainstream computers. In his book *Only the Paranoid Survive,* Andy Grove talks about how strategic inflection points happen in business when a new technology has the ability to improve performance by an order of magnitude. It is at these inflection points that there is a possibility to revolutionize instead of "evolutionize" an industry. Multicore, and soon manycore, processors represent just such an opportunity to mainstream software application developers, if they can harness the added performance and functionality potential.

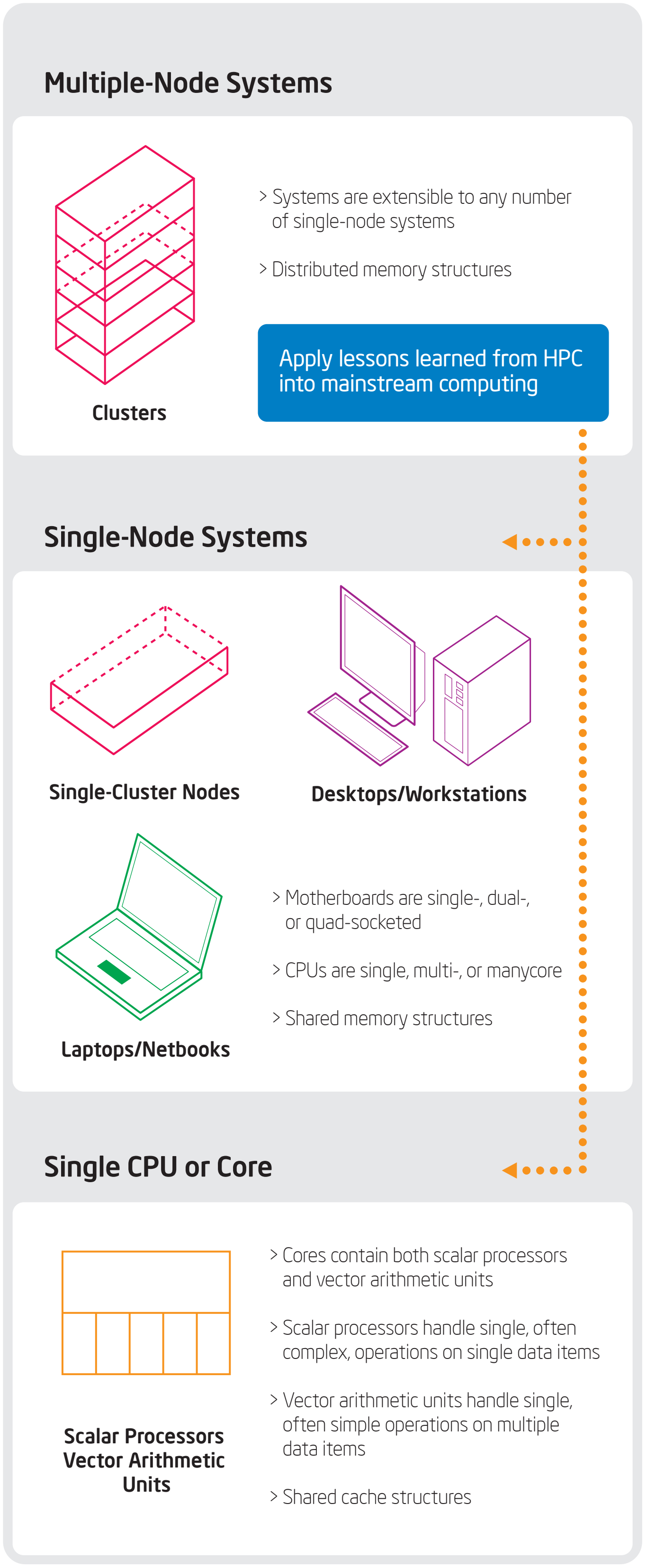

Multiple-Node Systems
Clusters
> Systems are extensible to any number of single-node systems
> Distributed memory structures
Apply lessons learned from HPC into mainstream computing
Single-Node Systems
Single-Cluster Nodes
Desktops/Workstations
Laptops/Netbooks
> Motherboards are single-, dual-, or quad-socketed
> CPUs are single, multi-, or manycore
> Shared memory structures
Single CPU or Core
Scalar Processors Vector Arithmetic Units
> Cores contain both scalar processors and vector arithmetic units
> Scalar processors handle single, often complex, operations on single data items
> Vector arithmetic units handle single, often simple operations on multiple data items
> Shared cache structures

Figure 1

Distributed and Shared Memory Systems

It is useful when talking about software parallelism to begin by talking about hardware platforms from the highest to the lowest performance. **Figure 1**. This is because we can apply lessons learned from the high end to the mainstream. High-performance computing uses massively parallel hardware and software platforms to solve some of the world's largest problems from climate change to decoding the human genome. These systems range from grid computers using idle cycles on widely dispersed systems over the Internet to clusters using message passing to communicate across various nodes located relatively close to each other (e.g., using the Intel® Message Passing Interface library (Intel® MPI) to synchronize information across cluster nodes). Both of these types of computing systems utilize distributed memory as compared to shared memory like that found in a single node within a distributed system, a workstation, a netbook, etc. With distributed-memory systems, there is a need for high-level coordination across nodes as well as parallelism within nodes. Normally an explicit message-passing model is used to coordinate multiple nodes. As those nodes have become more powerful, with multiple processors—each with multiple cores, multiple scalar processors, and vector arithmetic units—there has been a need to mix message passing across nodes with parallel software within each node.

Task Parallelism, Data Parallelism, and Vectorization

The widely accepted industry terms for node-level parallelism can be quite confusing, and often have different meanings depending on a variety of factors. For the sake of discussing how Intel is providing solutions for software parallelism, this article will define three key types of parallelism. **Figure 2**.

Task parallelism is the highest level of software parallelism. Tasking is generally needed for problems with irregular control structures that operate on irregular data sets. It allows a developer to break their application into logically distinct pieces, such as the render pipeline, AI, physics, and network I/O modules in a game engine. Each of these logical elements is assigned to a task or group of tasks to be completed concurrently. Other, more general forms of task parallelism patterns include message passing, tasking, eventing, and pipelining. However, these types of parallel code generally do not scale well with additional processing elements, since the number of logical parts of an application rarely grow over time, such that it is not possible to exploit Gustafson's Law. Nevertheless, task parallelism is still a vital first step to creating scalable software by reducing the serial portions of code in a given application.

Data parallelism is complementary to task parallelism. In fact, some of the most well-known and successful solutions for data parallel patterns are implemented using task parallel programming models. Data parallelism uses algorithms with regular control structures to operate on concurrent containers and other regular data structures. Much of the potential scalability of today's applications exists in such regular forms; examples include encoding audio or video. In addition, it is often easier to begin parallelizing a serial application by taking a serial control flow construct, such as a "for" loop, and turning it into a parallel for loop, before investigating the benefits that can be offered by rearranging the logical elements of the application with task parallelism. Examples of data parallelism patterns and algorithms include parallel "loops" (also known as "maps"), sorts, reductions, and scans.

Vectorization: Both task and data parallelism are ways a developer can spread application work across multiple processing elements like multiple cores or processors. Vectorization is a subset of data parallelism that allows you to take advantage of the vector

Parallelism Types	Memory Structure	Control Structure	Algorithm Types	
Task Parallelism	Distributed Shared	Irregular	Message Passing Tasks	Events Pipelines
Data Parallelism	Shared	Regular	Loops Sorts Reductions	Trees Graphs Lists
Vectorization	Shared (Cache)	Regular	Simple Array & Vector Operations	Elemental Functions SIMD

Figure 2

arithmetic units within most modern microprocessors. Vectorization, or Single Instruction Multiple Data (SIMD) as it is sometimes called, allows a developer to perform a simple operation on multiple pieces of data at the same time. Adding multiple arrays of vectors, for instance, can be easily and quickly performed by a vector arithmetic unit. When some published results show performance of math kernels in scientific, financial, and other computations, they are highlighting the use of such vector arithmetic units by mathematical libraries. Applications that spend much of their runtime performing relatively simple data parallel operations can usually benefit greatly from vectorization.

Compilers & Libraries

There are a number of ways that computer science researchers have found to make implementing software parallelism easier. There are entirely new languages, extensions for current languages, automatic compiler features, and runtime libraries comprised of low-level constructs in current languages and operating systems. Intel is a leader in researching and providing software products supporting each of these methods. Today, Intel uses two primary methods to help developers write parallel code: (1) through the use of compiler features and language extensions, and (2) through libraries. **Figure 3**.

When considering whether to use a language extension or a library implementation for the user's specific application, it is important to consider the potential benefits of each solution. Language extensions are enabled by compilers that can offer many benefits like lighter weight scheduling, and, therefore, better performance for fine-grain parallelism. Compilers also offer greater levels of abstraction that can make it easier to implement parallelism. These benefits may come at the cost of control, generality, and applicability. Depending on the exact design of the language extensions, the developer may not be able to explicitly control the implementation of parallelism in their code because they are only providing hints via keywords or pragmas to the compiler.

In contrast, libraries can work with existing compilers and infrastructure, but may have to use a less standard and less compact syntax to expose their features. Library-based solutions may offer greater control, more comprehensive feature sets, and better performance for coarse-grain parallelism, although they generally introduce additional overhead compared to compiler-based solutions. However, both compilers and libraries are effective and symbiotic in accessing and expressing parallelism.

Finally, the implementation of parallelism in both compiler extensions and libraries utilize low-level constructs available in the operating system like OS threads and locking mechanisms. Therefore, it should be obvious that on any particular machine, a savvy developer can always match or exceed the per-thread performance and system-level scalability offered by these abstractions given the time, expertise, and inclination. On the other hand, coding directly to low-level mechanisms instead of high-level abstractions may limit the portability and future scaling of an application. Intel offers a host of high-level abstractions that help developers overcome these limitations.

Parallelism Model Implementations

Comparisons	Language Extensions	Libraries
Compiler	Dependent	Independent
Standards Compliance	Can be published as Standards	Generally adhere to Standards
Portability	Less	More
Ease of Use	More	Less
User Control	Less	More
Parallelism Grain Size	Fine	Coarse

Figure 3

Pragma

A pragma is a compiler directive, essentially a "hint" embedded in the source code to indicate some direction to the compiler. If the compiler understands the hint, it can perform a special compilation. If it doesn't, it simply ignores it and does not generate a warning.

Intel® Parallel Building Blocks		
Selection Criteria		**Value Propositions**
Abstract - models must reduce the need to write parallelism using OS-dependent threads and synchronization primitives	**Supported** - models must work with Intel tools like Parallel Amplifier for performance tuning and Parallel Inspector for debugging	**Usable** - models provide easy to implement and test abstractions to utilize all available hardware parallelism
Interoperable - models must be able to coexist within an application andreliably exchange data	**Composable** - models must be able to be nested and otherwise combined with reliable and performant behaviors	**Reliable** - models are not prone to common multi-threading errors and they can be mixed and matched in interesting and useful ways
Performant - models must provide sufficient per-thread performance to productivity	**Scalable** - models must provide additional performance scaling when adding processing elements	**Future-Proof** - models provide automatic forward scaling to more processing elements along with sufficient per-thread performance
Open - models should work on multiple platforms where applicable so developers can use them anywhere	**Standard** - models should provide published standards where applicable so others can interoperate	**Portable** - models can be used on multiple OS, HW, IDE, and compiler platforms with flexible licensing

Figure 4

Intel's Family of Parallel Models

Understanding the different types of parallelism available to a developer and the ways in which they can utilize abstractions to create parallel applications lays a framework for evaluating parallel models. Intel's family of parallel models supports a smorgasbord of options for developers. These include native threads, auto-vectorization, auto-parallelization, OpenMP, OpenCL, and many others. However, there is no universal solution. Each of these solutions has drawbacks or does not address all of the types of parallelism that developers need to be successful.

For example, OpenMP is ideal for Fortran and C applications that are meant to fully utilize a hardware system, complete some work, and return the answer. But because of this behavior of assuming control over the entire hardware system, OpenMP does not compose well with itself nor interoperate with other parallelism methods by default. In high-performance computing, OpenMP is a great solution for problems that are so large they can use all available resources fully to solve a problem, but has limited applicability in common applications running on a personal laptop such as Web browsers, media players, and email clients. That being said, Intel has been and will continue to provide industry-leading support for OpenMP in our compilers.

Intel® Parallel Building Blocks

In response to these concerns when applying parallelism to the average application, Intel has made it easier to utilize task and data parallelism through Intel TBB. Intel TBB is a C++ template library with a broad range of features to specify and execute both task and data parallelism. It uses dynamic task scheduling, scalable memory allocation, parallel algorithms and data structures, synchronization primitives, and portable threads to offer a composable and interoperable solution for node-level parallelism. This generality is afforded by additional overhead compared to compiler-based models and sufficiently complex APIs. There are some capabilities that Intel TBB does not support by design: (1) utilizing vector arithmetic units within modern processors and (2) automatically target Intel's manycore co-processors.

These technical limitations motivated creation of the two newest members of Intel's family of parallel models: Intel® Cilk Plus and Intel® Array Building Blocks (Intel® ArBB). **Figure 5**.

Intel Cilk Plus is comprised of C and C++ language extensions implemented in the Intel® C/C++ Compiler. If you are new to parallel programming, this is the easiest way to get started. The extensions include three simple keywords to expose both task and data parallelism, as well as an easy syntax to explicitly vectorize portions of an application. It has the additional benefits of serial semantics and deterministic output.

Intel® TBB	Free open source version	www.threadingbuildingblocks.org
	Paid commercial version	www.threadingbuildingblocks.com
Intel® Cilk™ Plus	General information	http://cilk.com
	Intel® Cilk++ SDK for Microsoft C++ compiler users on Windows* and GCC compiler users on Linux*	http://software.intel.com/en-us/articles/intel-cilk/)
Intel® ArBB	General information	http://intel.com/go/ArBB
	Beta download	http://software.intel.com/en-us/articles/intel-array-building-blocks

Figure 5

Intel ArBB, formerly Intel® Ct Technology, is a C++ library like Intel TBB, which offers highly performant and scalable data parallelism and vectorization. It utilizes a JIT, or just-in-time, compiler to dynamically optimize for any given target heterogeneous hardware platform, including both manycore processors and vector arithmetic units. It can generate highly optimized machine code on the fly to take best advantage of the multiple processors, cores, and vector arithmetic units available on both CPUs and accelerators, which may comprise a distributed memory system. By using dynamic code generation, it can also overcome modularity overhead of C++. For instance, it can support virtual functions without their runtime cost. To learn more about Intel ArBB, refer to the next article. These three models for parallelism (Intel TBB, Intel Cilk Plus, and Intel ArBB) all share a common infrastructure. Each individual model also adheres to strict selection criteria that guarantee a number of compelling value propositions by default. **Figure 5**. These models are complementary and each provides unique value in particular application contexts. In combination, they form a single comprehensive solution for task parallelism, data parallelism, and vectorization, with interfaces implemented using both language extensions and libraries. Together, they provide a unified solution to parallelism and are available as part of Intel® Parallel Studio 2011 as Intel® Parallel Building Blocks (Intel® PBB). Interested developers can get started with Intel PBB today by referring to the above links. □

"Intel ArBB, formerly Intel® Ct Technology, is a C++ library like Intel TBB, which offers highly performant and scalable data parallelism and vectorization. "

INTEL® ARRAY BUILDING BLOCKS

By Michael McCool
Software Architect
Intel® Corporation

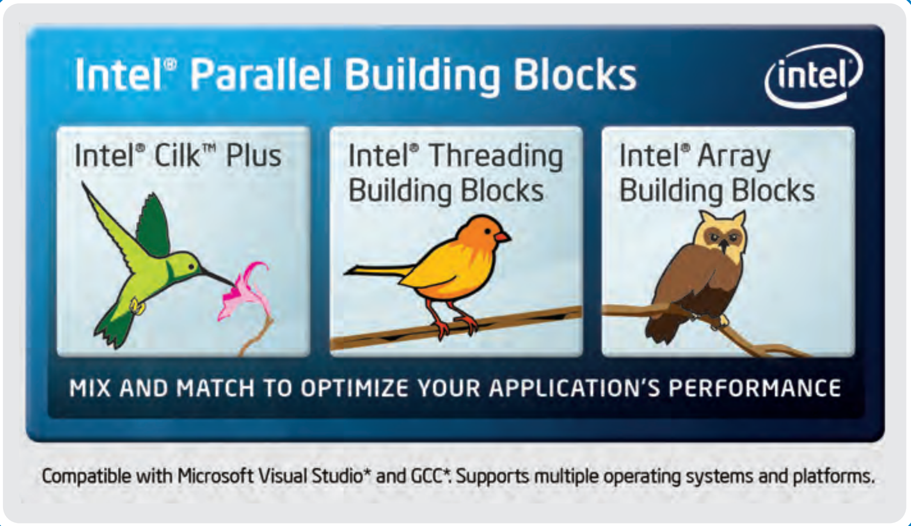
Intel® Array Building Blocks (Intel® ArBB) is a sophisticated and powerful platform for portable data-parallel software development. Intel ArBB will be available as a component of Intel® Parallel Building Blocks, along with several other tools and libraries for parallel programming.

Intel ArBB can be used to parallelize compute-intensive applications within a structured, deterministic-by-default framework. It also provides powerful runtime generic programming mechanisms, yet can be used with existing compilers. In particular, it has been verified to work with the Intel, Microsoft*, and gcc C++ compilers. Intel ArBB is currently in Beta, and feedback is appreciated; it can be downloaded today from http://intel.com/go/ArBB for either Windows or Linux.

Is Intel ArBB a language or a library? Yes—both at the same time. Intel ArBB is the answer to the following question: How can parallelism mechanisms in modern processor hardware, including vector SIMD instructions, be targeted in a portable, general way ***within*** existing programming languages? The answer is an ***embedded language***. Intel ArBB is a language extension implemented as an API. It has a library interface, but includes a capability for the dynamic generation and optimization of parallelized and vectorized machine language.

Modern processors include many mechanisms for increasing performance through parallelism: multiple cores, hyperthreading, superscalar instruction issue, pipelining, and single-instruction, multiple data (SIMD) vector instructions. The first two—multiple cores and hyperthreading—can be accessed through threads, although for efficiency, one may want to use lightweight tasks that share hardware threads. Instruction-level parallelism, such as superscalar instruction issue and pipelining, are invoked automatically by the processor, as long as the instruction stream avoids unnecessary data dependencies. However, the last form of parallelism, SIMD vector parallelism, can only be accessed by generating special instructions that explicitly invoke multiple operations at once: SIMD instructions. SIMD instructions perform the same operation on multiple components of a vector at once, so they are sometimes also called SIMD vector instructions.

Intel® Parallel Building Blocks (Intel® PBB) is a set of comprehensive parallel development models that supports multiple approaches to parallelism.


Intel® Parallel Building Blocks
intel
Intel® Cilk™ Plus
Intel® Threading Building Blocks
Intel® Array Building Blocks
MIX AND MATCH TO OPTIMIZE YOUR APPLICATION'S PERFORMANCE
Compatible with Microsoft Visual Studio* and GCC*. Supports multiple operating systems and platforms.

SIMD vector instructions are very powerful, and they are becoming more powerful over time. In current processors that support streaming SIMD extensions (SSE), four single-precision floating point instructions can be executed with a single SSE SIMD instruction. In next-generation AVX processors, the width of the SIMD instructions will double, so eight such operations can be executed at once. In the Intel® Many Integrated Core (MIC) architecture, the width doubles again, so 16 such operations can be executed at once.

Sign up for future issues | Share with a friend

"Intel® Parallel Building Blocks (Intel® PBB) actually includes three separate strategies for accessing vector operations in a portable manner. "

The theoretical peak floating-point performance of a processor is represented by the product of the number of cores, the width of the vector units, and the clock rate. While the clock rate is no longer scaling significantly, the number of cores and the SIMD vector width of each core continue to scale. Vectorization, that is, expressing computations using SIMD vector instructions, is essential to attain the peak performance of modern processors.

However, there are two problems. First, using SIMD vector units requires use of specific machine-language vector instructions. Second, different processors have different SIMD vector instruction extensions. The SSE, AVX, and MIC vector instructions are all different. While AVX machines can execute SSE instructions, this will not access the full performance potential of AVX processors. This latter issue is not so critical since current compiler technology does permit the generation of multiple code paths in a single binary. For example, when using the Intel® C++ Compiler, a single-source program can be compiled for both SSE and AVX machines, and the resulting program will use AVX code when possible. However, when using static compilers, developers still need to know in advance which set of processors they wish to target, and the problem remains: how is efficient vectorized code to be generated?

The traditional approach to supporting instruction set extensions is to modify the compiler to emit the new instructions, and then to recompile programs as necessary. However, for SIMD vector instructions this is not so easy. It is very difficult for a compiler to automatically identify serial structures in a program that can be mapped to SIMD vector instructions. It can be done sometimes, but it is better for the programmer to explictly indicate which operations in the program should use SIMD vector operations and how. This requires new constructs in the programming language that can be easily and reliably vectorized. Unfortunately, there is as yet no widely accepted machine-independent standard for specifying vectorization in C and C++.

Intel® Parallel Building Blocks (Intel® PBB) actually includes three separate strategies for accessing vector operations in a portable manner. The first strategy, which should not be overlooked, is to use a fixed-function library: Intel® Math Kernel Library (Intel® MKL) and Intel® Integrated Performance Primitives (Intel® IPP) include many mathematical operations that have already been vectorized. If the operation you need is part of these optimized libraries, that is often the best solution. If not, and you have to code the algorithm yourself, there are two other strategies available. First, you could use Intel® Cilk Plus, an extension to C and C++ that includes a notation to specify explicit vector operations on arrays. This notation is an extension to C/C++ available in the Intel C/C++ Compiler. The second general-purpose mechanism is Intel ArBB.

Intel ArBB is an embedded language, implemented as a C++ API, that in theory works with any ISO-standard C++ Compiler. It uses standard C++ mechanisms for its syntax, declaring types for collections of data and overloading operators so that operations can be expressed over those collections. In other words, it looks like a typical matrix-vector math library. However, there is a difference. In an ordinary library, the C/C++ Compiler generates the code statically. ArBB machine code is generated by the library itself, dynamically.

ArBB is very simple to use; we've provided a few examples below. To set the stage, however, we first need to discuss some basics. The ArBB C++ API defines both types and operations. Types include scalar types for floating point numbers, integers, and Booleans, as well as types for representing collections of these types and user-defined types based on them. The ArBB scalar types are used in place of the ordinary C++ types for floats and integers, and have names like f32 (for single-precision float), i32 (for signed 32-bit integers), and so forth. Using an ArBB scalar type indicates to ArBB that the corresponding machine language for operations on this type should be generated dynamically by ArBB and not statically by C++. There are also types to manage large collections of data. The simplest of these is called dense<T,D> and represents a contiguously stored (dense) multidimensional array with element type T and dimensionality D. The dimensionality is optional and defaults to 1. The element type T can be any ArBB scalar type or structures or classes with ArBB scalar types as elements.

There are two basic ways to specify parallel computations in ArBB: as sequences of operations over entire collections (vector mode), or as functions replicated over every element of a collection (elemental mode). Vector mode is the simplest: arithmetic operations on collections apply in parallel to the corresponding members of the collections. This works even if the element type is user-defined and the user has overloaded the operator themselves. For example, suppose we have four dense<f32> collections called A, B, C and D, all of the same size. Then the following expression will operate in parallel on all the elements of these collections:

```
A += (B/C) * D;
```

Note that in general when a collection appears on both the left and right side of an expression, ArBB generates a result "as if" all the inputs were read before any outputs are written. In practice, we have to put this expression inside a function and invoke it with a call operation. However, any sequence of parallel vector operations can be inside such a function:

```
void
doit(dense<f32>& A, dense<f32> B,
dense<f32> C, dense<f32> D)
{
 A += (B/C) * D;
}
 ...
call(doit)(A,B,C,D);
```

The way call actually works is that it calls the function to do it precisely once and ***observes*** (rather than actually performs) the sequence of ArBB type constructions, operations, and destructions generated by this function. It records this sequence, compiles it into optimized machine language, executes it (in parallel), and then caches it. The next time the same function is called, call does not invoke the C++ function again; it will just retrieve the internally generated machine code from its cache. For simple uses of Intel ArBB this is exactly what you want. In more advanced use cases, however, you may want to generate different versions of the operation from the same C++ function. For example, you can parameterize the sequence of Intel ArBB operations by ordinary C++ variables and control flow, and you can use this to generate variants of a computation. Managing this powerful mechanism for generic programming is enabled by another Intel ArBB type called a closure. A closure is an object that represents a captured Intel ArBB function; it is conceptually similar to a lambda function, but is dynamically generated. The return type of call is actually an appropriately typed closure. Another function, capture, is also available. It is similar to call in that it creates a closure, but it does not cache it, so it can be called repeatedly on the same C++ function to generate variants. Again, for simple uses of Intel ArBB explicit use of closures is not necessary, and you can just think of call as a straightforward function invocation.

BLOG highlights

Intel® Cilk™ Plus Specification and Runtime ABI Published for Free Download Now

JAMES REINDERS, DIRECTOR OF SOFTWARE DEVELOPMENT PRODUCTS

A Cilk Plus specification without an implementation would be noise. That is why we released a serious implementation first, followed shortly by a specification. Serious evaluation, production usage, and feedback are all possible as a result.

On November 2, we published the specification for the language and the runtime ABI for Intel Cilk Plus on cilk.com. This is an important step as we encourage adoption of these important capabilities in all compilers. We are in the early stages of discussions with others on how to best do this, and all agree that publishing a specification is a very important next step for the success of Cilk Plus.

We know that promoting a specification without an implementation would be a poor way to promote a language. Having an implementation that allows serious evaluation is a must. That is why we chose the order we did: implementation first, followed shortly by a specification.

We have full support for Cilk Plus in Intel's released compilers on Windows and Linux. These compilers and specifications build upon people, expertise, and technology acquired from Cilk Arts last year.

READ THE REST OF JAMES' POST:
Visit Go-Parallel.com

Browse other blogs exploring a range of related subjects at Go Parallel: Translating Multicore Power into Application Performance.

You can also write "elemental" functions over scalar Intel ArBB types:

```
void
kernel(f32& a, f32 b, f32 c, f32 d)
{
 a += (b/c)*d;
}
```

You can invoke elemental functions from inside a call by using the map operation. A map operation replicates the function over every element of the input containers.

```
void
doit(dense<f32>& A, dense<f32>
B, dense<f32> C, dense<f32> D)
{
  map(kernel)(A, B, C, D);
}
call(doit)(A,B,C,D);
```

It is also possible, from inside an elemental function, to access neighboring elements of the input. This makes it very easy to write stencil operations, such as convolutions. You can also pass in either an entire container ***or*** a single element to every argument of the map. Single-element arguments are replicated to match the size and shape of any containers used as arguments. For example, suppose we use the following:

```
void
doit(dense<f32>& A, f32 b, f32 c,
dense<f32> D)
{
 map(kernel)(A, b, c, D);
}
call(doit)(A,b,c,D);
```

with the same kernel function, but with the types of b and c matching the corresponding function argument exactly; in this case, f32. There will still be as many parallel instances of the kernel as there are elements in the collections A and D, but every instance will get a copy of the same value of b and c. In summary, call arguments need to match exactly, but map functions are polymorphic and any argument can either be a single element or a collection.

In addition to using these two basic patterns to express parallel operations, users of ArBB also have access to several collective operations that act on or take an entire container as an input. These operations can shift the contents of containers around, take cumulative sums (***prefix scans***), perform sets of reads and writes (known as ***scatters*** and ***gathers***), discard elements and pack the remainder into a contiguous sequence (known as ***pack***; the inverse is ***unpack***), or simply combine all elements into a single element. Combination of all the elements of a container into a single element is called a ***reduction***. For example, the following computes the dot product (sum of pairwise products) of two containers A and B:

```
f32 A_dot_B = add_reduce(A * B);
```

Elemental functions can also use control flow. During the process described above, ordinary C++ control flow is actually executed only when the function is "captured." This is incredibly useful for generic programming in order to specify variants, and to reduce the overhead of modularity and configuration. However, in order for control flow to be visible to ArBB and be compiled into the vector machine code generated by it, special macros need to be used to express "embeddable" control flow.

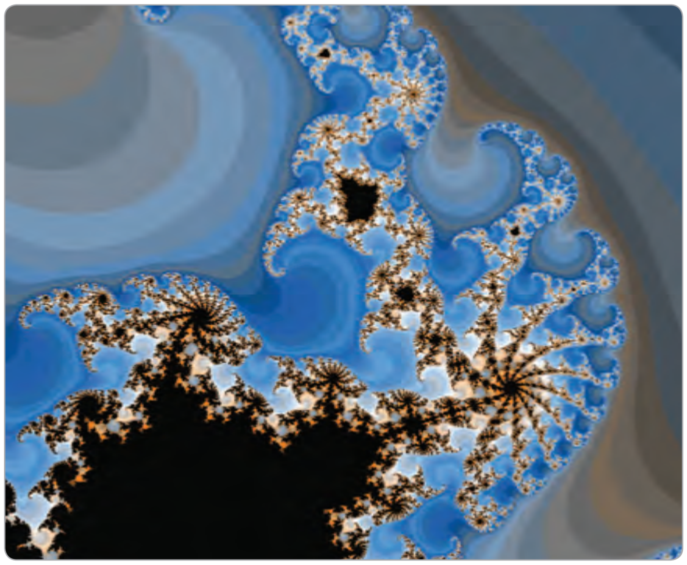


Figure 1

This is best shown by an example. The following program computes the Mandelbrot set, the famous fractal found by counting the number of iterations required for a complex quadtratic to diverge from a given starting point. Plotting this number of iterations over a region of the complex plane results in the above image. **Figure 1**.

An elemental function to compute a single pixel of this image is given by the following ArBB code:

```
int max_count = MAX_COUNT;
void mandel(i32& d, std::complex<f32> c) {
 i32 i;
 std::complex<f32> z = 0.0f;
 _for (i = 0, i < max_count, i++) {
   _if (abs(z) >= 2.0f) {
      _break;
   } _end_if;
   z = z*z + c;
  } _end_for;
  d = i;
}
```

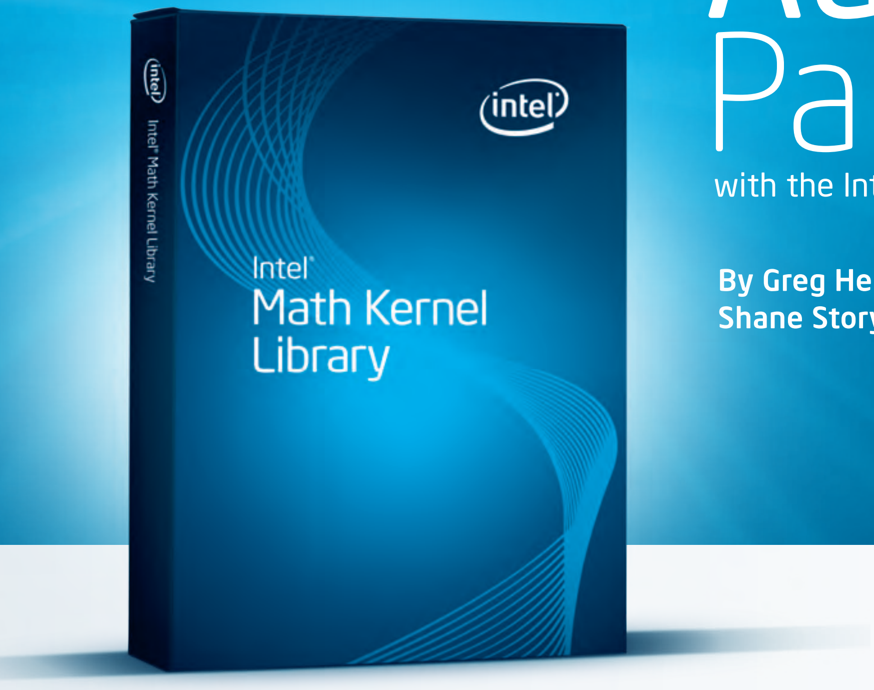
There are a few interesting things to note about this example. First, complex numbers can be expressed simply by using the std::complex type with an ArBB element type. This also works for user-defined types and, as mentioned above, for operator overloading on user types; such operator overloads even work for vector operations applied to collections of user types. Second, this function refers to a non-local C++ variable, max_count. In theory, we might want to capture this function with different values of this variable. This can be done with closures. If we use call, we will capture and "freeze" the value of this variable the first time we invoke this function. If we use capture, we can change the value and capture different versions and use closure objects to manage them. Finally, note that ArBB control flow has a few differences from C++ control flow: it uses a leading underscore, but also has closing keywords (such as _end_for) and the arguments to _for are separated by commas, not semicolons.

To actually get useful work done, we have to get data in and out of ArBB collections. This can be done in a variety of ways. ArBB actually supports an efficient STL-friendly interface based on iterators for sophisticated applications. However, the simplest way to get data in and out of ArBB is to simply associate an ArBB collection with a C++ array using bind, as follows. Note that we also have to use a helper call function "doit" to invoke the elemental function inside a map.

```
void
doit(dense<i32,2>& D,
dense<std::complex<f32>,2> P)
{
 map(mandel)(D,P);
}
dense<std::complex<f32>,2> pos;
bind(pos, c_pos, cols, rows)
dense<i32,2> dest;
bind(dest, c_dest, cols, rows)
call(doit)(dest, pos);
```

This article has presented a brief introduction to Intel Array Building Blocks. This system provides a portable mechanism for sophisticated and efficient data-parallel computation. In order to target vector instructions, while being processor and compiler independent, Intel ArBB includes a capability for dynamic code generation. This capability allows Intel ArBB to avoid the overhead of C++ in many cases, since the ArBB code generation is separate from that of the "host language," C++. Intel ArBB is a sophisticated and powerful system that provides access to a simple means to express efficient data parallel computations, and also supports unique and powerful mechanisms for generic, modular programming.

If you are interested in learning more about Intel ArBB or experimenting with it (again, it is currently in Beta, and feedback is appreciated) go to http://intel.com/go/ArBB.□



Automatic Parallelism
with the Intel® Math Kernel Library (Intel® MKL)

By Greg Henry, Intel® MKL Architect
Shane Story, Engineering Manager, Intel Corporation

The Intel® Math Kernel Library (Intel® MKL) provides software developers optimized and automatically parallelized mathematical library routines. Our routines are thread-safe and are applicable to many engineering, science, financial, and other applications. In this article, we provide an overview of the techniques Intel MKL uses to achieve the highest level of parallelism, as well as the hooks and knobs useful for getting the most from these threaded hotspots.

Intel MKL has a number of domains useful to developers who create applications for desktops, servers, and clusters. This includes industry standards like the Basic Linear Algebra Subroutines (BLAS) and the latest version of the Linear Algebra PACKage (LAPACK), as well as Fast Fourier Transforms (FFTs), Vector Math and Statistics Libraries (VML, VSL), a direct sparse solver (PARDISO), and sparse BLAS. To help lower the barrier to programming distributed memory architectures (clusters), Intel MKL includes ScaLAPACK, Parallel BLAS, and Cluster FFTs. Intel MKL is available on the latest versions of Linux, Windows, and Mac OS X. We have tuned code for Intel and AMD* hardware, including both IA-32 and Intel® 64 architectures.

The primary advantage of Intel MKL is that it makes the highest performance levels easily accessible to software developers. Within the software, we do automated dispatching to amortize the value of the underlying hardware features. This means users calling an industry-standard subroutine like DGEMM from the BLAS get performance improvements on different systems without having to re-link their applications. Intel MKL simply detects the hardware and dispatches code optimized for that processor, requiring no effort on behalf of the user. For example, the same application when linked with Intel MKL should run optimally on Intel® Core™2 Duo and Intel® Core™i7 processors because kernels optimized for both processors are already built in and dispatched during runtime.

Multicore machines are the latest trend in computing, offering high degrees of parallelism. While the potential for even higher performance is a natural side effect of an increased number of cores, the challenge of extracting that performance (and parallelism) falls squarely on the shoulders of the software developer. A library such as Intel MKL, where we have threaded most of the commonly used routines, is a simple and effective means of obtaining that parallelism.

Threading within Intel MKL is based on the industry standard OpenMP* specification. We thread in several of our domains: the direct sparse solver, LAPACK, BLAS, Sparse BLAS, VML, FFTs, and Cluster FFTs. For industry-standard components like LAPACK and the BLAS, our tuning goes beyond what one can find in the public domain. For example, in LAPACK we have added threading to some of the computational linear equation routines, orthogonal factorizations, singular value decompositions, and eigenproblems. In all cases, Intel MKL is thread-safe, so simultaneous execution of routines from multiple threads works correctly.

Users can set Intel MKL-specific variables such as MKL_NUM_ THREADS to specify the number of OpenMP* threads so as not to interfere with a user's other OpenMP* routines and environment variables. Intel MKL checks the Intel MKL-specific variables first; however, we are constrained by the underlying OpenMP environment. If neither the Intel MKL routines nor the OpenMP routines/variables are set, the underlying OpenMP default system will take precedence. If a developer builds an application for their own customers and wants control over the OpenMP environment (as opposed to allowing their users to experiment with environment variables), they can call Intel MKL threading service functions. Our threading service functions take precedence over our environment variables. Intel MKL works with the Intel® Compiler's OpenMP* libraries, in addition to those of Microsoft and GNU.

Developers can control the number of threads not only on a Intel MKL-wide level, but also on a domain-specific level with the MKL_ DOMAIN_NUM_THREADS environment variable or its corresponding service function, mkl_domain_set_num_threads(). For instance, if one wants all of Intel MKL to use two threads and the BLAS instead to use four, a user can set the variable with "MKL_ALL=2, MKL_BLAS=4." All environment variables are read only once in the course of a run. To change the behavior in the middle of a run requires calls to the service functions.

For developers building applications using a different threading model other than OpenMP, such as Intel® Cilk™ Plus Runtime Library, Intel® Thread Building Blocks, or pthreads in Linux, we suggest threading with the user's method of choice at the highest level, and either linking in the sequential Intel MKL, or setting MKL_NUM_ THREADS to "one" in the threaded version.

This usage model works well because threading is most effective when applied at the highest possible level—as it is in the current Intel MKL. For example, the original design and current public implementation of LAPACK depends on parallelism within the underlying BLAS routines. We found we could obtain better performance when we did threading at the LAPACK-level and called sequential BLAS, rather than relying on threading only in the BLAS. The critical observation is that the advantage of threading at a higher level increases with the number of threads. If we take the LAPACK routine DGETRF (double precision general matrix factorization via Gaussian elimination with partial row pivoting), fix a large matrix size on a manycore machine, and test the gap between threading just at the BLAS level versus the LAPACK routine level, the performance advantage increases as the number of threads increase.

Additionally, there are times when it is useful to take advantage of multiple levels and styles of parallelism, such as in a distributed memory cluster running MPI (Message Passing Interface) between the nodes. The Intel MKL benchmark MP LINPACK (which solves a cluster problem similar to DGETRF) uses hybrid MPI-OpenMP* parallelism for even greater performance. This is analogous to our previous statement regarding threading at the highest level. While running one MPI process per core is the most basic mechanism for parallelism on a cluster, running fewer MPI processes and putting OpenMP* calls into the code raises the threading level higher in the application and should yield performance gains.

Intel MKL depends on the underlying OpenMP* software to determine the number of threads. When the presence of MPI is detected and MPI has not been initialized for multithreading, Intel MKL will default to one thread. Likewise when called from inside an OpenMP parallel region, the default will be to one thread. If OpenMP gives us more threads than the number of physical cores (which might happen when HT is enabled), we will scale down the number of threads to match the number of physical cores. But there are times when our default choice may not be optimal, because of other aspects of the application we cannot detect. A user can set MKL_DYNAMIC to FALSE (its default is TRUE) or call mkl_set_dynamic() to try to override the number of threads we think will run optimally. Note that FFTs require both a setup and an execute stage, and the number of threads should be the same for both.

By taking advantage of the automatic parallelism Intel MKL provides, applications can get higher performance on modern multicore architectures. □

Web Bibliography:
[BLAS] http://www.netlib.org/blas/index.html
[LAPACK] http://www.netlib.org/lapack/index.html
[MKL] http://software.intel.com/en-us/intel-mkl/
[MPI] http://www.mcs.anl.gov/research/projects/mpi/
[MPI] http://www.intel.com/go/mpi
[OPENMP] www.openmp.org
[SCALAPACK] http://www.netlib.org/scalapack/index.html

When Print Statements and Timer Are Not Enough:

Making the Parallelism Investment More Effective

By Don Gunning, Nick Meng, and Paul Besl

As Intel® Architecture evolves, cluster software users are going to have to make their next investments in systems that employ greater amounts of parallelism.To support those investments, software developers will have to make changes to their software that can significantly impact performance. The changes usually involve many years of work and can lead to the implementation of mixed mode parallelism. In a small number of cases, the changes can be relatively minor. In either case, our experience indicates that the changes are not obvious.

We suggest the high-level features of the Intel® Cluster Toolkit Compiler Edition meet the above challenge, particularly, the new features in Intel® MPI 4.0 and the benefits of the Intel® Trace Analyzer and Collector.

In this article, we will see how real-world developers are applying Intel Trace Analyzer and Collector to find issues that would be undetectable with print statements and timer, correct those issues, and then deliver scaling performance with Intel® MPI 4.0 that is 30 to 50 percent higher than previously achieved (e.g., LSTC dyna is scaling past 1,500 cores; fluent is eclipsing 3,000 cores). Finally, we will suggest that a knowledgeable developer can deliver effective results in months that would take years to deliver with traditional tools.

Let's start by looking at how the application of the Intel Cluster Toolkit Compiler Edition 4.0 can be applied to real problems that happen to developers when they use the same software on new systems or apply the software to larger data sets.

We will show how Intel Trace Analyzer and Collector can be used to diagnose issues when previously estimated speedups are not achieved, and ensure that the quality of results is maintained when mixed mode parallelism is implemented to enable handling of larger data sets.

I just spent the money and the application runs slower

Frequently, software users will come to a software ISV and ask what the effect of a new Intel® multicore architecture will be on workloads. An ISV's user asked this question and got a very favorable estimate (yellow dash line in **Figure 1**). After having paid the money and installed the new Intel® 64 system, they saw the results (red line in **Figure 1**).

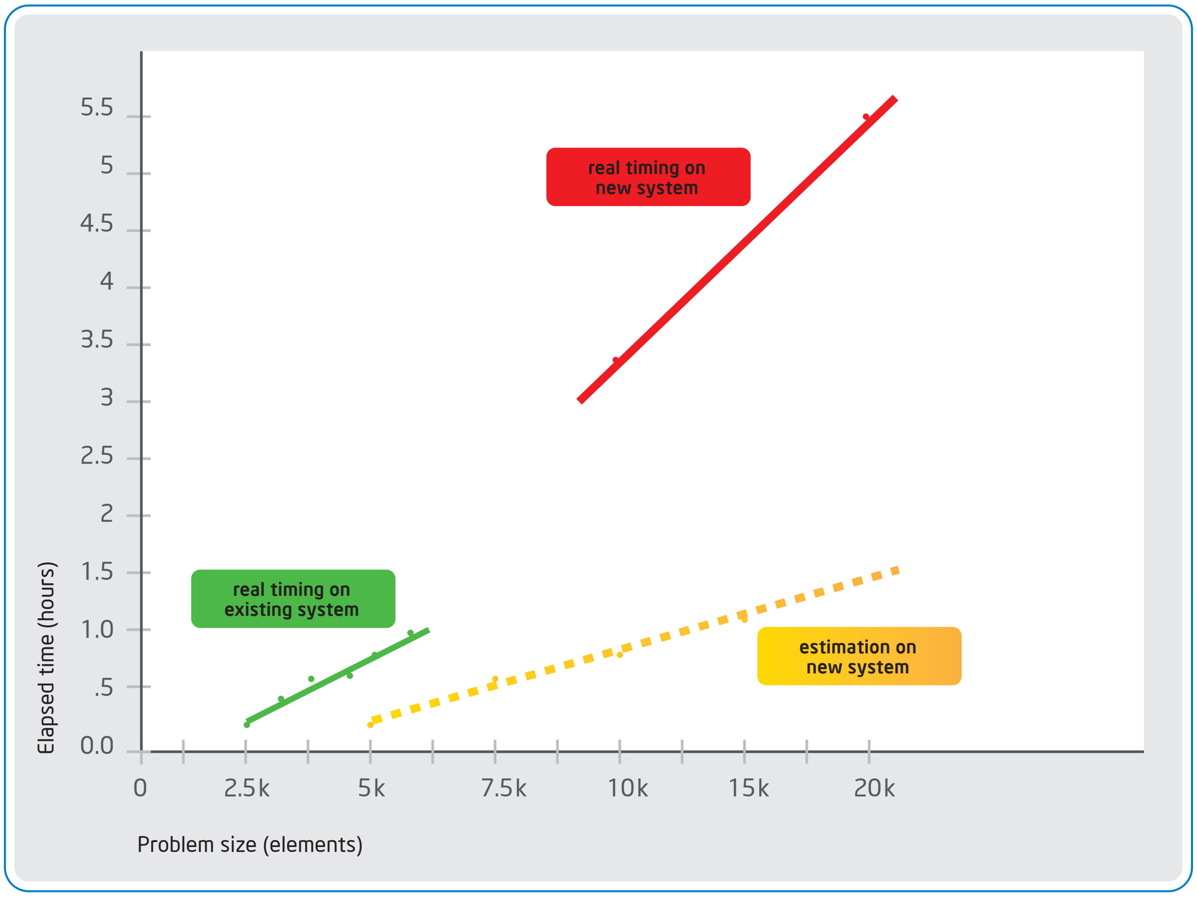


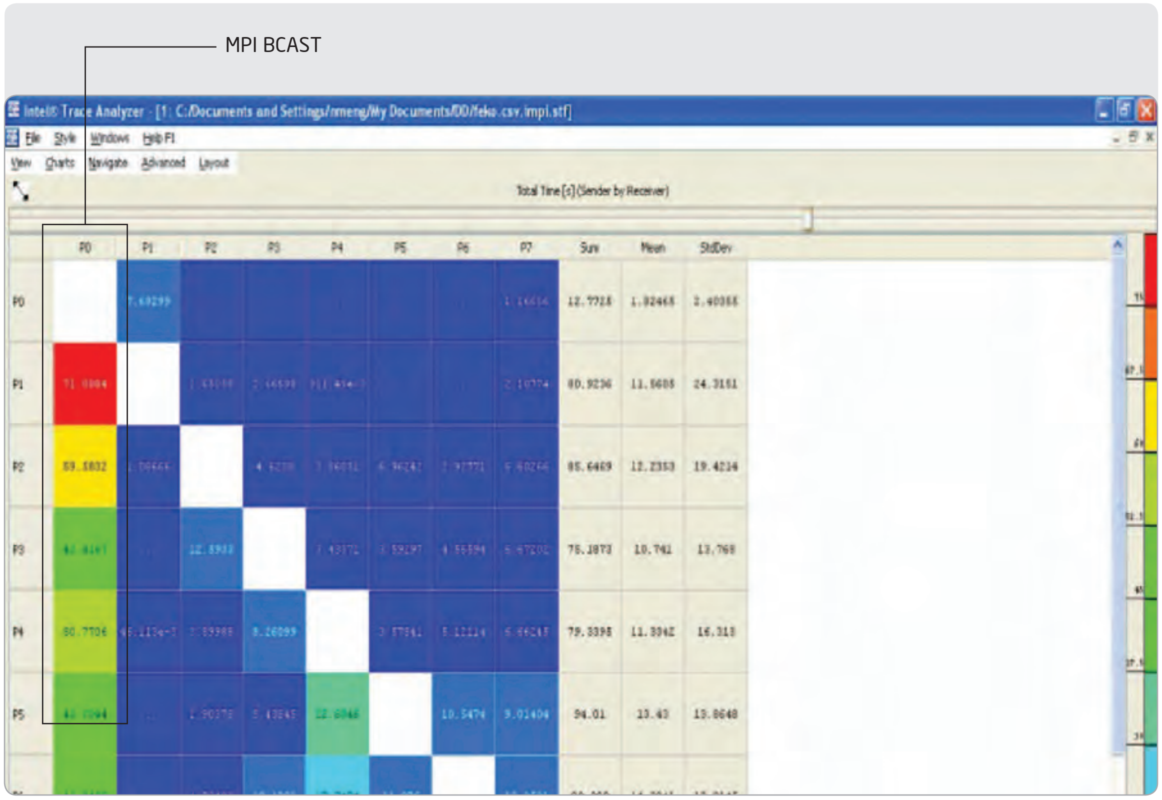
Figure 1. Practical testing and performance expectation

```
source /opt/intel/itac/8.0.1.001/bin/itacvars.sh
export LD_PRELOAD=/opt/intel/itac/8.0.1.001/slib/libVT.so
# 8p run
runexec small_model.pre -np 8 --mpi-options -trace --machines-file $PBS_NODEFILE
#256p run
runexec large_model.pre -np 256 --mpi-options -trace --machines-file $PBS_NODEFILE
```

Needless to say, there were some issues and diagnosis was necessary.

With Intel Trace Analyzer and Collector, the diagnosis was straightforward. Basically, you add "–trace" to your MPI command line or "–mpi-options –trace" in your run script file. Here is the run command with ITAC tool.

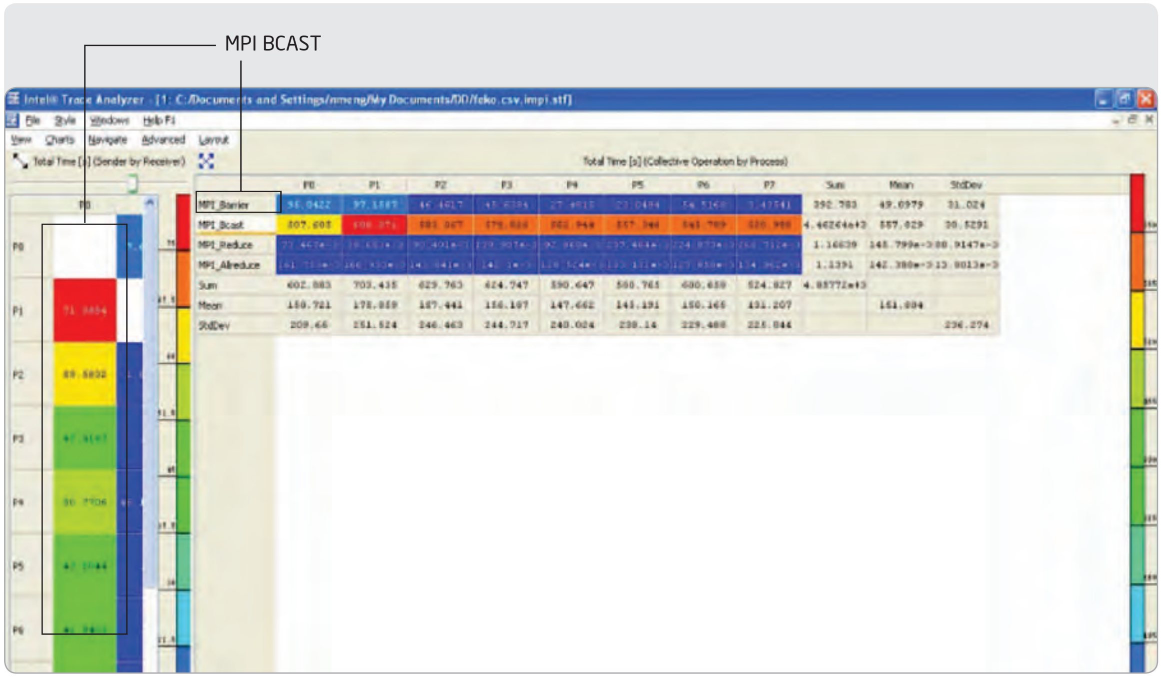
We tested two test cases with the Intel Trace Analyzer and Collector tool: a small test case with eight cores and a large test case with 256 cores. After we got STF trace files, we collected statistical data with the analyzer tool. **Figures 2** and **3** show the statistical data collected from the small test case in graph mode. In **Figure 2**, we can see the communication from P0 to P1-P7 is hot, and P0 to P1 is the hottest (see the outlined area).


MPI BCAST
Intel® Trace Analyzer - [1: C:/Documents and Settings/nmeng/My Documents/DD/feko.csv.impi.stf]
File Style Windows Help F1
View Charts Navigate Advanced Layout

Total Time [s] (Sender by Receiver)

	P0	P1	P2	P3	P4	P5	P6	P7	Sum	Mean	StdDev
P0		7.69299						1.16616	12.7728	1.82465	2.40355
P1	71.0084		1.53199	2.16598	711.434e-3			2.10774	80.9236	11.5605	24.3151
P2	69.5832	1.09666		4.6238	7.36031	6.96242	2.97771	6.60266	85.6469	12.2353	19.4214
P3	42.8167	[illegible]	12.8933		7.43172	3.59297	4.55594	5.67202	75.1873	10.741	13.768
P4	80.7706	65.113e-3	3.89988	8.26099		3.57541	5.12214	5.66245	79.3395	11.3342	16.313
P5	42.1094	[illegible]	1.90378	5.43545	12.6046		10.5474	9.01404	94.01	13.43	13.8648

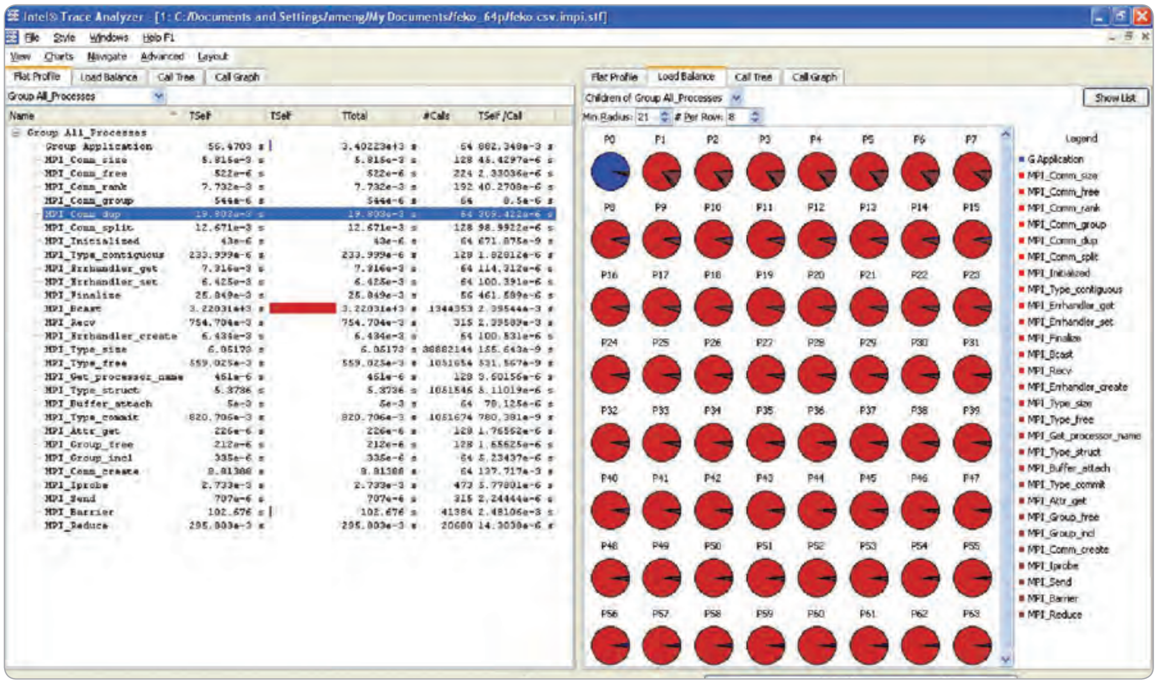
Figure 2. Total MPI collective function time in each MPI process generated by Intel® Trace Analyzer and Collector Tool. (Red indicates hot [busiest]; blue, cool [not so busy].)


MPI BCAST
Intel® Trace Analyzer - [1: C:/Documents and Settings/nmeng/My Documents/DD/feko.csv.impi.stf]
File Style Windows Help F1
View Charts Navigate Advanced Layout
Total Time [s] (Sender by Receiver)

Total Time [s] (Collective Operation by Process)

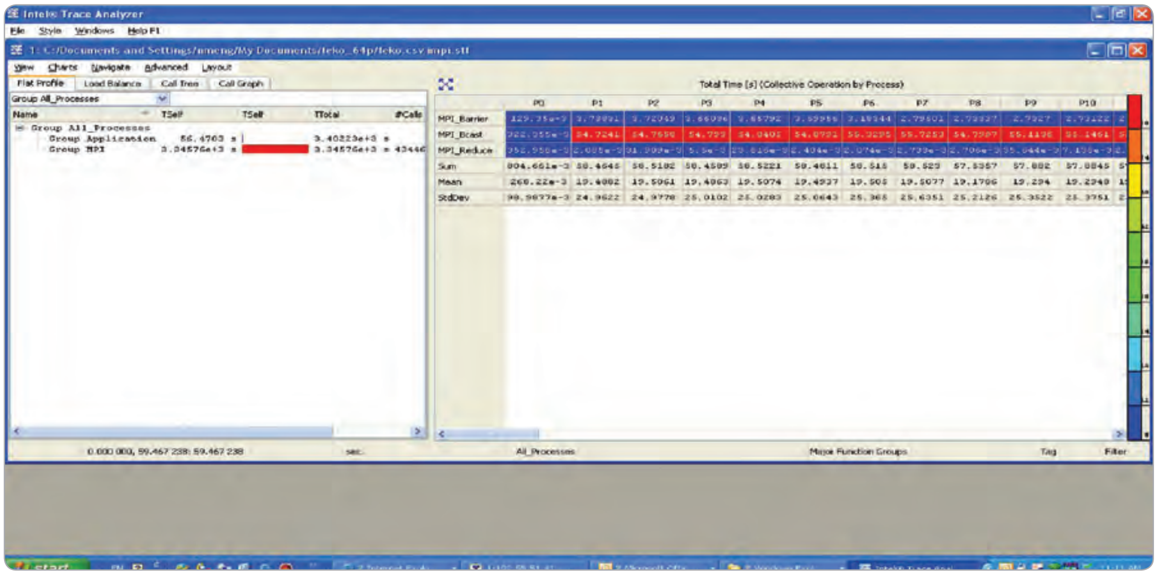
	P0	P1	P2	P3	P4	P5	P6	P7	Sum	Mean	StdDev
MPI_Barrier	56.0422	97.1587	46.4617	45.6394	27.4913	23.0494	56.5169	3.47541	392.783	49.0979	31.024
MPI_Bcast	507.605	598.271	583.067	579.828	562.946	557.346	541.789	520.980	4.46264e+3	557.829	30.5291
MPI_Reduce	77.4474e-3	19.6634e-3	90.491e-3	179.907e-3	92.9694e-3	237.464e-3	224.873e-3	260.712e-3	1.16639	145.799e-3	80.9147e-3
MPI_Allreduce	161.715e-3	166.332e-3	141.041e-3	142.1e-3	119.524e-3	133.111e-3	127.839e-3	134.362e-3	1.1391	142.388e-3	13.0013e-3
Sum	602.883	703.415	629.763	624.747	590.647	580.765	600.659	524.827	4.85772e+3		
Mean	150.721	175.859	157.441	156.187	147.662	145.191	150.165	131.207		151.804	
StdDev	209.66	251.524	246.463	244.717	240.024	238.14	229.488	225.844			236.274

Figure 3. Total time of MPI collective function in each MPI process generated by Intel® Trace Analyzer and Collector. (Red indicates hot; blue, cool.)



Name	TSelf	TSelf	TTotal	#Calls	TSelf /Call
Group All_Processes					
Group Application	56.4703 s		3.40223e+3 s	64	882.348e-3 s
MPI_Comm_size	5.816e-3 s		5.816e-3 s	128	45.4297e-6 s
MPI_Comm_free	522e-6 s		522e-6 s	224	2.33036e-6 s
MPI_Comm_rank	7.732e-3 s		7.732e-3 s	192	40.2708e-6 s
MPI_Comm_group	544e-6 s		544e-6 s	64	8.5e-6 s
MPI_Comm_dup	19.803e-3 s		19.803e-3 s	64	309.422e-6 s
MPI_Comm_split	12.671e-3 s		12.671e-3 s	128	98.9922e-6 s
MPI_Initialized	43e-6 s		43e-6 s	64	671.875e-9 s
MPI_Type_contiguous	233.999e-6 s		233.999e-6 s	128	1.82812e-6 s
MPI_Errhandler_get	7.316e-3 s		7.316e-3 s	64	114.312e-6 s
MPI_Errhandler_set	6.425e-3 s		6.425e-3 s	64	100.391e-6 s
MPI_Finalize	25.849e-3 s		25.849e-3 s	56	461.589e-6 s
MPI_Bcast	3.22031e+3 s		3.22031e+3 s	1344353	2.39544e-3 s
MPI_Recv	754.704e-3 s		754.704e-3 s	315	2.39589e-3 s
MPI_Errhandler_create	6.434e-3 s		6.434e-3 s	64	100.531e-6 s
MPI_Type_size	6.05173 s		6.05173 s	38882144	155.643e-9 s
MPI_Type_free	559.025e-3 s		559.025e-3 s	1051654	531.567e-9 s
MPI_Get_processor_name	461e-6 s		461e-6 s	128	3.60156e-6 s
MPI_Type_struct	5.3736 s		5.3736 s	1051546	5.11019e-6 s
MPI_Buffer_attach	5e-3 s		5e-3 s	64	78.125e-6 s
MPI_Type_commit	820.706e-3 s		820.706e-3 s	1051674	780.381e-9 s
MPI_Attr_get	226e-6 s		226e-6 s	128	1.76562e-6 s
MPI_Group_free	212e-6 s		212e-6 s	128	1.65625e-6 s
MPI_Group_incl	335e-6 s		335e-6 s	64	5.23437e-6 s
MPI_Comm_create	8.81388 s		8.81388 s	64	137.717e-3 s
MPI_Iprobe	2.733e-3 s		2.733e-3 s	473	5.77801e-6 s
MPI_Send	707e-6 s		707e-6 s	315	2.24444e-6 s
MPI_Barrier	102.676 s		102.676 s	41384	2.48106e-3 s
MPI_Reduce	295.803e-3 s		295.803e-3 s	20680	14.3038e-6 s

Figure 4. Load balance of large test case in pie mode on 256 cores generated by Intel® Trace Analyzer and Collector. (Red indicates MPI code percent execution time; blue indicates application percent execution time.)


Intel® Trace Analyzer
File Style Windows Help F1
1: C:/Documents and Settings/nmeng/My Documents/feko_64p/feko.csv.impi.stf
View Charts Navigate Advanced Layout
Flat Profile Load Balance Call Tree Call Graph
Group All_Processes
0.000 000, 59.467 238: 59.467 238
sec
All_Processes
Major Function Groups
Tag
Filter

Name	TSelf	TSelf	TTotal	#Calls
Group All_Processes				
Group Application	56.4703 s		3.40223e+3 s	
Group MPI	3.34576e+3 s		3.34576e+3 s	43446

Total Time [s] (Collective Operation by Process)

	P0	P1	P2	P3	P4	P5	P6	P7	P8	P9	P10
MPI_Barrier	129.35e-3	3.73891	3.72049	3.66096	3.65792	3.59956	3.18344	2.79501	2.73337	2.7327	2.73122
MPI_Bcast	222.255e-3	54.7241	54.7656	54.733	54.0405	54.0731	55.3295	55.7253	54.7987	55.4136	55.1461
MPI_Reduce	352.958e-3	2.085e-3	31.309e-3	5.5e-3	23.616e-3	2.404e-3	2.074e-3	2.739e-3	2.706e-3	35.044e-3	7.156e-3
Sum	804.661e-3	58.4645	58.5182	58.4509	58.5221	58.4811	58.515	58.523	57.5357	57.882	57.8845
Mean	268.22e-3	19.4882	19.5061	19.4863	19.5074	19.4937	19.505	19.5077	19.1786	19.294	19.2948
StdDev	98.9877e-3	24.9622	24.9778	25.0102	25.0283	25.0643	25.365	25.6351	25.2126	25.3522	25.3751

Figure 5. Profile data of large test case and total time of MPI collective function in each MPI process generated by Intel® Trace Analyzer and Collector.

In **Figure 3**, you can see MPI_Bcast is the hottest function in the test case (see the outlined area). We also did deep performance investigation with a large test case on 256 cores. **Figures 4, 5, 6,** and **7** show the same phenomena in the large test case.

Figure 4 displays the percentage of functions in pie mode: the blue area indicates computing cost of applications; the red area indicates the cost of MPI communications. You can see that the master MPI process pie is almost blue, and all slave MPI process pies are almost red. It means that there is a very serious load imbalance issue here. After we look at **Figures 6** and **7**, we find the serious load imbalance is from the MPI_BCAST function. **Figures 6** and **7** show the activities of MPI functions and

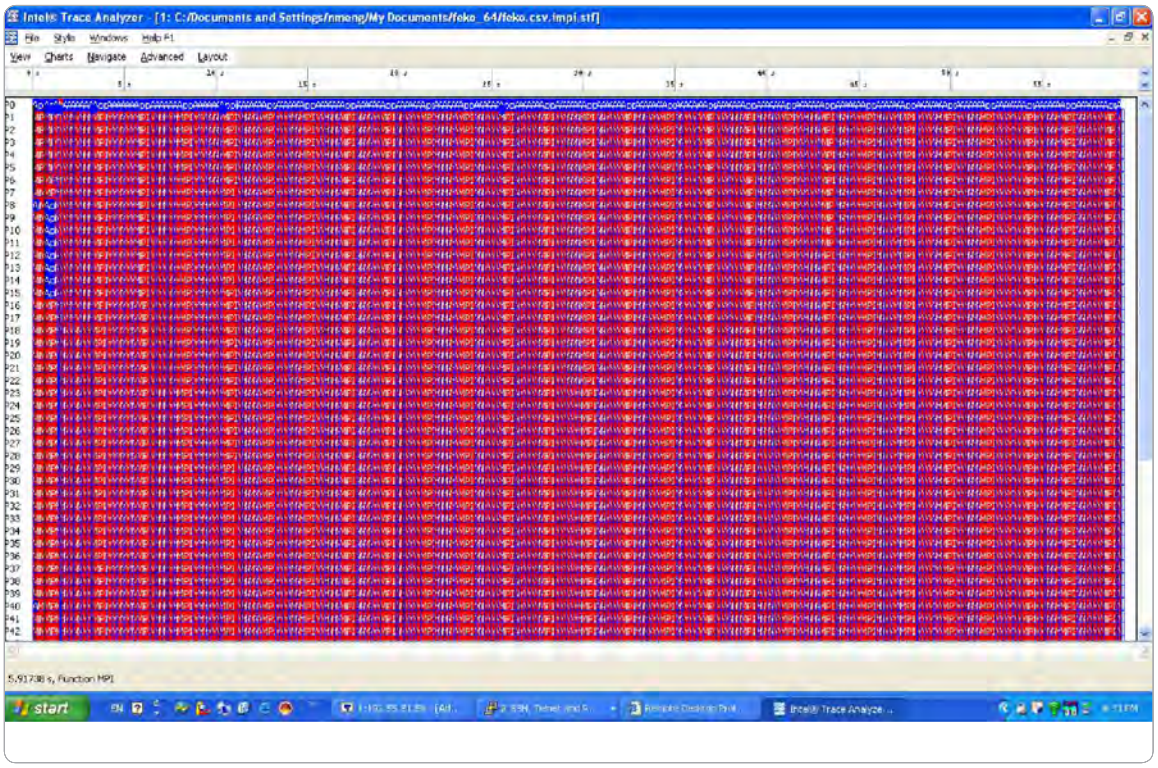

Intel® Trace Analyzer - [1: C:/Documents and Settings/nmeng/My Documents/feko_64/feko.csv.impi.stf]
5.91738 s, Function MPI

Figure 6. The activities of functions from 0 to 60 seconds generated by Intel® Trace Analyzer and Collector.

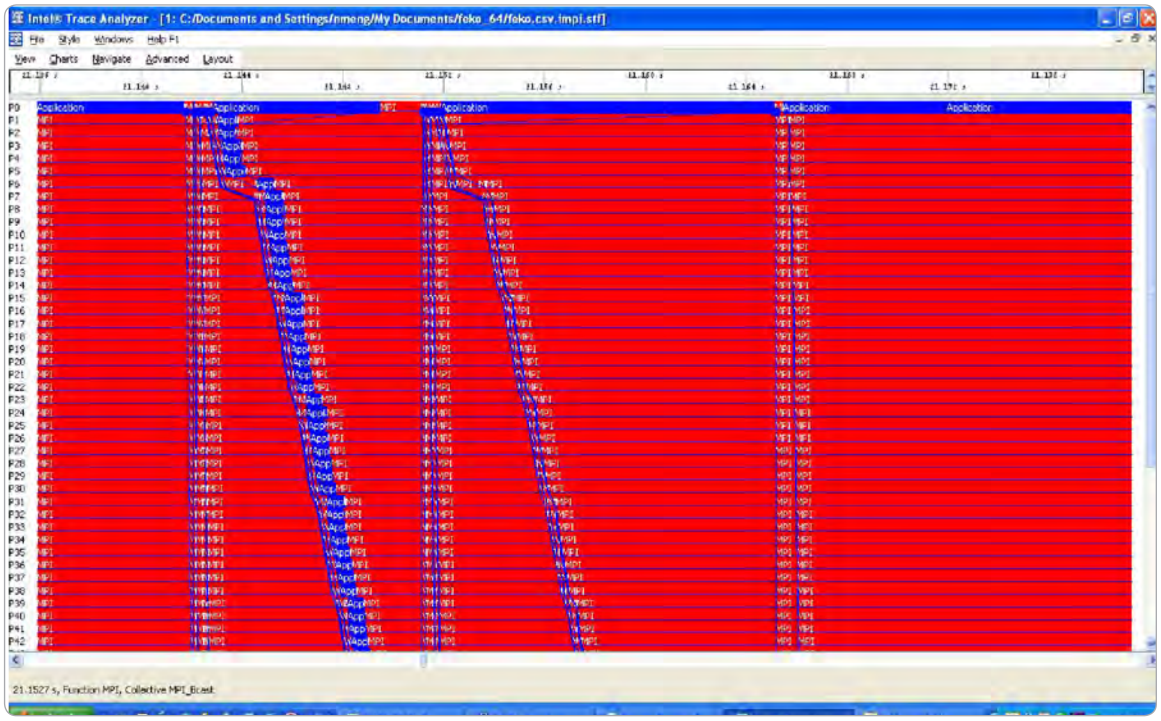

Intel® Trace Analyzer - [1: C:/Documents and Settings/nmeng/My Documents/feko_64/feko.csv.impi.stf]
21.1527 s, Function MPI, Collective MPI_Bcast

Figure 7. The activities of functions from 21.136 to 21.176 seconds generated by Intel® Trace Analyzer and Collector. (Red indicates MPI code; blue indicates application code. The horizontal axis represents time. The vertical axis represents separate MPI processes/ranks.)

application functions. The red area indicates MPI function activity; the blue area indicates the application function activity. Obviously, the large test case spent a lot time on MPI functions which is extremely abnormal.

In **Figure 4**, we noticed the master MPI process (pie in upper left corner) is blue and all slave MPI processes are almost red. Meanwhile, **Figures 5**, **6**, and **7** indicate that the load imbalance issue is clearly from a MPI _BCAST function. This means that the default (algorithm) setting of the MPI_BCAST function is not appropriate for this case. We need to select the right algorithm. We did some tests with a small cases and discovered the best setting for MPI_BCAST in this case: I_MPI_ADJUST_BCAST=4.

In this situation, the root cause of an unexpected problem was found quickly and the solution was easily implemented in large part due to the Intel Trace Analyzer and Collector's graphical data mining capability and the ISV developers' knowledge of the software. Finally, the user got reasonable performance on the new Intel 64 platform.

Now we will look at a more complex challenge.

Solving challenge tasks with mixed mode parallelism

Livermore Software Technology Corporation (LSTC) is continuously being challenged by users to deliver results faster on ever-increasing data set sizes. Further, in many cases the results must be consistent. LSTC offers shared and distributed memory versions of their software, with the distributed memory version offering better scalability than the shared memory version. The shared memory version uses OpenMP. The distributed memory version uses MPI.

We will now illustrate how the Intel Trace Analyzer and Collector was applied to LSTC DYNA to enable handling of significantly larger data sets on Intel® multicore architecture.

The challenge

LSTC supplies LS-DYNA, a general-purpose transient dynamic finite element program capable of simulating complex real-world problems. In very simple terms, LSTC sells crash simulation software that is used in manufacturing: automobile design, aerospace, consumer products, and bioengineering. To improve solution accuracy, the problem size is continually increasing with a non-trivial increase in computer time (e.g., solving a 10 million element problem can take 43 hours on a given cluster).

The challenge LSTC was faced with is the need for numerical consistency combined with limited network bandwidth, fixed node memory, and limited memory bandwidth which prevented LSTC users from scaling beyond a certain node count as problem size increased. This can significantly increase runtimes.

BLOG highlights

Condition Variable Support in Intel® Threading Building Blocks

WOOYOUNG KIM

One feature present in the proposed C++ standard specification (i.e., N3092) threading support library, which we began supporting since Intel® Threading Building Blocks (Intel® TBB) 3.0, is condition variable. As the C++1x proposal approaches the final approval, we expect using threads in conjunction with condition variables will become more popular.

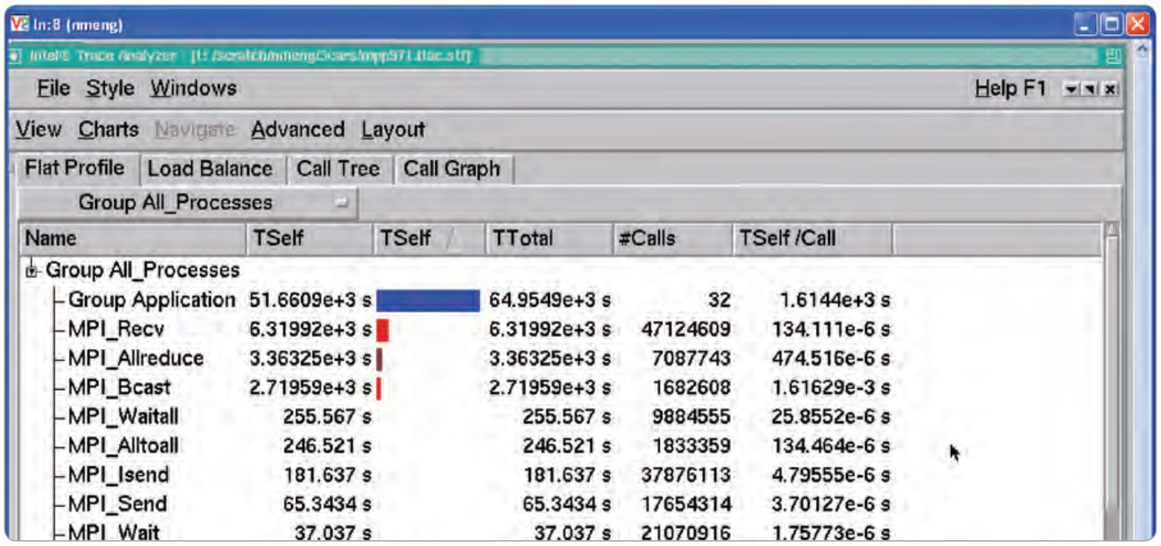
For example, Microsoft has already been supporting condition variables natively since Windows* Vista. Until Intel® TBB 3.0, Intel TBB used to provide only half of it (cf., std::thread – it used to be called tbb::tbb_thread). The following code example shows how to use the Intel TBB condition variable. For a Concise introduction to condition variables, see here or here.

```
using namespace std;
condition_variable my_condition;
tbb::mutex my_mtx;
bool present = false;

void producer() {
  unique_lock<tbb::mutex> ul( my_mtx );
  present = true;
  my_condition.notify_one();
}
void consumer() {
  while( !present ) {
  unique_lock<tbb::mutex> ul( my_mtx );
  my_condition.wait( ul );
  }
}
```

READ THE REST OF WOOYOUNG'S POST:
Visit **Go-Parallel.com**

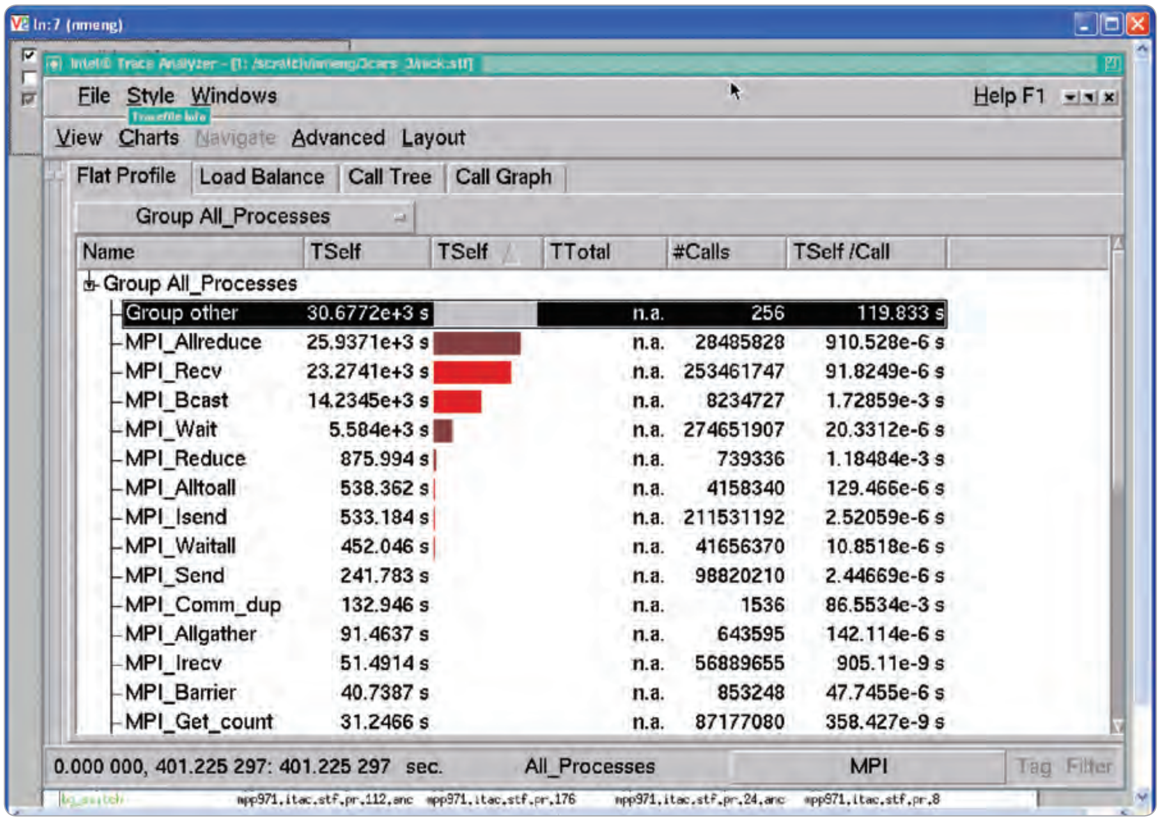
Browse other blogs exploring a range of related subjects at Go Parallel: Translating Multicore Power into Application Performance.

Sign up for future issues | Share with a friend


In:8 (nmeng)
Intel® Trace Analyzer
File Style Windows
Help F1
View Charts Navigate Advanced Layout
Flat Profile Load Balance Call Tree Call Graph
Group All_Processes

Name	TSelf	TSelf	TTotal	#Calls	TSelf /Call
Group All_Processes					
Group Application	51.6609e+3 s	🟦	64.9549e+3 s	32	1.6144e+3 s
MPI_Recv	6.31992e+3 s	🟥	6.31992e+3 s	47124609	134.111e-6 s
MPI_Allreduce	3.36325e+3 s	🟥	3.36325e+3 s	7087743	474.516e-6 s
MPI_Bcast	2.71959e+3 s	🟥	2.71959e+3 s	1682608	1.61629e-3 s
MPI_Waitall	255.567 s		255.567 s	9884555	25.8552e-6 s
MPI_Alltoall	246.521 s		246.521 s	1833359	134.464e-6 s
MPI_Isend	181.637 s		181.637 s	37876113	4.79555e-6 s
MPI_Send	65.3434 s		65.3434 s	17654314	3.70127e-6 s
MPI_Wait	37.037 s		37.037 s	21070916	1.75773e-6 s

Figure 8. Profile data of a customer model on four nodes generated by Intel® Trace Analyzer and Collector.



Name	TSelf	TSelf	TTotal	#Calls	TSelf /Call
Group All_Processes					
Group other	30.6772e+3 s		n.a.	256	119.833 s
MPI_Allreduce	25.9371e+3 s	🟥	n.a.	28485828	910.528e-6 s
MPI_Recv	23.2741e+3 s	🟥	n.a.	253461747	91.8249e-6 s
MPI_Bcast	14.2345e+3 s	🟥	n.a.	8234727	1.72859e-3 s
MPI_Wait	5.584e+3 s	🟥	n.a.	274651907	20.3312e-6 s
MPI_Reduce	875.994 s	🟥	n.a.	739336	1.18484e-3 s
MPI_Alltoall	538.362 s	🟥	n.a.	4158340	129.466e-6 s
MPI_Isend	533.184 s	🟥	n.a.	211531192	2.52059e-6 s
MPI_Waitall	452.046 s	🟥	n.a.	41656370	10.8518e-6 s
MPI_Send	241.783 s		n.a.	98820210	2.44669e-6 s
MPI_Comm_dup	132.946 s		n.a.	1536	86.5534e-3 s
MPI_Allgather	91.4637 s		n.a.	643595	142.114e-6 s
MPI_Irecv	51.4914 s		n.a.	56889655	905.11e-9 s
MPI_Barrier	40.7387 s		n.a.	853248	47.7455e-6 s
MPI_Get_count	31.2466 s		n.a.	87177080	358.427e-9 s

```
0.000 000, 401.225 297: 401.225 297   sec.      All_Processes        MPI        Tag  Filter
mpp971.itac.stf.pr.112.anc  mpp971.itac.stf.pr.176  mpp971.itac.stf.pr.24.anc  mpp971.itac.stf.pr.8
```

Figure 9. Profile data of a customer model on 32 nodes generated by Intel® Trace Analyzer and Collector.

Cluster Configuration	Intel® Xeon® 7560 1 node/w 32 core	Intel® Xeon® 5560 Cluster 8 nodes, 8 cores per node; Total: 64 cores
MPP (MPI) version elapsed time	44013s	18521s
Hybrid MPI and OpenMP version elapsed time	7047s	5541s
Speed Up	6.25	3.34

Figure 10. LSTC standard implicit model benchmark CYL1E6.

The search for a solution

With a large complex problem, LSTC in collaboration with Intel, applied Intel® Cluster Tools to introduce hybrid scaling. The following illustrates how Intel Trace Analyzer and Collector was used to discover issues that print statements and timer would never show. The main point is that these methods were applied on over a hundred routines and enabled solution discovery in months rather than years.

MPP DYNA performance

As the number of MPI processes increases, so does the number of sub-domains and communication costs. This can cause load imbalances and high communication overhead. Ultimately, it resulted in poor parallel efficiency.

Intel Trace Analyzer and Collector was used to quickly and efficiently pinpoint this aspect of the problem. **Figures 8** and **9** show that MPI collective functions are performing well on four node configuration, but becoming a performance inhibitor on a 32-node configuration.

As these screens demonstrate, what's needed is to combine OpenMP within one node (which engages all cores there), together with MPI cross nodes. We can reduce the number of MPI processes as much as possible. Then, we can reduce the overhead of MPI functions. This procedure was performed on over 100 routines to assess the change.

The solution

LSTC was able to combine the shared memory version of LS DYNA with the MPP DYNA to obtain HYBRID LS-DYNA. This combined parallelism version did the following:

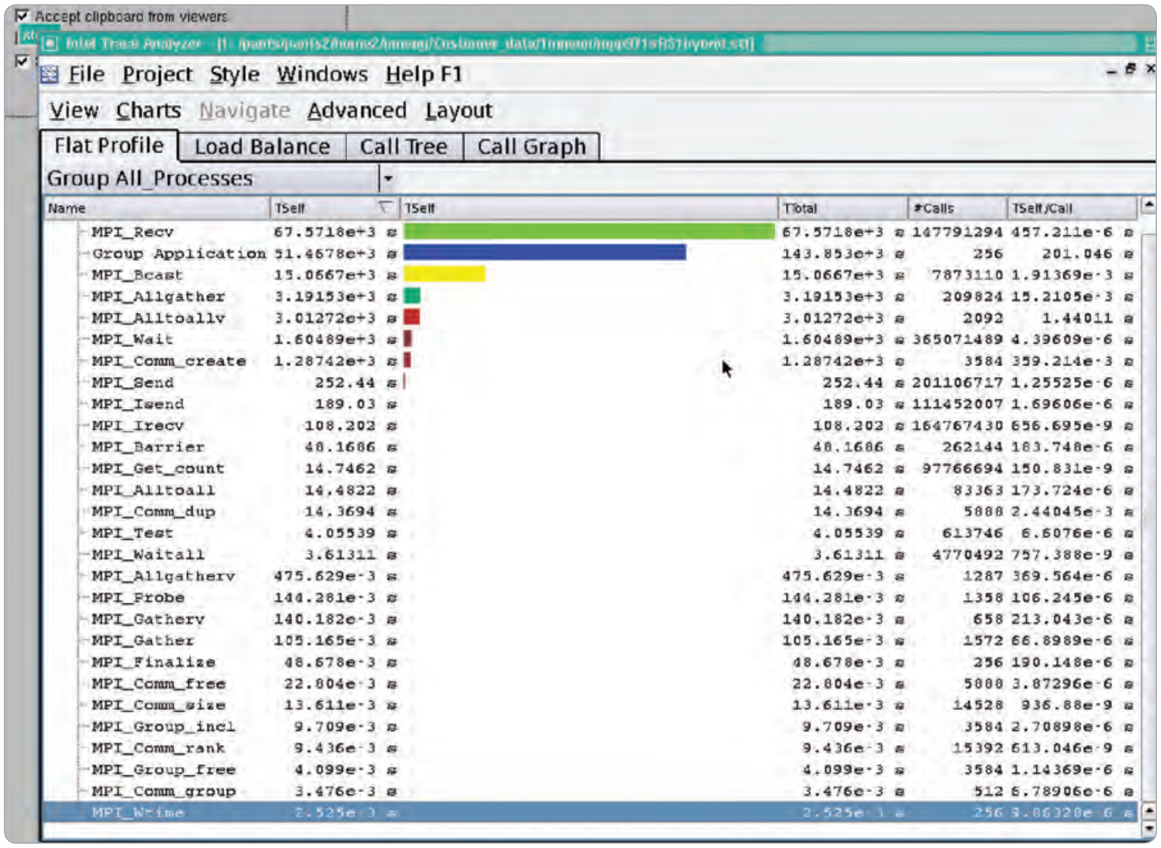
Maintained the numerically consistent results feature required by the user

> Once OpenMP* code and MPI code are combined into one code, we can take advantage of consistent features in the OpenMP code under certain conditions. As a result, customers get higher quality and numerically consistent results during the design cycle.

Increased the scalability of LS-DYNA and the effectiveness of Intel multicore node architecture

> As a result of reducing the overhead cost of MPI functions, customers can efficiently run their production model with more cores on Intel multicore node architecture. Especially for implicit solver users, they can take all cores and get maximum performance with fixed memory space and restricted I/O performance.

Figure10 illustrates some of the performance increases that the Intel Cluster Tools helped to achieve.


Accept clipboard from viewers
File Project Style Windows Help F1
View Charts Navigate Advanced Layout
Flat Profile Load Balance Call Tree Call Graph
Group All_Processes

Name	TSelf	TSelf	TTotal	#Calls	TSelf/Call
MPI_Recv	67.5718e+3 s		67.5718e+3 s	147791294	457.211e-6 s
Group Application	51.4678e+3 s		143.853e+3 s	256	201.046 s
MPI_Bcast	15.0667e+3 s		15.0667e+3 s	7873110	1.91369e-3 s
MPI_Allgather	3.19153e+3 s		3.19153e+3 s	209824	15.2105e-3 s
MPI_Alltoallv	3.01272e+3 s		3.01272e+3 s	2092	1.44011 s
MPI_Wait	1.60489e+3 s		1.60489e+3 s	365071489	4.39609e-6 s
MPI_Comm_create	1.28742e+3 s		1.28742e+3 s	3584	359.214e-3 s
MPI_Send	252.44 s		252.44 s	201106717	1.25525e-6 s
MPI_Isend	189.03 s		189.03 s	111452007	1.69606e-6 s
MPI_Irecv	108.202 s		108.202 s	164767430	656.695e-9 s
MPI_Barrier	48.1686 s		48.1686 s	262144	183.748e-6 s
MPI_Get_count	14.7462 s		14.7462 s	97766694	150.831e-9 s
MPI_Alltoall	14.4822 s		14.4822 s	83363	173.724e-6 s
MPI_Comm_dup	14.3694 s		14.3694 s	5888	2.44045e-3 s
MPI_Test	4.05539 s		4.05539 s	613746	6.6076e-6 s
MPI_Waitall	3.61311 s		3.61311 s	4770492	757.388e-9 s
MPI_Allgatherv	475.629e-3 s		475.629e-3 s	1287	369.564e-6 s
MPI_Probe	144.281e-3 s		144.281e-3 s	1358	106.245e-6 s
MPI_Gatherv	140.182e-3 s		140.182e-3 s	658	213.043e-6 s
MPI_Gather	105.165e-3 s		105.165e-3 s	1572	66.8989e-6 s
MPI_Finalize	48.678e-3 s		48.678e-3 s	256	190.148e-6 s
MPI_Comm_free	22.804e-3 s		22.804e-3 s	5888	3.87296e-6 s
MPI_Comm_size	13.611e-3 s		13.611e-3 s	14528	936.88e-9 s
MPI_Group_incl	9.709e-3 s		9.709e-3 s	3584	2.70898e-6 s
MPI_Comm_rank	9.436e-3 s		9.436e-3 s	15392	613.046e-9 s
MPI_Group_free	4.099e-3 s		4.099e-3 s	3584	1.14369e-6 s
MPI_Comm_group	3.476e-3 s		3.476e-3 s	512	6.78906e-6 s
MPI_Wtime	2.525e-3 s		2.525e-3 s	256	9.86328e-6 s

Figure 11. Profile data of the 1M model on 128 nodes generated by Intel® Trace Analyzer and Collector.

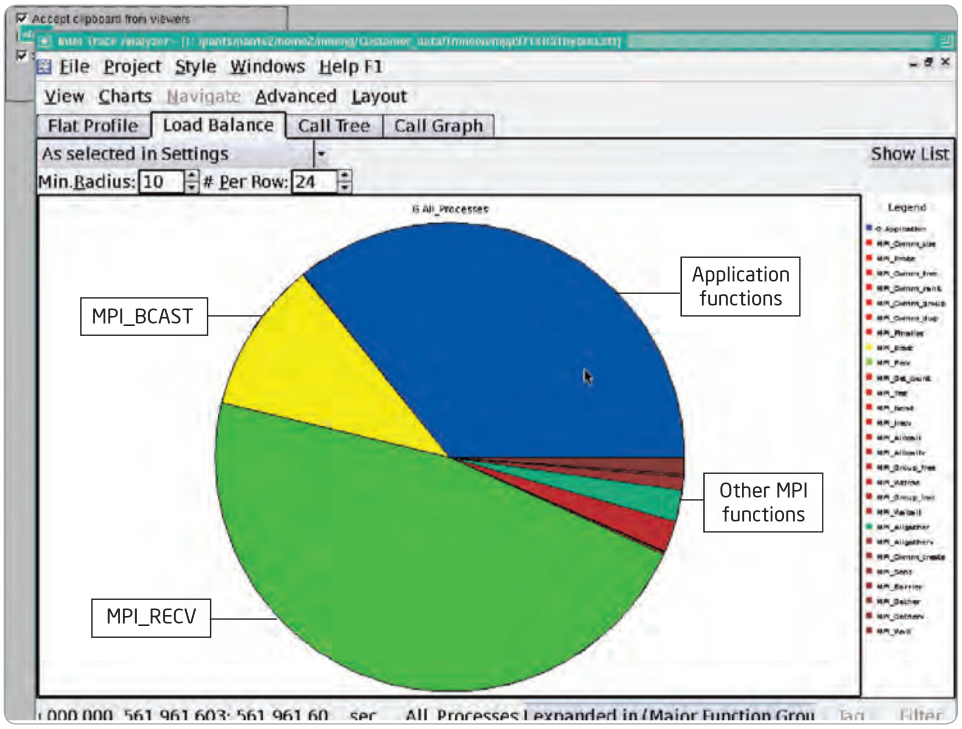


Figure 12. Load balance of the 1M model in pie mode on 128 nodes generated by Intel® Trace Analyzer and Collector.

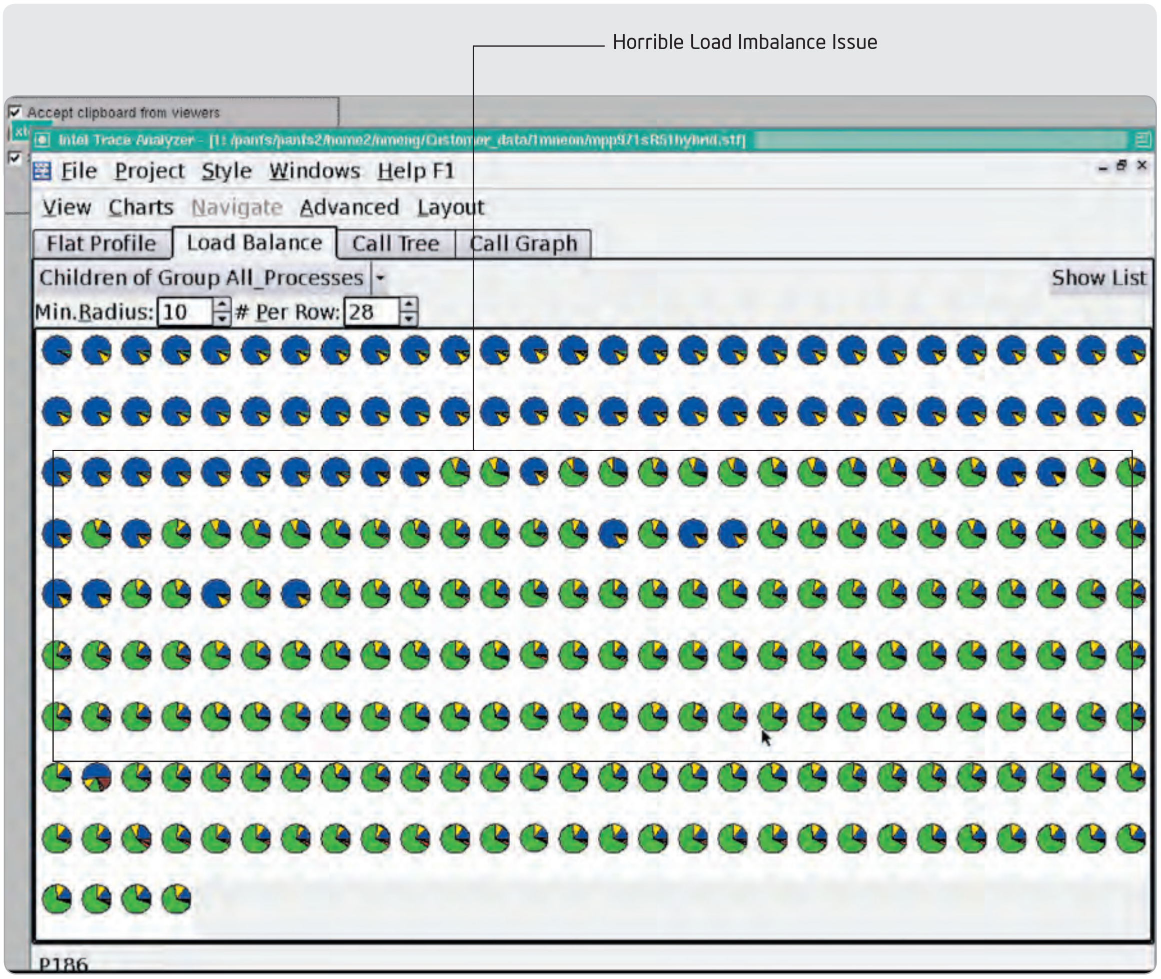

Horrible Load Imbalance Issue
Accept clipboard from viewers
Intel Trace Analyzer - [1: /panfs/panfs2/home2/nmeng/Customer_data/1mneon/mpp971sR51hybrid.stf]
File Project Style Windows Help F1
View Charts Navigate Advanced Layout
Flat Profile
Load Balance
Call Tree
Call Graph
Children of Group All_Processes
Show List
Min.Radius: 10
# Per Row: 28
P186

Figure 13. Load balance of the 1M model in pie mode on 128 nodes generated by Intel® Trace Analyzer and Collector.

Solving problems with ITAC during development

During code development, we selected a one million model as a verification test case. The model is tested with HYBRID LS-DYNA using Intel® MPI 4.0 library. During the verification test, we encountered a serious performance issue. We quickly reproduced the performance issue with Intel Trace Analyzer and Collector 8.0.1 library. **Figures 11**, **12**, and **13** show the root cause.

As you can see, the blue area in each pie indicates the percentage of application function computing cost, the green area in each pie indicates the percentage of MPI_RECV function computing cost, and the yellow area in each pie indicates the percentage of MPI_BCAST function computing cost. We found root cause (serious load imbalance issue) quickly and easily with the Intel Trace Analyzer and Collector. The serious load imbalance issue is from MPI_RECV function. The issue was fixed quickly. Finally, the target performance was achieved.

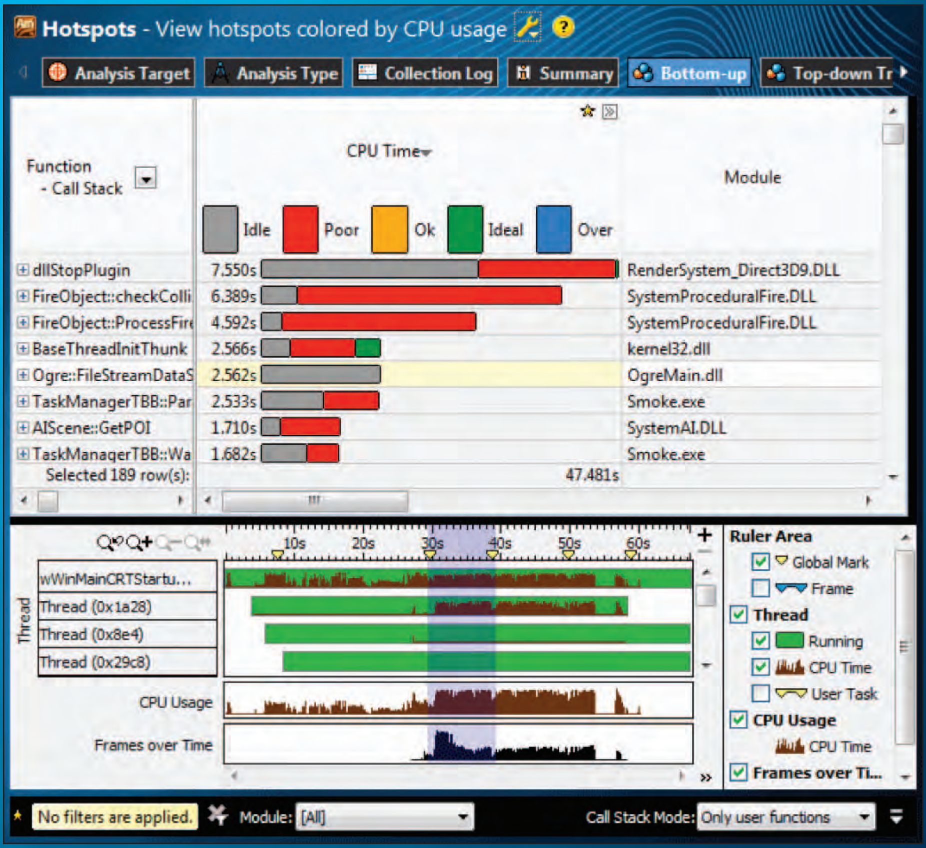
Conclusion

We believe and have shown that the Intel cluster tools can significantly reduce the time and effort to achieve performance increases from years to months. Further, the tools can help when performance is not what is expected.

Finally, we have worked to reduce the learning curve so that experienced parallelism enablers and knowledgeable application developers can quickly gain the additional insights that the cluster tools provide. ☐

New from the makers of
Intel® VTune™ Performance Analyzer
and Intel® Visual Fortran Compiler

INTEL® PARALLEL STUDIO XE



Hotspots - View hotspots colored by CPU usage

Analysis Target | Analysis Type | Collection Log | Summary | Bottom-up | Top-down Tr

Function - Call Stack	CPU Time	Module
dllStopPlugin	7.550s	RenderSystem_Direct3D9.DLL
FireObject::checkColli	6.389s	SystemProceduralFire.DLL
FireObject::ProcessFir	4.592s	SystemProceduralFire.DLL
BaseThreadInitThunk	2.566s	kernel32.dll
Ogre::FileStreamDataS	2.562s	OgreMain.dll
TaskManagerTBB::Par	2.533s	Smoke.exe
AIScene::GetPOI	1.710s	SystemAI.DLL
TaskManagerTBB::Wa	1.682s	Smoke.exe
Selected 189 row(s):	47.481s	

No filters are applied. Module: [All] Call Stack Mode: Only user functions

Achieve enhanced developer productivity

Intel® Parallel Studio XE 2011 combines ease-of-use innovations with advanced functionality for high performance, scalability, and code robustness on both Linux and Windows.

The ultimate all-in-one performance toolkit

The integrated suite helps high-performance computing and enterprise developers boost performance, reliability, and security. Squeeze the most out of applications that never have enough performance, including simulation, video rendering, seismic analysis, and medical imaging.

Advanced compilers and libraries

Intel® Composer XE

Advanced memory, threading, and security analyzer

Intel® Inspector XE

Advanced performance profiler

Intel® VTune™ Amplifier XE

Rock your code. Rock your world.

Get a free 30-day trial of Intel Parallel Studio XE today at http://software.intel.com/en-us/articles/intel-parallel-studio-xe/.

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

TAKE PERFORMANCE TO THE EXTREME.

INTRODUCING INTEL® PARALLEL STUDIO XE

From one-person start-ups to enterprises with thousands of developers working on a single application, Intel® Parallel Studio XE 2011 extends industry-leading development tools for unprecedented application performance and reliability.

Advanced compilers and libraries	Advanced memory, threading, and security analyzer	Advanced performance profiler
Intel® Composer XE	Intel® Inspector XE	Intel® VTune™ Amplifier XE

Rock your code. Rock your world.

Get a free 30-day trial of Intel Parallel Studio XE today at http://software.intel.com/en-us/articles/intel-parallel-studio-xe/.

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