Introducing Intel® Parallel Studio 2011
by Leila Chucri

Intel® Cilk Plus for Fine-Grained Parallelism
by Krishna Ramkumar

The World’s First Sudoku ‘Thirty-Niner’
by Stephen Blair-Chappell
Tired of bottlenecks?

Introducing Intel® Parallel Amplifier 2011

Quickly find multicore performance bottlenecks without needing to know the processor architecture or assembly code.

A powerful threading and performance profiler, Intel Parallel Amplifier helps Microsoft Visual Studio® C/C++ developers fine-tune applications to ensure cores are fully exploited and new capabilities supported.

Find the functions in your application that consume the most time.

The source view shows you the exact location on your source.

Rock your code.

ROCK YOUR WORLD.

Intel Parallel Amplifier 2011 is part of Intel® Parallel Studio 2011, the ultimate all-in-one performance toolkit for serial and parallel C/C++ applications.


© 2010, Intel Corporation. All rights reserved. Intel, the Intel logo, Intel Core, and Intel VTune are trademarks of Intel Corporation in the U.S. and other countries. *Other names and brands may be claimed as the property of others.

CONTENTS

Letter from the Editor

Breaking Ground and Building Trust, by James Reinders 4

James Reinders, lead evangelist and director of Intel® Software Development Products, addresses recent innovations in apps and tools, highlights key 2010 milestones, and explores what's next in the new year and beyond.

Intel® Parallel Studio 2011: Getting Your App Ready for Multicore Just Got Easier, by Leila Chucri 6

Engineer Leila Chucri introduces Intel® Parallel Studio 2011, examining new features with an eye to enabling the development environment for multicore.

Using Serial Modeling Tools to Tame the Parallel Beast, by John Pieper 12

Addressing many of the key reasons why parallelism is considered difficult, Intel® Parallel Advisor offers a proven threading methodology, and enables parallel and serial modeling.

Intel® Cilk Plus, by Krishna Ramkumar 16

Intel® Cilk Plus adds fine-grained task parallelism support to C and C++, making it easy to add parallelism to both new and existing software, and efficiently exploit multiple processors.

Nine Tips to Parallel Programming Heaven

by Stephen Blair-Chappell 19

In this interview, Dr. Yann Golanski shares with us his favorite tips on parallel programming. The tips are based on investigative work on parallel n-body simulation code carried out during his doctoral studies.

The World’s First Sudoku* “Thirty Niner”

by Stephen Blair-Chappell 20

Lars Petersen and Håvard Graff, two talented engineers from Oslo, share with us how they created what may be the world’s first Sudoku* puzzle that has 39 clues.

© 2010, Intel Corporation. All rights reserved. Intel, the Intel logo, Intel Core, and Intel VTune are trademarks of Intel Corporation in the U.S. and other countries. *Other names and brands may be claimed as the property of others.
Multicore processors have brought parallel computing into the mainstream. Software developers are embracing parallelism tools and programming models. In September 2010, after a successful summer of beta testing with customers around the world, we are introducing Intel® Parallel Studio 2011 (see the article on Intel Parallel Studio 2011 in this issue), which includes a range of new features, including:

- A comprehensive set of parallelism development models – Intel® Parallel Building Blocks (Intel® PBB)
- An innovative threading assistant – Intel® Parallel Advisor 2011
- Full support for Microsoft Visual Studio 2010*, as well as 2005 and 2008
- Premier technical support

Both Intel Parallel Studio 2011 and Intel PBB build on prior products, provide even more support for parallel programming, and are backward compatible.

When we decide to rely on a product, we are looking for signs that it is worthy of our trust. The key for me is figuring out the commitment level from Intel and very committed customers. When we decide to rely on a product, we are looking for signs that it is worthy of our trust. The key for me is figuring out the commitment level from Intel and very committed customers.

Both Intel Parallel Studio 2011 and Intel PBB build on prior products, provide even more support for parallel programming, and are backward compatible.

When we decide to rely on a product, we are looking for signs that it is worthy of our trust. The key for me is figuring out the commitment level from Intel and very committed customers. When we decide to rely on a product, we are looking for signs that it is worthy of our trust. The key for me is figuring out the commitment level from Intel and very committed customers.

With Intel® Cilk Plus, we have focused a compiler implementation. Unlike Intel TBB, Intel® Cilk Plus involves the compiler in optimizing and managing parallelism. Benefits include: (1) code that is easier to write and comprehend because it is better integrated into the language through use of keywords and native syntax, and (2) compiler optimizations that allow for more avoidance of data race conditions, and slightly better performance. The compiler understands these four parts of Intel® Cilk Plus, and it is therefore able to help with compile-time diagnostics, optimizations, and runtime error checking. Intel® Cilk Plus has an open specification, so other compilers may also implement these exciting new C/C++ language features. Intel® ArBB has not stood still; the latest version includes support for FIFO scheduling, additional support for C++ 0x, a new container (concurrent_unordered_map), a “Design Patterns” manual, and Microsoft Visual Studio 2010 support. Also available is a community preview feature called tbb::graph, which is a unified approach that lets applications use graphs to express parallelism. This is a valuable feature that can be used to implement anything from simple static dependency graphs to complex message-passing actor-like graphs. It is an evolving new feature that adds substantial coordination capabilities.

Intel® Cilk Plus is made up of four things:

- Three simple but very powerful keywords to express parallelism: cilk_for, for parallel loops, and cilk_spawn + cilk_sync for parallel function invocation
- A solution, called hyperobjects, to deal with contention for shared variables by replicating them as local copies to match tasks created by the new keywords, and for providing their reduction back to a shared value at the conclusion of a parallel computation
- Array notations for data parallel operations on array slices, such as a[0] = b[0] + c[0] to sum two vectors, or a[0] + a[1] + a[2] = sum(a[0:2])
- Array notations used with elemental functions to allow computations such as __seed_map(omega, Z, x, y, z)

With Intel® Cilk Plus, we have focused a compiler implementation. Unlike Intel TBB, Intel® Cilk Plus involves the compiler in optimizing and managing parallelism. Benefits include: (1) code that is easier to write and comprehend because it is better integrated into the language through use of keywords and native syntax, and (2) compiler optimizations that allow for more avoidance of data race conditions, and slightly better performance. The compiler understands these four parts of Intel® Cilk Plus, and it is therefore able to help with compile-time diagnostics, optimizations, and runtime error checking. Intel® Cilk Plus provides a compiler implementation.

Intel® ArBB has not stood still; the latest version includes support for FIFO scheduling, additional support for C++ 0x, a new container (concurrent_unordered_map), a “Design Patterns” manual, and Microsoft Visual Studio 2010 support. Also available is a community preview feature called tbb::graph, which is a unified approach that lets applications use graphs to express parallelism. This is a valuable feature that can be used to implement anything from simple static dependency graphs to complex message-passing actor-like graphs. It is an evolving new feature that adds substantial coordination capabilities.
Getting Your App Ready for Multicore Just Got Easier

Intel® Parallel Studio 2011

By Leila Chucri
Product Marketing,
Intel Corporation

Leila Chucri introduces Intel® Parallel Studio 2011, examining new features with an eye to enabling the development environment for multicore.

Multicore: The Game Changer

Once relegated to a small niche of performance-hungry apps like gaming, parallel programming is making speedo inroads with mainstream developers. Designing applications that exploit the performance and scalability attributes of multicore technology is no longer considered “nice to have” functionality. Today, every developer knows that as the number of processor cores increases, serial applications can no longer rely on the latest processor chip to automatically provide greater headroom or a better user experience. Parallel programming enables applications to fully utilize multicore innovations and stay competitive. Whether optimizing serial code or creating new applications for multicore, parallel programming has evolved to improve proficiency with flexible design models and tools that make applications reliable and fast.

Making Parallel Application Design – Simple, Efficient, and Fast

Over a year ago, Intel launched the first product to ease mainstream developers adopting multicore: Intel® Parallel Studio. Intel Parallel Studio was the first development tool to enable auto-scaling features through a comprehensive set of compilers, threaded libraries, memory error detection, and analysis tools. Although other companies have jumped on this new development environment, including Microsoft with Visual Studio PPL*, Intel maintains its strong leadership with the introduction of Intel® Parallel Studio 2011. The latest edition includes the first tool to guide a developer on where their application can benefit most from multicore, plus a new set of task and data parallel models. This quick overview introduces some of the noteworthy new features of Intel Parallel Studio 2011.

Intel Parallel Studio 2011 Revealed

The advanced tools and methodologies of Intel Parallel Studio 2011 help developers thrive in the multicore world. It enables Microsoft Visual Studio 2005*, 2008*, 2010* C/C++ developers to easily ramp up to parallelism and achieve the application performance and scalability benefits of multicore processor technology today, and in the future.

What’s New

- Expanded Threading Options with Intel® Parallel Building Blocks (Intel® PBB): Intel® Parallel Building Blocks is a comprehensive and complementary set of parallel models that provide solutions ranging from general purpose to specialized parallelism. It enables developers to mix and match parallel models within an application to suit its specific environment and needs, providing a simple yet scalable way to develop for multicore.
- Intel® Threading Building Blocks 3.0: The widely adopted, award-winning C++ template library solution provides flexible, cross-platform parallelism solution enabling general parallelism.
- Intel® Cilk Plus: Built into the Intel® C/C++ Compiler, it provides an easy-to-use, well-structured model that simplifies parallel development, verification and analysis.
- Intel® Array Building Blocks: An API backed by a sophisticated runtime library, it provides a generalized data parallel programming solution that frees developers from dependencies on particular low-level parallelism mechanisms or hardware architectures (available in beta at: software.intel.com/en-us/data-parallel/)
- Intel® Parallel Advisor: An innovative threading guide simplifies and demystifies parallel application design.
- Microsoft Visual Studio 2010 support

“Intel® TBB was surprisingly quick and simple to implement, and made the Simul weather SDK really fly on the Intel® Core™ i7 processor. With close to linear scaling, Simul Weather* and Intel’s tools open up great opportunities for game developers to integrate dynamic weather and clouds.”

Roderick Kennedy,
Principal and founder of Simul Software

**Intel® Parallel Advisor:** An innovative threading guide simplifies and demystifies parallel application design.

Take the easy path to parallelism with Intel Parallel Advisor to design your parallel application and tap into the performance and scalability advantages of multicore.

Whether you are new to parallelism or currently developing parallel applications, you will find Intel® Parallel Advisor an essential tool to use during the design and discovery phase of parallelizing code.

Intel Parallel Advisor offers visibility to developers on where to consider parallelism in C/C++ code; helps developers identify and experiment with parallel opportunities; exposes performance benefits; identifies how to restructure the application; as well as what time saving tools to use to debug and tune the application; and enables developers to evaluate the return on their investment before committing to the process of parallelizing their application.

Intel Parallel Advisor gives developers a clear roadmap for parallelizing applications, with step-by-step guidance for threading applications using widely adopted, proven abstractions for parallelism; provides the steps for task-oriented parallelism; and targets coarse-grain parallelism.

**Intel® Parallel Composer 2011**

Intel® Parallel Composer 2011—C/C++ optimized compiler with Intel® Integrated Performance Primitives’ advanced libraries and Intel® Parallel Building Blocks (Intel® PBB), a set of comprehensive parallel development models. Together, these tools help enhance performance and streamline parallel application development.

Intel Parallel Composer helps simplify adding parallelism with the newest versions of Intel® Compilers and libraries for parallelism and thousands of lines of ready-to-use code. Supports the entire spectrum of parallel expression from simple to complex, data to task, by offering application-class-specific, pre-threaded, and thread-safe libraries. Saves time and takes advantage of multicore processors with automatic libraries.

**“Here at trading systems Lab, we got a 10% to 20% performance boost in the multimode trading simulator that’s used in our TsL Algo Auto-Design Platform by using the C++ compiler in Intel® Parallel Studio. The compatibility with Microsoft Visual C++ is great, and we’re looking forward to using more parallelism features in Parallel Studio.”**

Mike Barna, President trading systems Lab

---

**Intel® Parallel Advisor:**

*“The Intel® Parallel Advisor design approach was instrumental in introducing parallelism into our code. The survey feature helped improve our code by finding areas in our serial code that took a lot of CPU time, and where our code would benefit from parallelism.”*

**Dr. William Ortung**

Emeritus Professor of Chemistry

*“Intel® Parallel Advisor reduces the risk of adding parallelism, since it helps focus effort in the right place, avoiding wasted implementation effort.”*

**Matt Osterberg**

Vickery Research Alliance

---

**Intel® Parallel Composer 2011:**

*“As a Microsoft Visual Studio* C/C++ developer for many years, and with no previous experience with parallel programming, Intel® Parallel Advisor proved to be a major boon in making it easier and efficient to implement parallelism through the planning and production phase.”*

**Brian Reynolds**

Brian Reynolds Research

---

**Design Phase**

Intel® Parallel Advisor analyzes the executing serial program as developers progress through the methodology.

1. Survey Target—Helps focus on hot call trees and loops as locations to experiment with parallelism.
2. Annote Sources—Enables developers to insert Parallel Advisor annotations into their sources to describe parallel experiments.
3. Check Suitability—Helps evaluate the performance of their parallel experiment by displaying the performance projection for each parallel site and how each site’s performance impacts the entire program.
4. Check Correctness—Assists developers by identifying data issues such as races in the parallel experiment.

---

**Build and Debug Phase**

Optimizing Compiler with Libraries Boosts Application Performance

---

Verify Phase
Dynamic Analysis Memory and Threading Checker

Catching software defects early in the development cycle can save you time and cost, and increase your ROI. Intel® Parallel Inspector provides a comprehensive solution for serial and multithreading error checking. Pinpoints memory leaks and memory corruption as well as thread data racers and deadlocks.

```
<table>
<thead>
<tr>
<th>Problem Sets</th>
<th>Sources</th>
<th>Modules</th>
<th>Object Size</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>F1</td>
<td>main.cpp</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>F2</td>
<td>main.cpp</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>F3</td>
<td>main.cpp</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>F4</td>
<td>main.cpp</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>F5</td>
<td>main.cpp</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>F6</td>
<td>main.cpp</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>F7</td>
<td>main.cpp</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>F8</td>
<td>main.cpp</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>F9</td>
<td>main.cpp</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>F10</td>
<td>main.cpp</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

"The performance benefits of multicore and manycore are critical to SIMULIA's business. Intel® Parallel Inspector provides a powerful way to develop parallel code compared to traditional methods, which can be lengthy and costly—especially if the price of unstable code is paid by the customer."

Matt Dunbar,
Chief Architect SIMULIA

Tune Phase
Threading and Performance Profiler

By removing the guesswork and analyzing performance behavior in Windows® applications, Intel® Parallel Amplifier 2011 provides developers quick access to scaling information for faster and improved decision making. Fine-tune for optimal performance, ensuring cores are fully exploited and new capabilities supported.

"Thanks Intel, you guys rock! I decided to give Intel® Parallel Amplifier a run. I was delighted when it pointed me to the right source line that was taking much of the time. I made the change, and voilà, our app is now almost 10 times faster. The GUI is very easy to use, in my opinion."

Dat Chu,
Research Assistant Computational Biomedicine Lab
University of Houston

Addressing many of the key reasons why parallelism is considered difficult, Intel® Parallel Advisor offers a proven threading methodology, and enables parallel and serial modeling.
A lot of programmers seem to think parallelism is hard.

A quick Internet search will yield several blogs talking about the difficulty of writing parallel programs (or parallelizing existing serial code). There do seem to be many stumbling blocks waiting to trip up the novice. Here's a representative list (based on one by Anwar Ghuloum):

- **Finding the parallelism:** This can be difficult because in writing code for serial performance, we often use memory in ways that limit the available parallelism. Simple fixes for serial performance often complicate the original algorithm and hide the parallelism that is present.

- **Avoiding the bugs:** (data races, deadlocks, and other synchronization problems.) Certainly, there is a class of bugs that affect parallel programs that serial programs don't have. And in some cases, they are worse, because timing-sensitive bugs are often hard to reproduce—especially in a debugging session.

- **Tuning performance:** (granularity, throughput, cache size, memory bandwidth.) Serial programmers have to worry about memory locality, but for parallel programs, the programmer also has to consider the parallel overheads, and unique problems, like false-sharing of cache lines.

- **Future proofing:** Serial programmers don't worry about whether the code they are writing will run well on next year's processors—it's the job of the processor companies to maintain upward compatibility. But parallel programmers need to think about how that code will run on a wide range of machines, whether there are one or two or more processors. Software that is tuned for today's quad-core processors may still be running unchanged on future 16-, 32-, or even 64-core machines.

- **Using modern programming methods:** Object-oriented programming makes it much less obvious where the program is spending its time.

Other reasons that parallel programming is considered hard include the complexity of the effort, not enough help for developers unfamiliar with the techniques, and a lack of toolset for dealing with parallel code. When adding parallelism to existing code, it can also be difficult to make all the changes needed to add parallelism all at once, and to ensure that there is enough testing to eliminate timing-sensitive bugs.

**Intel® Parallel Advisor** is a new tool for developers who need to add parallelism to existing serial code, and it helps to address these problems. Intel Parallel Advisor makes it easier to parallelize. It increases the programmer's return on investment by focusing effort where it matters. It helps the programmer identify problems early, so that little effort is wasted on unproductive directions. The key to Intel Parallel Advisor's success is its reliance on a well-proven method of introducing parallelism: serial modeling.

“By modeling parallel overheads for a range of processors, Intel® Parallel Advisor encourages the programmer to resolve scaling issues at the design stage.”

Evolving serial code to parallel with Intel Parallel Advisor

Intel Parallel Advisor leads the programmer with a proven threading method. This methodology was developed by Intel engineers over years of working with customers to thread their applications. The key to the method is to consistently check and resolve problems early, while slowly evolving the code from pure serial, to serial but capable of being run in parallel, to parallel.

The first step is to measure where the application spends time—effort spent in hot areas will be effective, while effort spent elsewhere is wasted. The next step is to insert annotations to describe a potential parallelization. The annotations form an experiment: what would happen if this code ran in parallel? The annotations do not create actual parallelism, but describe where in the serial program parallelism could be inserted. Intel Parallel Advisor observes the execution of the program with annotations, and uses the serial behavior to predict the performance and bugs that might occur if the program actually executed in parallel.

Intel Parallel Advisor models the performance of the theoretical parallel model. This allows the programmer to check the effect of overheads of parallelization. Checking early in the process, while the program is still serial, ensures that time is not wasted on parallelizations that are doomed to poor performance. Either the annotations are changed to model a different parallelization that resolves the performance issue, or, if no alternative is practical, the programmer can focus on a more profitable location.

The first step is to measure where the application spends time—effort spent in hot areas will be effective, while effort spent elsewhere is wasted. The next step is to insert annotations to describe a potential parallelization. The annotations form an experiment: what would happen if this code ran in parallel? The annotations do not create actual parallelism, but describe where in the serial program parallelism could be inserted. Intel Parallel Advisor observes the execution of the program with annotations, and uses the serial behavior to predict the performance and bugs that might occur if the program actually executed in parallel.

Intel Parallel Advisor models the performance of the theoretical parallel model. This allows the programmer to check the effect of overheads of parallelization. Checking early in the process, while the program is still serial, ensures that time is not wasted on parallelizations that are doomed to poor performance. Either the annotations are changed to model a different parallelization that resolves the performance issue, or, if no alternative is practical, the programmer can focus on a more profitable location.

Intel Parallel Advisor also models the correctness of the theoretical parallel program described by the annotations. This lets the tool detect race conditions and other synchronization errors, while still running the serial program. Because the program still runs serially, it is easy to debug and test, and computes the same results. The programmer can change the program—either the annotations or the code itself—to resolve the potential races. After each change, the program remains a serial program (with annotations) and can be tested and debugged using the normal processes.

When the program has fully evolved, the result is a correct serial program with annotations describing a parallelization with known good performance and no synchronization issues.

The value of the Intel Parallel Advisor method

Intel Parallel Advisor addresses many of the key reasons why parallelism is considered difficult. Evolving a program step-by-step is much easier than attempting to convert all at once. Taking small steps and testing after each change allows a more comfortable development model. It also reduces risk—at any point, the parallelization effort can be suspended and resumed later. The partially evolved code is still a correct serial program. In fact, it’s often a simpler, easier-to-understand program, since the changes to enable parallelism tend to regularize and simplify the code.

Focusing on the hot parts of the program ensures that time is spent effectively. It also encourages the programmer to consider coarsely-grained task parallelism, which is more likely to result in good scaling. Intel Parallel Advisor’s performance model builds on this approach. By modeling parallel overheads for a range of processors, Intel Parallel Advisor encourages the programmer to resolve scaling issues at the design stage.

Serial modeling gives the programmer the best of both worlds. The code remains serial, computes the same answer, and uses the same test system, even as it evolves to parallelism. The serial modeling tools allow the programmer to focus on critical problems that might occur in the parallel program without the difficulty inherent in non-deterministic execution.

When the method “fails”

Another advantage of the Intel Parallel Advisor method becomes evident when we consider what happens if the parallelization effort fails. If the programmer is unable to find a correct, high-performing parallelization, the annotations can be deleted or commented out. Further, since the effort of adding annotations and modeling performance and correctness is much lower than the effort of implementing parallelism, the end result—the original serial program—is reached much more quickly with Intel Parallel Advisor, so the amount of “wasted” effort is minimized.

**Design it right the first time with Intel® Parallel Advisor.**

Intel Parallel Advisor is available as part of Intel® Parallel Studio. Download an evaluation kit, and see how easy parallelism can be.
By Krishna Ramkumar  
Technical Consulting Engineer, Intel Corporation

Intel® Cilk Plus adds fine-grained task parallelism support to C and C++, making it easy to add parallelism to both new and existing software, and efficiently exploit multiple processors.

**Introduction**

It has been five years since we had free lunches. I am referring to the legendary article by Microsoft's Herb Sutter on the multicore revolution and the imperative to write software that takes advantage of multiple execution units on the same chip or die. As Sutter points out, this entails a fundamental turn or a paradigm shift in programming for the multicore era. From a software development perspective, it clearly means that developers have to revisit the implementation of their software and “parallelize” their erstwhile serial application. They have to decompose their application into tasks, and identify tasks that are independent of each other so that such tasks can be run in “parallel” (you may read it as “at the same time”). Also, with hardware support for vectorization constantly improving, it is definitely wise to harness data parallelism in your program. In order to take advantage of multicore platforms with advanced vectorization capabilities, easy and efficient programming models are required to realize both task and data parallelism in your program. In order to take advantage of multicore computer systems. It is the result of nearly two decades of research by some of the industry’s smartest minds. Intel Cilk Plus is a technology that adds task and data parallelism support to C and C++, making it easy to add parallelism to both new and existing software to efficiently tap the potential of multiple processors.

Intel Cilk Plus is particularly well suited for, but not limited to “divide and conquer” algorithms. This strategy solves problems by breaking them into sub-problems (tasks) that can be solved independently and then combining the results. Recursive functions are often used for divide-and-conquer algorithms, and are well supported by Intel Cilk Plus. Tasks can either be functions or iterations of a loop. Intel Cilk Plus keywords identify function calls and loops that can run in parallel. The Intel Cilk Plus runtime schedules these tasks to run efficiently on the available processors.

From a C/C++ developer perspective, it is a simple yet powerful model because it adds very little overhead in transforming an existing serial program to a parallel implementation. This simplicity may seem limiting at first when compared to OpenMP or Intel TBB, both of which come with many parallel programming constructs. However, Intel Cilk Plus has all the features needed to support the implementation of parallel algorithms. Its sophistication is in its simplicity.

Intel Cilk Plus requires minimal to no intrusion into the source code, making a serial program just a few keywords away from parallelism, thus easing the transition to parallelism for programmers trained to solve problems serially. In fact, a program parallelized and executed by the Intel Cilk Plus runtime steals work from chunks that have busy workers and runs them on chunks that have idle workers. This makes software development a natural progression for serial programmers, while also providing the performance benefits of parallelism.

**Intel Cilk Plus**

Intel Cilk Plus is a programming methodology/platform designed to enable developers to take advantage of today’s multicore computer systems. Intel Cilk Plus is a technology that adds task and data parallelism support to C and C++, making it easy to add parallelism to both new and existing software to efficiently tap the potential of multiple processors.

Intel Cilk Plus is particularly well suited for, but not limited to “divide and conquer” algorithms. This strategy solves problems by breaking them into sub-problems (tasks) that can be solved independently and then combining the results. Recursive functions are often used for divide-and-conquer algorithms, and are well supported by Intel Cilk Plus. Tasks can either be functions or iterations of a loop. Intel Cilk Plus keywords identify function calls and loops that can run in parallel. The Intel Cilk Plus runtime schedules these tasks to run efficiently on the available processors.

From a C/C++ developer perspective, it is a simple yet powerful model because it adds very little overhead in transforming an existing serial program to a parallel implementation. This simplicity may seem limiting at first when compared to OpenMP or Intel TBB, both of which come with many parallel programming constructs. However, Intel Cilk Plus has all the features needed to support the implementation of parallel algorithms. Its sophistication is in its simplicity.

Intel Cilk Plus requires minimal to no intrusion into the source code, making a serial program just a few keywords away from parallelism, thus easing the transition to parallelism for programmers trained to solve problems serially. In fact, a program parallelized and executed by the Intel Cilk Plus runtime steals work from chunks that have busy workers and runs them on chunks that have idle workers. This makes software development a natural progression for serial programmers, while also providing the performance benefits of parallelism.

By using reducers, you only need to identify the global variables as reducers where they are declared. No logic needs to be restructured. In contrast, most other concurrency platforms have a hard time expressing race-free parallelization of this kind of code. The reason is that reductions in most languages are tied to a control construct. For example, reduction in OpenMP is tied to the parallel for loop, and Intel Cilk Plus is suitable for both procedural as well as an object-oriented codebase. Intel TBB can also be used to parallelize C++ code where portability to non-Intel compilers is required. Intel Cilk Plus and Intel TBB are different interfaces for similar underlying threading runtime implementation. Both implement work/task stealing where the runtime steals work from chunks that have busy workers and runs them on chunks that have idle workers. This is what makes software development using Intel Cilk Plus and Intel TBB scalable.

**Relationship/Co-existence with Intel TBB**

Intel TBB is a widely used C++ template library for parallelism that extends C++ by abstracting away thread management, thereby allowing straightforward parallel programming. A full-blown discussion of Intel TBB is beyond the scope of this article. The interested reader could pick up a copy of the book titled Intel Threading Building Blocks written by Intel’s James Reinders and published by O’Reilly, which has a comprehensive treatment of the concepts and features of Intel TBB.

The underlying symmetry in the runtime shared by Intel Cilk Plus and Intel TBB is extremely useful for developers. While Intel Cilk Plus is suitable for both procedural as well as an object-oriented codebase, Intel TBB can also be used to parallelize C++ code where portability to non-Intel compilers is required. Intel Cilk Plus and Intel TBB are different interfaces for similar underlying threading runtime implementation. Both implement work/task stealing where the runtime steals work from chunks that have busy workers and runs them on chunks that have idle workers. This makes software development using Intel Cilk Plus and Intel TBB scalable.

**Conclusion**

When it comes to programming in parallel, the importance of synchronization primitives, thread-safe container data structures, and scalable memory allocators cannot be overemphasized. Thanks to Intel TBB being implemented as a C++ standards-compliant parallel library, a program written using Intel Cilk Plus can use all of the above features, which are implemented by Intel TBB in an elegant fashion. We already discussed above that Intel Cilk Plus can use Intel TBB’s synchronization primitives, if needed. Intel Cilk Plus can also use Intel TBB’s thread-safe containers and scalable memory allocators.
THE PARALLEL UNIVERSE

Exploiting data-parallel hardware

More often than not, in software programs that massage a lot of data there is scope for processing the data in parallel when the same operation is to be performed on all of the data. Fortunately, modern computer systems come with additional instruction sets and associated registers that are capable of processing one operation on multiple pairs of operands at the same time. This is called vectorization. The Intel® Compiler automatically analyzes a program and generates vector code that exploits vector hardware capabilities. Besides automatic vectorization, it also has features that let you explicitly enforce vectorization in the form of compiler directives and C/C++ language extensions. These features can be used along with any programming platform such as Intel Cilk Plus, Intel TBB, or OpenMP, which gives developers a lot more choice and flexibility.

The #pragma simd compiler directive can be used for loop vectorization. The simd pragma is used to guide the compiler to vectorize a loop. Vectorization using the simd pragma complements (but does not replace) the fully automatic approach.

The C/C++ extensions (part of Intel Cilk Plus) provide explicit data parallel array notations so that the compiler can perform transformations to generate vectorized code. As an example, the following array notation can be used if element-wise multiplication of arrays b and c is to be performed.

\[
\text{a}[:,] = \text{b}[:,] \cdot \text{c}[:,]
\]

To summarize, Intel Cilk Plus is a language extension with support for task parallelism so as to make full use of multiple cores and data parallelism that can use vectorization capabilities within each core.

Combined, these extensions can harness the power of modern microprocessors.

Intel Cilk Plus is part of Intel® Parallel Composer. We encourage you to try Intel Cilk Plus by downloading Intel® Parallel Composer from the following link: http://software.intel.com/en-us/articles/intel-software-evaluation-center/. The download also includes documentation that provides detailed reference information about Intel Cilk Plus.

Examples illustrating use of Intel Cilk Plus keywords

Listing 1: cilk_spawn and cilk_sync

```c
void mergesort(int a[], int left, int right) {
    if(left < right) {
        int mid = (left + right)/2 ;
        cilk_spawn mergesort(a, left, mid);
        mergesort(a, mid+1, right);
    }
}
```

Listing 2: cilk_for

```c
/* The below calls to the mergesort function are independent of each other. In other words, they are task parallel. This parallelism can easily be expressed by using the keyword cilk_spawn as shown below. */

// Use of cilk_sync ensures that the threads spawned in this function join at this point. Serial execution resumes after this point */

cilk_sync;
merge[a, left, mid, right];
```

By Stephen Blair-Chappell

In this interview, Dr. Yann Golanski shares with us his favorite tips on parallel programming. The tips are based on investigative work on parallel n-body simulation code carried out during his doctoral studies.

1. **Just buy a faster machine.**
   - First look at how much it will cost you to make your program parallel. If it will take, say two months of coding, can you just buy a faster machine that will give you the speedup you want? Of course, once you reach the limits of a machine’s speed, you are going to have to do some parallelization.

2. **Start small.**
   - Don’t try to make everything parallel at once, just work on small bits of code.

3. **Starting from scratch?**
   - Use someone else’s wheel. If you are starting from scratch, see what other people have done. Learn from others. Don’t reinvent the wheel.

4. **Find a way of logging.**
   - Debugging your application. Make sure you have a way of tracing what your application is doing. If necessary, buy some software tools that will do the trick. Prints on their own will probably not help.

5. **Look at where the code is struggling.**
   - Examine the runtime behavior of your application. Profile the code with Intel® VTune™ Performance Analyzer. The hotspots you find should be the ones to make parallel.

6. **Write a parallel version of the algorithm.**
   - Try revisiting the algorithm so it is parallel-friendly.

7. **Stop when it’s good enough.**
   - When you think it’s good enough, stop. Step back, go for a pint. Have set goals—when you’ve achieved them, you’re done.

8. **Tread carefully. You are walking on eggs, and some eggs are land mines.**
   - Take care with the parallel code. Some innocent errors could blow up your program. Use a good tool to check for any data races and other parallel errors.

9. **Get the load-balancing right.**
   - Once you’ve made your code parallel, make sure all the threads are doing equal amounts of work.

The tips were recorded over a pleasant Thai meal in the City of York. Between the various dishes, Dr. Golanski spoke about the advice he’d give to someone starting to parallelize an application. At the end of the meal, the restaurant owner asked if we would mention his establishment in the WRoX book, and we did. He said, “If you are ever in the center of York, look for Siam House on Goodramgate.”

---

The complete case study, along with hands-on examples, will be available in the WRoX book, Parallel Programming with Intel® Cilk Plus, Stephen Blair-Chappell, Andrew Stoklosa, and Andrew Younski, published by Wiley Publishing Inc. ISBN 9780470891650 (March 2011). The complete case study, along with hands-on examples, will be available in the WRoX book, Parallel Programming with Intel® Cilk Plus, Stephen Blair-Chappell, Andrew Stoklosa, and Andrew Younski, published by Wiley Publishing Inc. ISBN 9780470891650 (March 2011).

Photo credit: NASA, Esa, and M. Livio and the Hubble 20th Anniversary Team (STScI).
By Stephen Blair-Chappell
Intel Compiler Labs
Lars Peters Endresen and Håvard Graff, two talented engineers from Oslo, share with us how they created what may be the world’s first Sudoku puzzle that has 39 clues.

Step 1: Changing the Algorithm

Rather than using a brute force method to create all the different possible solutions, we decided to take an existing puzzle, remove one or two clues, and then use a recursive solver to produce a new puzzle.

To find a new 17-clue puzzle, we start with an 18-clue puzzle, remove two clues, and then use a recursive solver to produce a new puzzle. To find a new 17-clue puzzle, we start with an 18-clue puzzle, remove two clues, and then use a recursive solver to produce a new puzzle. We use the same technique to find the ‘thirty-niner’. Taking an existing three-eighter, we remove one clue and then add two new clues — we call this a “-1 + 2” algorithm.

Adding SSE Intrinsics

SSE intrinsics are compiler-generated, assembler-coded functions that can be called from C++ code, and provide low-level access to SIMD functionality without the need to use inline assembly. Compared to using inline assembly, intrinsics can improve code readability, assist instruction scheduling, and help reduce debugging effort. Intrinsics provide access to instructions that cannot be generated using the standard constructs of the C and C++ languages.

The Intel® Compiler supports a wide range of architectural extensions, from the early MMX instructions to the latest generation of SSE4.2 instructions.

The code in Figure 2 shows how 128-bit SSE2 registers are used in the Sudoku code. The first version of the Sudoku generator did not use SSE instructions or intrinsics. Reworking the first version of the code to use SSE2 registers took a significant amount of time. The adding of SSE intrinsics gave us a speedup of several hundred.

Using SSE intrinsics does have its drawbacks. You can end up locking your implementation to a particular generation of architecture. The long names of the SSE functions can make your C++ code almost unreadable, and there is a significant learning curve for the programmer to climb. In the case of the Sudoku generator, the performance improvement far outweighed the extra effort that was needed.

Step 3: Adding Parallelism

We used OpenMP* tasks, which are defined in the OpenMP 3.0* standard. OpenMP 3.0 is supported by the Intel® C++ Compiler beginning with version 11.1. Adding the parallelism took about two weeks of work — which felt a lot of effort at the time, but in relation to the length of the project the time was fairly short and well spent.

Figure 3 gives an example of using OpenMP tasks. One of the most difficult aspects of adding the OpenMP code was to grasp how variables were treated. Data in OpenMP can be shared or private. The fine-tuning of our code to get the right scope level for our variables took several iterations. Much of the time taken in adding the parallelism was reworking the code so that there was less need to share data between the different running tasks, and making sure that there were no dependencies between the different loops that were parallelized.

Using Intel® Cilk Rather than OpenMP

Today, Intel® Parallel Studio supports a number of different ways of parallelizing a program. The diagram in Figure 4 shows Intel® Parallel Studio’s support for parallel programming.

As stated earlier, the Sudoku thirty-niner was parallelized using OpenMP; it being much easier to use than native threads.

Adding SSE instructions does have its drawbacks. You can end up locking your implementation to a particular generation of architecture. The long names of the SSE functions can make your C++ code almost unreadable, and there is a significant learning curve for the programmer to climb. In the case of the Sudoku generator, the performance improvement far outweighed the extra effort that was needed.

Figure 2: Using SSE compiler intrinsics to improve performance

THE PARALLEL UNIVERSE
Intel Cilk Plus is the Easiest Way to Parallelize an Existing Program

When making code parallel, the programmer has to be careful that no data races are introduced. If there are any global variables then the code should be reworked so that the scope of the variables is restricted—by, for example, using local or automatic variables rather than global variables. If it is impossible to remove all global variables, then access to such variables should be protected so that only one thread at a time can modify them. In Cilk the easiest way to deal with global and shared variables is to declare them to be a reducer. When a worker accesses a reducer it is given its own private view that it can safely manipulate. Views are then later merged in the serial part of the code by a call to get_value(). The code in Figure 7 shows how the global gNumCilkPuzzlesSolved is declared to be a reducer_opadd. The call to get_value is called from the serial part of the code; combines all the values of the reducer, thus obtaining the correct value.

Copyright © 2008–2014 Intel Corporation. All rights reserved.
int gNumCilkPuzzlesSolved; // global variable
- gNumCilkPuzzlesSolved++; // somewhere in parallel code
- int Tmp = gNumCilkPuzzlesSolved; // somewhere in serial code
(a) The global variable gNumCilkPuzzlesSolved is unsafe to use in parallel code.

```c
#include <cilk/reducer_oppad.h>
cilk::reducer_opadd<int> gNumCilkPuzzlesSolved;
- gNumCilkPuzzlesSolved++; // in parallel code
- int Tmp = gNumCilkPuzzlesSolved.get_value(); // in serial code
```
(b) The global variable is declared to be a reducer, making it safe to access.

In our original OpenMP solution once the parallel code was added to the project, it was rewarding to see that on an SMT quad core—which can support eight hardware threads—all eight hardware threads were kept busy. See Figure 8.
Later experimentations using Cilk showed that the Cilk solution performed equally well to the OpenMP one, and was much easier to implement. Figure 9 shows the three new “thirty-niners” that were found using our Sudoku generator.

**Intel® Cilk Plus**

Intel® Cilk Plus is made up of these main features:
- A set of keywords, for expression of task parallelism
- Reducers, which eliminate contention for shared variables among tasks by automatically creating views of them for each task and reducing them back to a shared value after task completion
- Array notations, which provide data parallelism for sections of C/C++ notation to manipulate arrays
- Elemental functions, which enable data parallelism of whole functions or operations which can then be applied to whole or parts of arrays or scalars
- The simd pragma, which lets you express vector parallelism for utilizing hardware SIMD parallelism while writing standard compliant C/C++ code with an Intel® Compiler.

**The Results**

Figure 7. Fixing data race problems by using Intel Cilk Plus reducers

Figure 8. With the parallelism implemented in the code, each hardware thread ran at 100 percent use.

Figure 9. The three 39 minimal solutions found using the minus-one-plus-two search

**Lars Peters Endresen and Håvard Graff** wrote the original program using OpenMP. The design was later rewritten by Stephen Blair-Chappell using Intel® Cilk Plus.

The complete case study, along with hands-on examples, will be available in the Wrox book Parallel Programming with Intel Parallel Studio. Stephen Blair-Chappell and Andrew Stokes, Wrox Publishing Inc. ISBN 9780470891650 (March 2011)
WHICH COMES FIRST:
parallel languages or parallel programming patterns?

CLAY BRESHARES,
COURSEWARE ARCHITECT/INSTRUCTOR,
INTEL CORPORATION

On the shuttle to the recent UPRC (Universal Parallel Computation Research Center) Annual Summit meeting on the Microsoft campus in Redmond, WA, I was listening in on a discussion about parallel programming patterns. Being a parallel programmer, I was interested in what people (and these were some of the experts in the field) had to say about parallel programming patterns, how they are evolving, and how they will impact future parallel coders. The discussion turned to whether patterns would affect programming languages directly or remain something that would be constructed from statements of the language. I think I'm in the former camp. Here's why.

For those of us that were programming when Elvis was still alive, think back to writing with assembly language. For the most part, there were instructions for Load, Store, Add, Compare, Jump, plus some variations on these and other miscellaneous instructions. To implement a counting/indexing loop you would use something like the following:

```
for (i = 0; i < N; ++i) {
    Loop Body
}
```

This is a programming pattern. (Surprised?) With the proper conditional testing and jumping (goto) instructions within the programming language, this pattern can be implemented in any imperative language.

Since this pattern proved to be so useful and pervasive in the computations being written, programming language designers added syntax to "automate" the steps above. For example, the for-loop in C:

```
for (i = 0; i < N; ++i) {
    Loop Body
}
```

From the evidence above, I think future parallel programming languages or language extensions supporting parallelism will be influenced by the parallel programming patterns we define and use today. And nothing will remain static. Ralph Johnson (Design Patterns), during his presentation, remarked that some of the original patterns saw early use, but this use has slacked off. Two reasons he noted for this was that some of the patterns couldn't easily be implemented in Java* and modern OO languages had better ways to accomplish the same tasks—most likely these new languages found inspiration from the patterns and their usage.

For an answer to the question posed in the title, it boils down (no pun intended) to the old chicken-and-egg paradox. There were algorithms/patterns to do computations before there were computers; prior to that, these algorithms were modifications of previous algorithms influenced by the tools available. Looking forward, though, we're still in the relative stages of infancy for programming, let alone parallel programming. Clearly, the next generation of parallel programming languages or libraries or extensions bolted onto serial languages will still be alive, think back to writing with assembly language. For the most part, there were instructions for Load, Store, Add, Compare, Jump, plus some variations on these and other miscellaneous instructions. To implement a counting/indexing loop you would use something like the following:

```
for (i = 0; i < N; ++i) {
    Loop Body
}
```

Parallel programming patterns will be abstractions that can be "crudely" implemented in current languages and parallel libraries, like the pseudocode above. New languages (or language extensions) will make programming parallel patterns easier and less error prone. From the example above, OpenMP* has the syntax to do this, but it only takes a single line added to the serial code:

```
#pragma omp for
for (i = 0; i < N; ++i) {
    Loop Body
}
```

This allows the programmer to have the parallelism handled by her/his compiler. For an answer to the question posed in the title, it boils down (no pun intended) to the old chicken-and-egg paradox. There were algorithms/patterns to do computations before there were computers; prior to that, these algorithms were modifications of previous algorithms influenced by the tools available. Looking forward, though, we're still in the relative stages of infancy for programming, let alone parallel programming. Clearly, the next generation of parallel programming languages or libraries or extensions bolted onto serial languages will be influenced by the patterns we use now for specifying parallel computations.

WHERE SHOULD I ADD PARALLELISM?

```
Loop Body
```

```
Initialize counter
LOOP: test end condition, goto EXIT if done
Loop Body
Increment counter
goto LOOP
EXIT: next statement
```

```
#pragma omp for
for (i = 0; i < N; ++i) {
    Loop Body
}
```

From the evidence above, I think future parallel programming languages or language extensions supporting parallelism will be influenced by the parallel programming patterns we define and use today. And nothing will remain static. Ralph Johnson (Design Patterns), during his presentation, remarked that some of the original patterns saw early use, but this use has slacked off. Two reasons he noted for this was that some of the patterns couldn't easily be implemented in Java* and modern OO languages had better ways to accomplish the same tasks—most likely these new languages found inspiration from the patterns and their usage.

For an answer to the question posed in the title, it boils down (no pun intended) to the old chicken-and-egg paradox. There were algorithms/patterns to do computations before there were computers; prior to that, these algorithms were modifications of previous algorithms influenced by the tools available. Looking forward, though, we're still in the relative stages of infancy for programming, let alone parallel programming. Clearly, the next generation of parallel programming languages or libraries or extensions bolted onto serial languages will be influenced by the patterns we use now for specifying parallel computations.
Optimization Notice

Intel® compilers, associated libraries and associated development tools may include or utilize options that optimize for instruction sets that are available in both Intel® and non-Intel microprocessors (for example SIMD instruction sets), but do not optimize equally for non-Intel microprocessors. In addition, certain compiler options for Intel compilers, including some that are not specific to Intel micro-architecture, are reserved for Intel microprocessors. For a detailed description of Intel compiler options, including the instruction sets and specific microprocessors they implicate, please refer to the “Intel® Compiler User and Reference Guides” under “Compiler Options.” Many library routines that are part of Intel® compiler products are more highly optimized for Intel microprocessors than for other microprocessors. While the compilers and libraries in Intel® compiler products offer optimizations for both Intel and Intel-compatible microprocessors, depending on the options you select, your code and other factors, you likely will get extra performance on Intel microprocessors.

Intel® compilers, associated libraries and associated development tools may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include Intel® Streaming SIMD Extensions 2 (Intel® SSE2), Intel® Streaming SIMD Extensions 3 (Intel® SSE3), and Supplemental Streaming SIMD Extensions 3 (Intel® SSSE3) instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors.

While Intel believes our compilers and libraries are excellent choices to assist in obtaining the best performance on Intel® and non-Intel microprocessors, Intel recommends that you evaluate other compilers and libraries to determine which best meet your requirements. We hope to win your business by striving to offer the best performance of any compiler or library; please let us know if you find we do not.

Notice revision #20101101

The Ultimate All-in-One Performance Toolkit

Introducing Intel® Parallel Studio 2011

Intel® Parallel Studio 2011 simplifies and speeds the analysis, compiling, debugging, error-checking, and tuning of your serial and parallel apps. With Intel Parallel Studio, you get everything you need to optimize legacy serial code, exploit multicore, and scale for manycore.

INTEL® PARALLEL STUDIO COMPONENTS

Intel Parallel Studio supports every stage of the development lifecycle.

**Innovative threading assistant**
Intel® Parallel Advisor 2011: Follow the steps for task-oriented parallelism and target coarse-grain parallelism.

**Optimizing compiler and threaded libraries**
Intel® Parallel Composer 2011: A simple recompile with Intel Parallel Composer can yield better performance.

**Memory and threading error checker**
Intel® Parallel Inspector 2011: Quickly find memory errors in your single and multithreaded applications.

**Threading and performance profiler**
Intel® Parallel Amplifier 2011: Find the functions in your application that consume the most time.

© 2010, Intel Corporation. All rights reserved. Intel and the Intel logo are trademarks of Intel Corporation in the U.S. and other countries. *Other names and may be claimed as the property of others.

Rock your code.
ROCK YOUR WORLD.