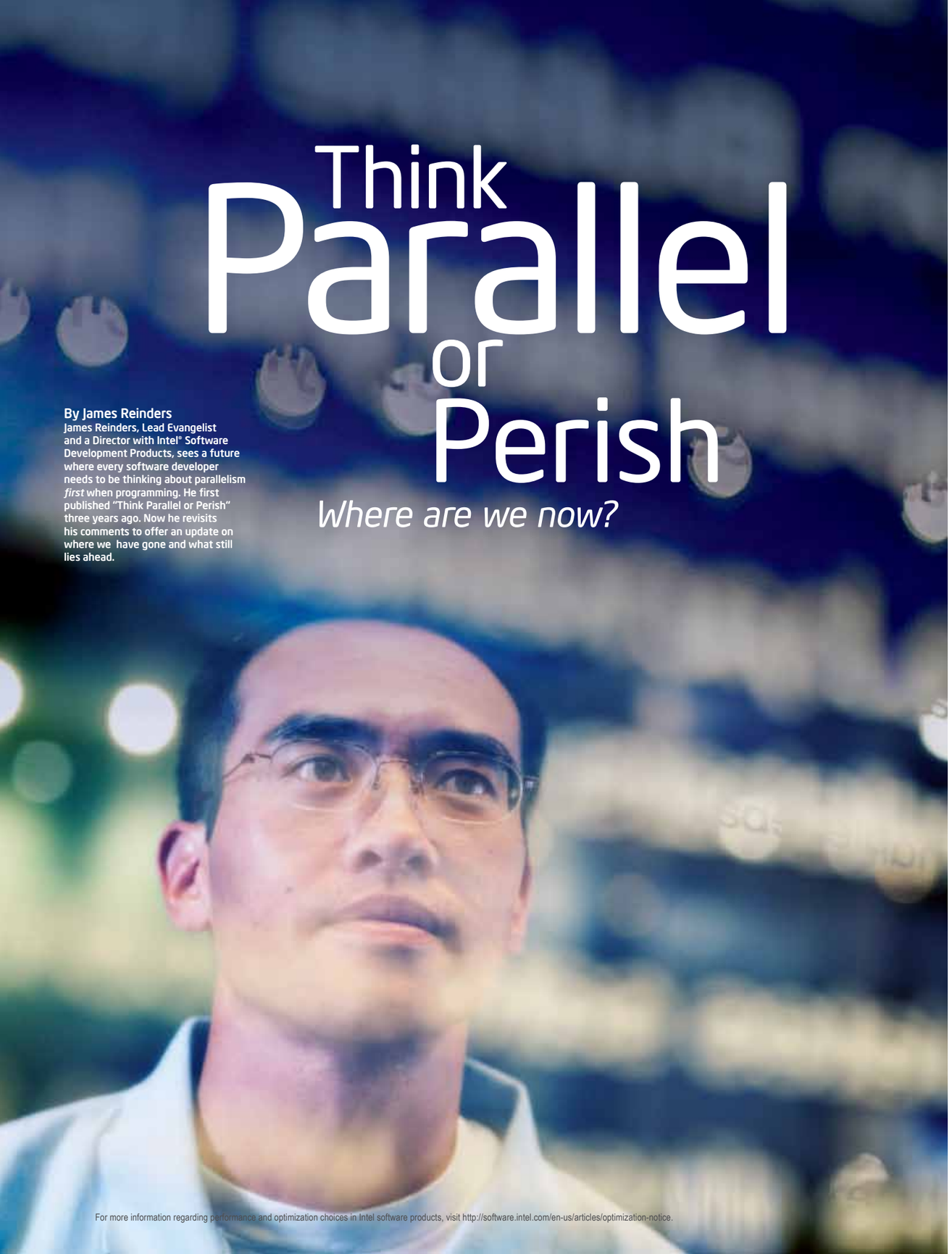
WELCOME TO THE PARALLEL UNIVERSE

Letter to the Editor
by parallelism author and expert James Reinders

Are You Ready to Enter a Parallel Universe: Optimizing Applications for Multicore
by Intel software engineer Levent Akyil

Contents



Think Parallel or Perish

Where are we now?

By James Reinders
James Reinders, Lead Evangelist and a Director with Intel® Software Development Products, sees a future where every software developer needs to be thinking about parallelism *first* when programming. He first published "Think Parallel or Perish" three years ago. Now he revisits his comments to offer an update on where we have gone and what still lies ahead.

Three years ago, as I thought about my own future in software development, I penned a piece noting that as software developers we all need to personally "Think Parallel" or face a future where others will do so and we will cease to be competitive. I am still confident that software development in 2016 will not be kind to programmers who have not learned to "Think Parallel."

Some of us will write parallel programs, some of us will not. Some will write using new languages to implement in, changing the way we do most everything. Most will not.

What we must all do is learn to "Think Parallel." Understanding how parallelism is and will be used on computers, in applications, and in our environment is critical to our future. Grasping how to use the good and avoid the bad in this new world, intuitively, is the challenge ahead of us in "Thinking Parallel."

I still have my note stuck to my cube wall: "Memo to self: The fastest way to unemployment is to not hone my programming skills for exploiting parallelism."

This year, 2009, is the fifth year in which Intel is shipping multicore processors. Those five years were not the first five years of multicore processors, but they were the five years that drove x86 processors to multicore in servers and desktop and laptop computers. In this time span, we've gone from a few multicore processor offerings to "game over" for single-core processors.

This year, 2009, we will also witness another monumental milestone: this will be the first year that Intel has a production manycore processor. Manycore processors differ from multicore processors by having a more sophisticated system of interconnecting processors, caches, memories, and I/O. This is needed at about 16 or so cores—so manycore processors can be expected to have more cores than multicore processors. The world has never seen a high-volume manycore processor, but over the next few years it will.

A good design comes from a human developer, not the tools.

This dawning of manycore processors, as profound as it is, will not affect very many programmers in 2009. It will, however, make this new parallel era even more real than multicore processors have. The opportunity for hardware to be enormously parallel is staggering.

The "not parallel" era we are now exiting will appear to be a very primitive time in the history of computers when people look back in a hundred years. The world works in parallel, and it is time for computer programs to do the same. We are leaving behind an era of too little parallelism, to which I say: "It's about time!" Doing one thing at a time is "so yesterday."

In less than a decade, a programmer who does not "Think Parallel" first will not be a programmer.

What's Intel doing to help?

When I first wrote about this a few years ago, Intel was already a longtime champion for the OpenMP* standard (www.openmp.org) and had many tools for MPI support. Our performance analysis tools and debuggers were ahead of their time with top to bottom support for parallelism. We also had emerging tools for finding deadlocks and race conditions in parallel programs.

Since then, we've helped update OpenMP to specification version 3.0 and were among the very first to support it in our compilers. We launched our own MPI library, which is more portable and higher performing than prior options.

We created Intel® Threading Building Blocks (Intel® TBB), which has become the best and most popular way to program parallelism in C++.

And, we have gone to beta with our Intel® Parallel Studio project. This takes everything we've done to date, and makes it much more intuitive, robust, and exciting. Soon, it will be a full product from Intel.

These new projects are in addition to all the tools we had a few years ago. We are growing and expanding our tools to support customers, while they expand to use parallelism. We are generally not forcing obsolescence as we expand to offer parallelism. Our customers appreciate that.

I'm also quite excited about Intel's Ct technology, and we'll be talking more about that as the year goes on. It is one of several more projects we have been experimenting with, and talking with customers about, to guide us on what more we should be doing to help.

We are building tools that help with parallel programming. We are embracing standards, extending current languages, using library APIs—all with an eye on scalability, correctness, and maintainability.

It is still up to us, as software developers, to know what to do with these wonderful tools. Just as before parallelism, a good design comes from the human developer—not the tools. Parallelism will be no different. Hence, we humans need to work on "Think Parallel."

James Reinders
Portland, Oregon
April 2009

James Reinders is Chief Software Evangelist and Director of Software Development Products, Intel Corporation. His articles and books on parallelism include *Intel Threading Building Blocks: Outfitting C++ for Multicore Processor Parallelism*. Find out more at www.go-parallelcom.

PARALLELIZATION METHODOLOGY

The Intel® Software Development Tools solve four stages of parallel application development:

1. Code and Debug with Intel® Parallel Composer: Use source code analysis capabilities of Intel Parallel Composer to identify errors in the source code during compilation phase. To enable source code analysis you will need to use the **/Qdiag-enable:sc {[1|2|3]}** compiler option. The number specifies the severity level of the diagnostics (1=all critical errors, 2=all errors, and 3=all errors and warnings). The following scenarios represent the various ways that Intel Parallel Composer supports coding of parallel applications. These scenarios are likely to manifest in different applications:

SCENARIO 1: User decides to use OpenMP 3.0* to add parallelism.

SCENARIO 2: User has code with cryptography and wants to parallelize. Add Intel® Integrated Performance Primitives (Intel® IPP) to replace cryptography code. Add OpenMP for parallelism.

SCENARIO 3: User adds Intel® Threading Building Blocks (Intel® TBB) pipeline.

SCENARIO 4: User has audio/video code and wants to parallelize. Add Intel® IPP to pertinent domains. Add Intel TBB to parallelize additional regions.

SCENARIO 5: User has code with Windows* threads. Add Intel TBB with Lambda function support to new regions to parallelize.

For the scenarios where OpenMP is used to introduce parallelism, use the **Intel® Parallel Debugger Extension for Microsoft Visual Studio*** (http://software.intel.com/en-us/articles/preview/parallel-debugger-extension) to debug the OpenMP code. The debugger has a set of features that can be accessed through the Microsoft Visual Studio Debug pull-down menu. The debugger can provide insights into thread data sharing violations, detect re-entrant function calls, provide access to SSE registers, provide access to OpenMP data structures, such as tasks, task wait lists, task spawn trees, barriers, locks, and thread teams. To debug an application with the Intel Parallel Debugger the application must be built with the Intel® C++ Compiler using the **/debug:parallel** option.

2. Verify with Intel® Parallel Inspector: Run Intel Parallel Inspector to find threading errors such as data races and deadlocks and memory errors that are likely to occur. If you found errors, fix them and rebuild the application with Intel Parallel Composer.

3. Tune with Intel® Parallel Amplifier: Intel Parallel Amplifier enables you to get the maximum performance benefit from available processor cores. Use Intel Parallel Amplifier's concurrency analysis to find processor underutilization cases where the code is not fully utilizing processor resources. Implement performance improvement changes and rebuild the application with Intel Parallel Composer. Use Intel Parallel Inspector to check for possible threading errors and then verify the effects of performance improvement changes with Intel Parallel Amplifier.

4. Check in the code.

5. Go back to step 1 and find a new region to add parallelism to.

The "Parallelizing N-Queens with the Intel® Parallel Composer" tutorial provided with Parallel Composer provides hands-on training on how to apply each of the parallelization techniques discussed in this document to implement a parallel solution to the N-Queens problem, which is a more general version of the *Eight Queens Puzzle* (http://en.wikipedia.org/wiki/N-queens). The tutorial is located in the "Samples\NQueens" folder under the Intel Parallel Composer main installation folder.

TECHNICAL SUPPORT

Visit Intel's online Community Support User Forums and Knowledge Base to get all of the help you need from our own tools and parallelism experts, and your fellow developers. Go to www.intel.com/software/support to start your search.

Welcome to the Parallel Universe

Writing Parallel Code **Safely**

Writing multithreaded code to take full advantage of multiple processors and multicore processors is difficult. The new Intel® Parallel Studio should help us bridge that gap.
By Peter Varhol

With the explosion of multicore processors, the pressure is now on application developers to make effective use of this computing power. Developers are increasingly going to have to better identify opportunities for multithreading and independent parallel operation in their applications, and to be able to implement those techniques in their code.

But these are difficult activities; understanding how to protect data and system resources during parallel operations requires technical expertise and attention to detail. The likelihood of errors, such as race conditions or overwritten data, is high.

And identifying areas in code where parallel execution is feasible is a challenge in and of itself. It typically requires a deep level of understanding of not only the code, but also of the flow of application execution as a whole. This is very difficult to achieve with current tools and techniques.

As a result, few developers want to write parallel code that takes advantage of multiple processors or processor cores, or are technically prepared to do so. This kind of coding is considered specialized and niche, even though virtually all modern servers and many workstations and laptops use multicore processors.

But multiple processor systems and multicore processors aren't going away; and in fact, the industry is going to see processors with an increasing number of cores over the next several years. It is likely that even desktop systems over the next several years will have 16 or more processor cores. Both commercial and custom software has to make better use of the computing power being afforded by these processors.

It's difficult. But tools are emerging that can make a difference in the development of parallelized applications for multicore processors and multiple processors. For example, Intel® Parallel Studio provides essential tools for working with multithreaded code, enabling developers to more easily identify errors in threads and memory, and to be able to identify and analyze bottlenecks.

Intel Parallel Studio is available for beta program download at www.intel.com/software/parallelstudio. It consists of Intel® Parallel Composer, Intel® Parallel Inspector, and Intel® Parallel Amplifier. Together, these tools enable developers building multithreaded applications for parallel execution to quickly construct and debug applications that are able to make more

efficient use of today's processors.

Intel Parallel Composer consists of a parallel debugger plug-in that simplifies the debugging of parallel code and helps to ensure thread accuracy. It also includes Intel® Threading Building Blocks and Intel® Integrated Performance Primitives, which provide a variety of threaded generic and application-specific functions enabling developers to quickly add parallelism to applications.

Intel Parallel Inspector detects threading and memory errors and provides guidance to help ensure application reliability. Intel Parallel Amplifier is a performance profiler that makes it straightforward to quickly find multicore performance bottlenecks without needing to know the processor architecture or assembly code.

CHALLENGE OF PARALLEL CONSTRUCTION

Developers increasingly find that their application code doesn't fully utilize modern multicore processors. In many cases they can achieve high performance on a single core, but without a significant effort geared toward making the code operate in parallel, cannot scale to make use of the cores available.

Parallel code is that which is able to execute independently of other code threads. On a single processor or single core system, it can offer at least the illusion of speeding up performance, because processor downtime on one thread can be used to run other threads. On multiprocessor and multicore systems, it is essential to take full advantage of multiple separate execution pipelines.

It's not easy. Most code is written in either a straightforward execution fashion or a top-down fashion. Either way, developers don't have a good opportunity to look at their code from the standpoint of operations that can be parallelized. The common development processes of today simply don't afford them the opportunity to do so.

Instead, developers have to view their code within the context of application execution. They have to be able to envision the execution of the application, and the sequence of how the source code will execute once a user is running that application. Operations that can occur independently in this context are candidates for parallel computation.

But that's only the initial step in the process. Once developers have determined what execution paths are able to run in parallel, they then have to make it happen. Threading the code isn't enough; developers have to be able to protect the threads once they launch and execute independently. This involves setting up critical sections of the code, ensuring that nothing related (or nothing else at all, depending on how critical) can execute while that section is running.

However, critical sections tend to lock resources they are working with, slowing execution and sometimes leading to deadlock. If the resources remain locked and other threads cannot execute, and other threads also hold resources that are required by the running thread, no further execution can occur. Because such a deadlock involves different resources from multiple threads, identifying the resources that cause the problem and locking out those resources at the right time is a highly detailed and error-prone activity.

Memory leaks and similar memory errors in threads are also difficult to identify and diagnose in parallel execution. Memory leaks are fairly common in C and C++ code as memory is allocated for use but not automatically returned to the free memory list. These are especially difficult to detect because they can occur in any running thread, and typically don't manifest themselves during the short test runs often performed by developers and testers. Rather, lost memory accumulates over time, initially slowing down application execution as the heap becomes larger, and finally causing program crashes as the heap exceeds allocated memory.

THE RACE CONDITION

If multiple threads are running simultaneously and the result is dependent upon which thread finishes first, it is known as a race condition. The threads exchange data during execution, and that data may be different, depending on when a particular thread is running, and how far along it is in executing.

Of course, the threads are not supposed to be exchanging data while they are supposedly executing independently. However, the protections provided in the code—usually some form of a mutex or critical section—are either absent or not sufficient, allowing data to be exchanged among threads while they should be in their critical sections.

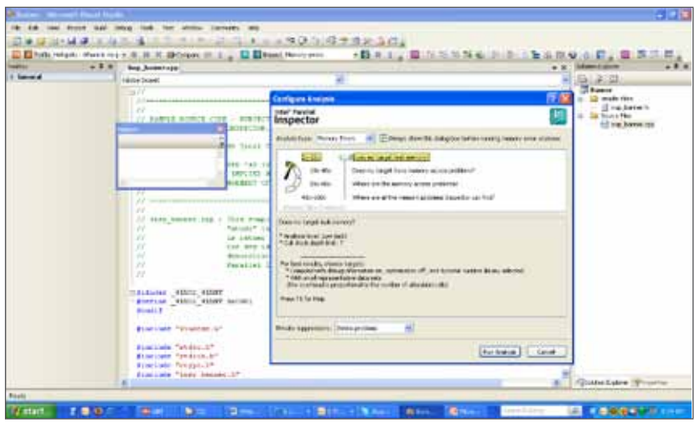


Figure 1. Intel® Parallel Studio's Intel® Parallel Inspector enables developers to identify memory leaks and other memory errors in running multithreaded applications.

This is called a race condition, where the race of threads to completion affects the result of the computation. It is among the most insidious and difficult errors to find in programming, in part because it may only occur some of the time. This turns one of the most fundamental of tenets of computer execution on its head—that anything a computer does is deterministic.

In reality, unless an error occurs all of the time, it is almost impossible to identify and analyze in code. Race conditions can only be found almost by accident—stumbled upon because developers know that something is wrong, yet are not able to localize that issue. Tools that provide developers with an in-depth and detailed look at thread execution may be the only way to identify these and other highly technical and difficult-to-identify issues in multithreaded code.

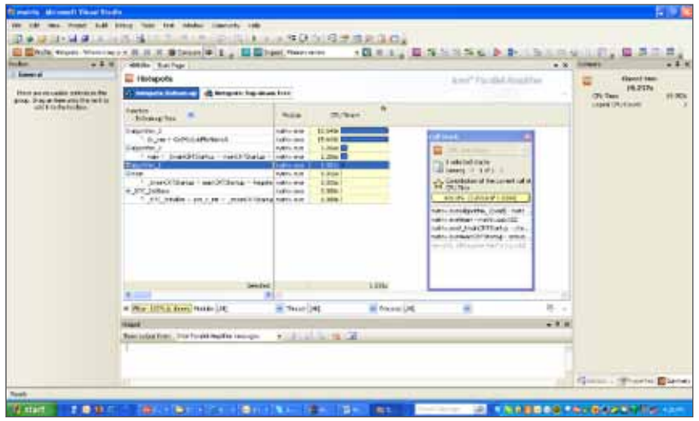


Figure 2. Intel® Parallel Amplifier provides a way for developers to analyze the performance of their applications from the standpoint of multithreaded execution. It provides a performance profiler, as well as the ability to analyze idle time and bottlenecks.

INTEL HELPS FIND PARALLEL BUGS

Intel Parallel Studio's components enable developers to build multithreaded applications, which are better able to take advantage of multiple processors or processor cores, than single threaded ones. This plug-in to Microsoft Visual Studio* provides memory analysis, thread analysis, debugging, threaded libraries, and performance analysis for these complex programs.

Intel Parallel Inspector lets developers analyze memory use in multithreaded applications. Memory is a resource that is commonly misused in single-threaded applications, as developers allocate memory and don't reclaim that memory after the operations have been completed. Finding and fixing these errors in multithreaded programs can be a difficult and time-consuming task.

Intel Parallel Inspector can also help developers detect and analyze threading errors such as deadlock, and will also help detect race conditions. It does so by observing the behavior of individual threads and how they use and free memory. For race conditions, it can look at what threads are completing first and how that affects results.

Intel Parallel Amplifier is first and foremost a performance profiler—it tells you where your code is spending most of its time—the hotspots, so to speak. But it does more than that. It also looks at where a program or its threads are waiting, and also looks at where concurrency might not be performing satisfactorily. In other words, it would identify idle threads, waiting within the program, and resources that remain idle for inordinate lengths of time.

Intel Parallel Composer is the cornerstone to the package. The parallel libraries and Intel Threading Building Blocks C++ template library enables developers to abstract threads to tasks and to more easily create multithreaded applications that have the ability to execute in parallel. These libraries provide developers with a variety of threaded generic and application-specific functions, which enable developers to quickly add parallelism to applications.

Intel Parallel Composer also provides a debugger specifically designed for working with multithreaded applications, and a C++ compiler to build code.

Any one of these tools by itself isn't sufficient to assist greatly in the development of multithreaded code for parallel execution. Writing multithreaded code can occur without existing, debugged libraries for parallel execution. Finding memory leaks is possible without a multithreading memory tool. And analyzing deadlock and race conditions is conceivable without specialized tools for looking at thread execution. But to write, analyze, and debug multithreaded code intended for high-performance operation on multicore processors can't easily be done without the ability to easily add parallel functions, debug those functions, look for deadlock and race conditions, and analyze performance.

Intel Parallel Studio provides the software tools for doing all of this, within the context of Microsoft Visual Studio. By using Intel Parallel Studio, developers can rapidly build, analyze, and debug multithreaded applications for use in demanding environments for parallel execution.

However we might write parallel code in the future, Intel is now providing tools that might help us do so. It is the only way that we can start using the computing power the last few years have afforded us. And if we don't take advantage of that computing power, computer users are losing out on what Intel is delivering in terms of new and innovative technology.

Peter Varhol is principal at Technology Strategy Research, LLC, an industry analysis and consulting firm.

Are You Ready to Enter a Parallel Universe: **Optimizing Applications** for Multicore

"I know how to make four horses pull a cart—I don't know how to make 1,024 chickens do it." —*Enrico Clementi*

By Levent Akyil

A look at parallelization methods made possible by the new Intel® Parallel Studio—designed for Microsoft Visual Studio* C/C++ developers of Windows* applications.

The introduction of multicore processors started a new era for both consumers and software developers. While bringing vast opportunities to consumers, the increase in capabilities and processing power of new multicore processors puts new demands on developers, who must create products that efficiently use these processors. For that reason, Intel is committed to providing the software community with tools to preserve the investment it has in software development and to take advantage of the rapidly growing installed base of multicore systems. In light of this commitment, Intel introduced Intel® Parallel Studio for Microsoft Visual Studio* C/C++ developers and opened the doors of the parallel universe to a wider range of software developers. Intel Parallel Studio will not only help developers create applications that will be better positioned in the highly competitive software marketplace, but it will also add parallelism for multicore that forward scales to manycore.

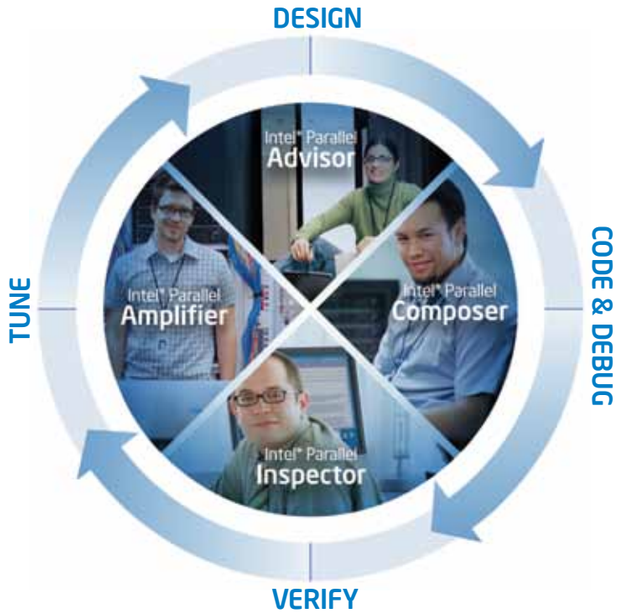


Figure 1: The four steps in parallel software development

In this article, I'll discuss Intel Parallel Studio products and the key technologies introduced for different aspects of software development (Figure 1). Intel Parallel Studio is composed of the following products: Intel® Parallel Advisor (design), Intel® Parallel Composer (code/debug), Intel® Parallel Inspector (verify) and Intel® Parallel Amplifier (tune).

Intel Parallel Composer speeds up software development by incorporating parallelism with a C/C++ compiler and comprehensive threaded libraries. By supporting a broad array of parallel programming models, a developer can find a match to the coding methods most appropriate for their application. Intel Parallel Inspector is a proactive "bug finder." It's a flexible tool that adds reliability regardless of the choice of parallelism programming models. Unlike traditional debuggers, Intel Parallel Inspector detects hard-to-find threading errors in multithreaded C/C++ Windows* applications and does root-cause analysis for defects such as data races and deadlocks. Intel Parallel Amplifier assists in fine-tuning parallel applications for optimal performance on multicore processors by helping find unexpected serialization(s) that prevents scaling.

INTEL PARALLEL COMPOSER

Intel Parallel Composer enables developers to express parallelism with ease, in addition to taking advantage of multicore architectures. It provides parallel programming extensions, which are intended to quickly introduce parallelism. Intel Parallel Composer integrates and enhances the Microsoft Visual Studio environment with additional capabilities for parallelism at the application level, such as OpenMP 3.0*, lambda functions, auto-vectorization, auto-parallelization, and threaded libraries support. The award-winning Intel® Threading Building Blocks (Intel® TBB) is also a key component of Intel Parallel Composer that offers a portable, easy-to-use, high-performance way to do parallel programming in C/C++.

Some of the key extensions for parallelism Intel Parallel Composer brings are:

- **Vectorization support:** The automatic vectorizer (also called the auto-vectorizer) is a component of the Intel® compiler that automatically uses SIMD (Single Instruction Multiple Data) instructions in the MMX™, Intel® Streaming SIMD Extensions (Intel® SSE, SSE2, SSE3 and SSE4 Vectorizing Compiler and Media Accelerators) and Supplemental Streaming SIMD Extensions (SSSE3) instruction sets.
- **OpenMP 3.0 support:** The Intel compiler performs transformations to generate multithreaded code based on a developer's placement of OpenMP directives in the source program. The Intel compiler supports all of the current industry-standard OpenMP directives and compiles parallel programs annotated with OpenMP directives. The Intel compiler also provides Intel-specific extensions to the OpenMP Version 3.0 specification, including runtime library routines and environment variables. Using /Qopenmp switch enables the compiler to generate multithreaded code based on the OpenMP directives. The code can be executed in parallel on both uniprocessor and multiprocessor systems.
- **Auto-parallelization feature:** The auto-parallelization feature of the Intel compiler automatically translates serial portions of the input program into equivalent multithreaded code. Automatic parallelization determines the loops that are good work-sharing candidates, and performs the dataflow analysis to verify correct parallel execution. It then partitions the data for threaded code generation as needed in programming with OpenMP directives. By using /Qparallel, compiler will try to auto-parallelize the application.
- **Intel Threading Building Blocks (Intel TBB):** Intel TBB is an award-winning runtime-based parallel programming model, consisting of a template-based runtime library to help developers harness the latent performance of multicore processors. Intel TBB allows developers to write scalable applications that take advantage of concurrent collections and parallel algorithms.
- **Simple concurrent functionality:** Four keywords (__taskcomplete, __task, __par, and __critical) are used as statement prefixes to enable a simple mechanism to write parallel programs. The keywords are implemented using OpenMP runtime support. If you need more control over parallelization of your program, use OpenMP features directly. In order to enable this functionality, use /Qopenmp compiler switch to use parallel execution features. The compiler driver automatically links in the OpenMP runtime support libraries. The runtime system manages the actual degree of parallelism.
- **C++ lambda function support:** C++ lambda expressions are primary expressions that define function objects. Such expressions can be used wherever a function object is expected; for example, as arguments to Standard Template Library (STL) algorithms. The Intel compiler implements lambda expressions as specified in the ISO C++ document, which is available at

EXAMPLE

```
void sum (int length, int *a, int *b, int *c )
{
 int i;
 for (i=0; i<length; i++)
            c[i] = a[i] + b[i];
}

//Serial call
sum(1000, a, b, c);
```

```
// Using concurrent functionality
__taskcomplete
{
 __task sum(500, a, b, c);
 __task sum(500, a+500, b+500, c+500);
}
```

www.open-std.org/jtc1/sc22/wg21/docs/papers/2008/.
By using */Qstd=c++0x* switch, lambda support can be enabled.

- **Intel® Integrated Performance Primitives (Intel® IPP):** Now part of Intel Parallel Composer, Intel IPP is an extensive library of multicore-ready, highly optimized software functions for multimedia data processing and communications applications. Intel IPP provides optimized software building blocks to complement Intel compiler and performance optimization tools. Intel IPP provides basic low-level functions for creating applications in several domains, including signal processing, audio coding, speech recognition and coding, image processing, video coding, operations on small matrices, and 3-D data processing.
- **Valarray:** Valarray is a C++ standard template (STL) container class for arrays consisting of array methods for high-performance computing. The operations are designed to exploit hardware features such as vectorization. In order to take full advantage of valarray, the Intel compiler recognizes valarray as an intrinsic type and replaces such types by Intel IPP library calls.

EXAMPLE

```
// Create a valarray of integers
valarray_t::value_type ibuf[10] = {0,1,2,3,4,5,6,7,8,9};
valarray_t vi(ibuf, 10);

// Create a valarray of booleans for a mask
maskarray_t::value_type mbuf[10] = {1,0,1,1,1,0,0,1,1,0};
maskarray_t mask(mbuf,10);

// Double the values of the masked array
vi[mask] += static_cast<valarray_t> (vi[mask]);
```

- **Intel® Parallel Debugger Extension:** The Intel Parallel Debugger Extension for Microsoft Visual Studio is a debugging add-on for the Intel® C++ compiler's parallel code development features. It doesn't replace or change the Visual Studio debugging features; it simply extends what is already available with:
  - **Thread data sharing analysis to detect accesses to identical data elements from different threads**
  - **Smart breakpoint feature to stop program execution on a re-entrant function call**
  - **Serialized execution mode to enable or disable the creation of additional worker threads in OpenMP parallel loops dynamically**
  - **Set of OpenMP runtime information views for advanced OpenMP program state analysis**
  - **SSE (Streaming SIMD Extensions) register view with extensive formatting and editing options for debugging parallel data using the SIMD (Single Instruction, Multiple Data) instruction set**

As mentioned above, the Intel Parallel Debugger extension is useful in identifying thread data sharing problems. Intel Parallel Debugger Extension uses source instrumentation in order to detect data sharing problems. To enable this feature, set /debug: parallel by enabling *Enable Parallel Debug Checks* under Configuration Properties > C/C++ > Debug. Figure 2 shows Intel Parallel Debugger Extension breaking the execution of the application upon detecting two threads accessing the same data.

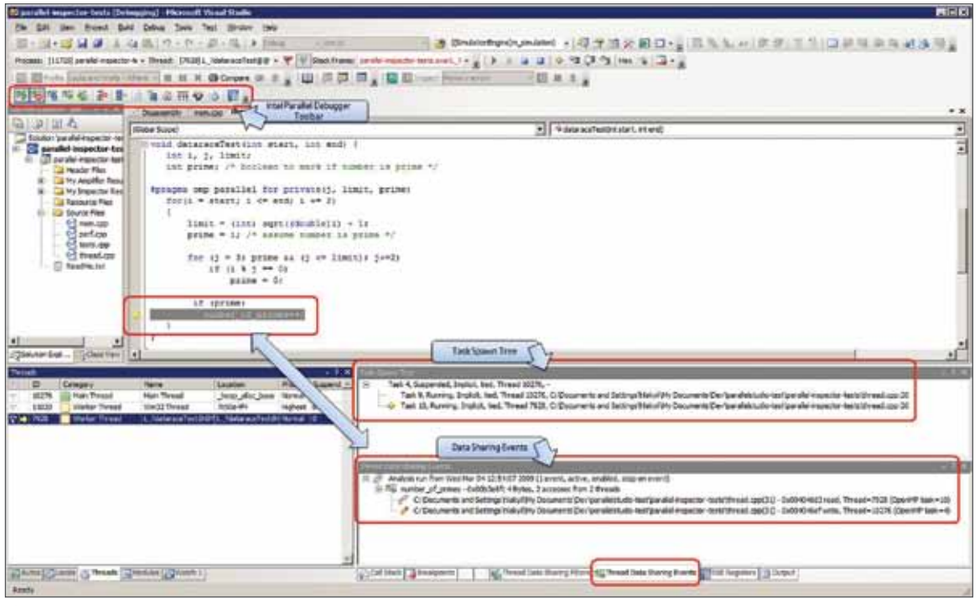


Figure 2:
Intel® Parallel Debugger Extension can break the execution upon detecting a data sharing problem

INTEL PARALLEL INSPECTOR

"It had never evidenced itself until that day ... This fault was so deeply embedded, it took them weeks of poring through millions of lines of code and data to find it."

–Ralph DiNicola
Spokesman for U.S.-Canadian task force investigating the Northeast 2003 blackout

Finding the cause of errors in multithreaded applications can be a challenging task. Intel Parallel Inspector, an Intel Parallel Studio tool, is a proactive bug finder that helps you detect and perform root-cause analysis on threading and memory errors in multithreaded applications.

Intel Parallel Inspector enables C and C++ application developers to:

- Locate a large variety of memory and resource problems including leaks, buffer overrun errors, and pointer problems
- Detect and predict thread-related deadlocks, data races, and other synchronization problems
- Detect potential security issues in parallel applications
- Rapidly sort errors by size, frequency, and type to identify and prioritize critical problems

Intel Parallel Inspector (Figure 3) uses binary instrumentation technology called Pin to check memory and threading errors. Pin is a dynamic instrumentation system provided by Intel (www.pintool.org), which allows C/C++ code to be injected into the areas of interest in a running executable. The injected code is then used to observe the behavior of the program.

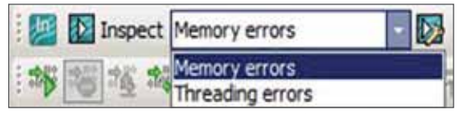


Figure 3: Intel® Parallel Inspector toolbar

INTEL PARALLEL INSPECTOR MEMORY ANALYSIS LEVELS

Intel Parallel Inspector uses Pin in different settings to provide four levels of analysis, each having different configurations and different overhead, as seen in Figure 4. The first three analysis levels are targeted for memory problems occurring on the heap while the fourth level can also analyze the memory problems on the stack. The technologies employed by Intel Parallel Inspector to support all the analysis levels are the Leak Detection (Level 1) and Memory Checking (Levels 2–4) technologies, which use Pin in various ways.

LEVEL 1 The first analysis level helps to find out if the application has any memory leaks. Memory leaks occur when a block of memory is allocated and never released.

LEVEL 2 The second analysis level detects if the application has invalid memory accesses including uninitialized memory accesses, invalid deallocation, and mismatched allocation/deallocation. Invalid memory accesses occur when a read or write instruction references memory that is logically or physically invalid. At this level, invalid partial memory accesses can also be identified. Invalid partial accesses occur when a read instruction references a block (2 bytes or more) of memory where part of the block is logically invalid.

LEVEL 3 The third analysis level is similar to the second level except that the call stack depth is increased to 12 from 1, in addition to the enhanced dangling pointer check being enabled. Dangling pointers access/point to data that no longer exist. Intel Parallel Inspector delays a deallocation when it occurs so that the memory is not available for reallocation (it can't be returned by another allocation request). Thus, any references that follow the deallocation can be guaranteed as invalid references from dangling pointers. This technique requires additional memory and the memory used for the delayed deallocation list is limited; therefore Intel Parallel Inspector must eventually start actually deallocating the delayed references.

LEVEL 4 The fourth analysis level tries to find all memory problems by increasing the call stack depth to 32, enabling enhanced dangling pointer check, including system libraries in the analysis, and analyzing the memory problems on the stack. The stack analysis is only enabled at this level.

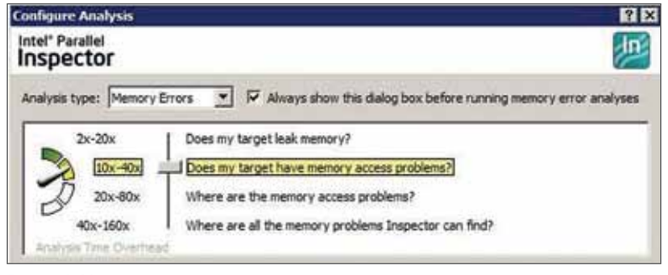


Figure 4: Memory errors analysis levels

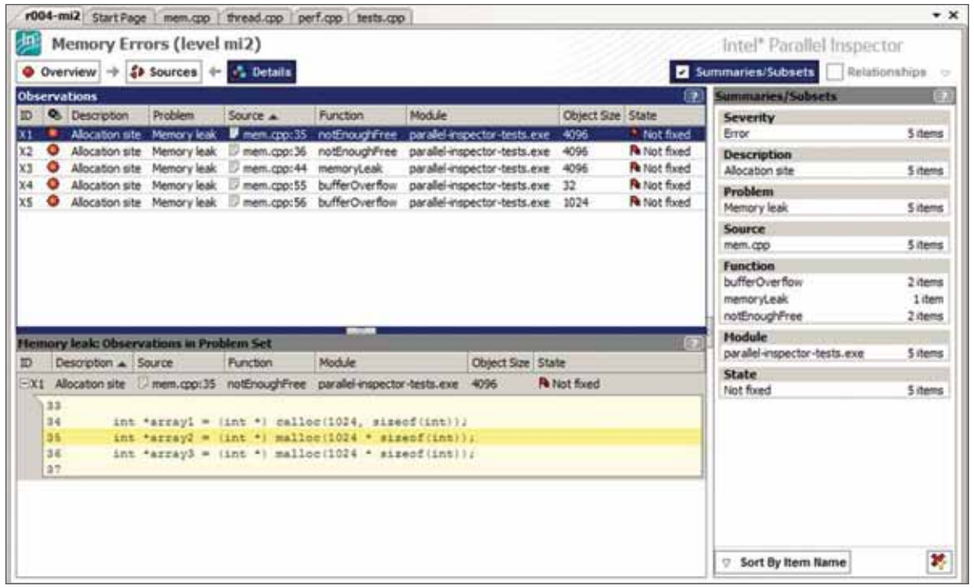


Figure 5: Intel® Parallel Inspector analysis result showing memory errors found

As seen in Figure 5, it is possible to filter the results by the severity, problem description, source of the problem, function name, and the module.

INTEL PARALLEL INSPECTOR THREADING ERRORS ANALYSIS LEVELS

Intel Parallel Inspector also provides four levels of analysis for threading errors (Figure 6).

LEVEL 1 The first level of analysis helps determine if the application has any deadlocks. Deadlocks occur when two or more threads wait for the other to release resources such as mutex, critical section, thread handle, and so on, but none of the threads releases the resources. In this scenario, no thread can proceed. The call stack depth is set to 1.

LEVEL 2 The second analysis level detects if the application has any data races or deadlocks. Data races are one of the most common threading errors and happen when multiple threads access the same memory location without proper synchronization. The call stack depth is also 1 in this level. The byte level granularity for this level is 4.

LEVEL 3 Like the previous level, Level 3 tries to find data races and deadlocks, but additionally tries to detect where they occur. The call stack depth is set to 12 for finer analysis. The byte level granularity for this level is 1.

LEVEL 4 The fourth level of analysis tries to find all threading problems by increasing the call stack depth to 32, and by analyzing the problems on the stack. The stack analysis is only enabled at this level. The byte level granularity for this level is 1.

The main threading errors Intel Parallel Inspector identifies are data races, (Figure 7, p.13), deadlocks, lock hierarchy violations, and potential privacy infringement.

Data races can occur in various ways. Intel Parallel Inspector will detect write-write, read-write, and write-read race conditions:

- Write-write data race condition occurs when two or more threads write to the same memory location
- Read-write race condition occurs when one thread reads from a memory location, while another thread writes to it concurrently
- Write-read race condition occurs when one thread writes to a memory location, while a different thread concurrently reads from the same memory location

In all cases, the order of the execution will affect the data that is shared.

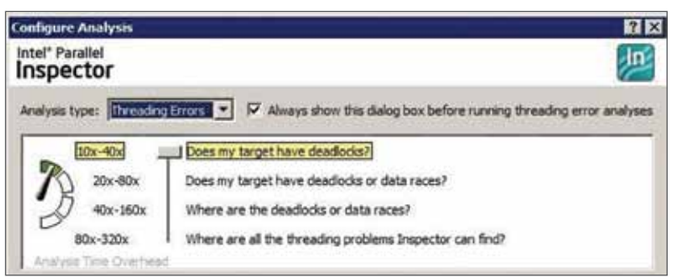


Figure 6: Threading errors analysis levels

Welcome to the Parallel Universe

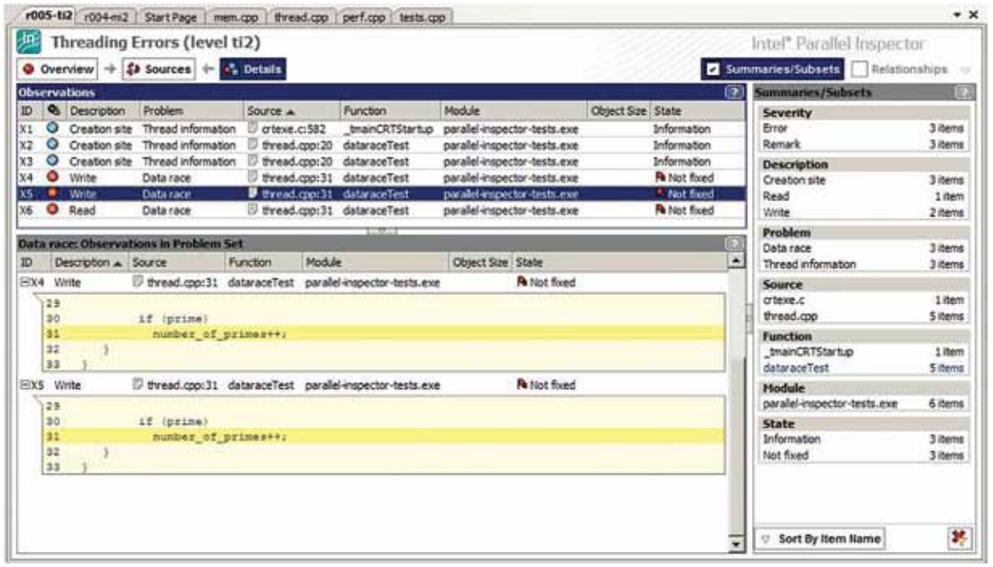


Figure 7: Intel® Parallel Inspector analysis result showing data race issues found

INTEL PARALLEL AMPLIFIER

> "Programmers waste enormous amounts of time thinking about, or worrying about, the speed of noncritical parts of their programs ... We should forget about small efficiencies, say about 97 percent of the time: premature optimization is the root of all evil."
>
> *—Donald Knuth (adapted from C. A. R. Hoare)*

Multithreaded applications tend to have their own unique sets of problems due to the complexities introduced by parallelism. Converting a serial code base to thread-safe code is not an easy task. It usually has an impact on development time, and results in increasing the complexity of the existing serial application. The common multithreading performance issues can be summarized in a nutshell as follows:

- Increased complexity (data restructuring, use of synchronization)
- Performance (requires optimization and tuning)
- Synchronization overhead

In keeping with Knuth's advice, Intel Parallel Amplifier (Figure 8) can help developers identify the bottlenecks of their code for optimization that has the most return on investment (ROI). Identifying the performance issues in the target application

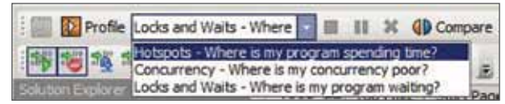


Figure 8: Intel® Parallel Amplifier toolbar

and eliminating them appropriately is the key to an efficient optimization.

With a single mouse click, Intel Parallel Amplifier can perform three powerful performance analyses. These analysis types are known as hotspot analysis, concurrency analysis, and locks and waits analysis. Before explaining each analysis level it is beneficial to explain the metrics used by Intel Parallel Amplifier.

Elapsed Time The elapsed time is the amount of time the application executes. Reducing the elapsed time for an application when running a fixed workload is one of the key metrics. The elapsed time for the application is reported in the summary view.

CPU Time The CPU time is the amount of time a thread spends executing on a processor. For multiple threads, the CPU time of the threads is aggregated. The total CPU time is the sum of the CPU time of all the threads that run during the analysis.

Wait Time The wait time is the amount of time that a given thread waited for some event to occur. These events can be events such as synchronization waits and I/O waits.

HOTSPOT ANALYSIS

By using a low overhead statistical sampling (also known as stack sampling) algorithm, hotspot analysis (Figure 9) helps the developer understand the application flow and identify the sections of code that took a long time to execute (hotspots). During hotspot analysis, Intel Parallel Amplifier profiles the application by sampling at certain intervals using the OS timer. It collects samples of all active instruction addresses with their call sequences upon each sample. Then it analyzes and displays these stored sampled instruction pointers (IP), along with the associated call sequences. Statistically collected IP samples with call sequences enable Intel Parallel Amplifier to generate and display a call tree.

CONCURRENCY ANALYSIS

Concurrency analysis measures how an application utilizes the available processors on a given system. The concurrency analysis helps developers identify hotspot functions where processor utilization is poor, as seen in Figure 10. During the concurrency analysis, Intel Parallel Amplifier collects and provides information on how many threads are active, meaning threads that are either running or are queued and are not waiting at a defined waiting or blocking API. The number of running threads corresponds to the concurrency level of an application. By comparing the concurrency level with the number of processors, Intel Parallel Amplifier classifies how the application utilizes the processors in the system.

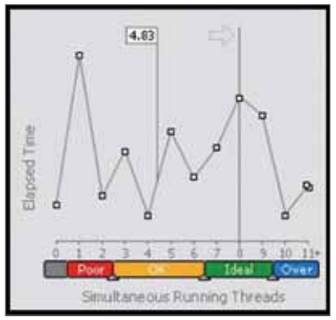


Figure 11: Intel® Parallel Amplifier concurrency analysis results summary view

The time values in the concurrency and locks and waits windows correspond to the following utilization types (Figure 11):

Idle: All threads in the program are waiting—no threads are running. There can be only one node in the Summary tab graph indicating idle utilization.

Poor: Poor utilization. By default, poor utilization is when the number of threads is up to 50% of the target concurrency.

OK: Acceptable (OK) utilization. By default, OK utilization is when the number of threads is between 51% and 85% of the target concurrency.

Ideal: Ideal utilization. By default, ideal utilization is when the number of threads is between 86% and 115% of the target concurrency.

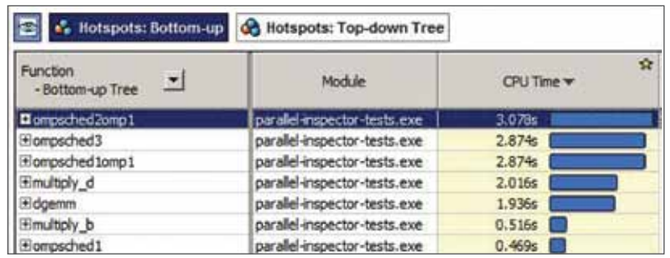


Figure 9: Intel® Parallel Amplifier Hotspot analysis results

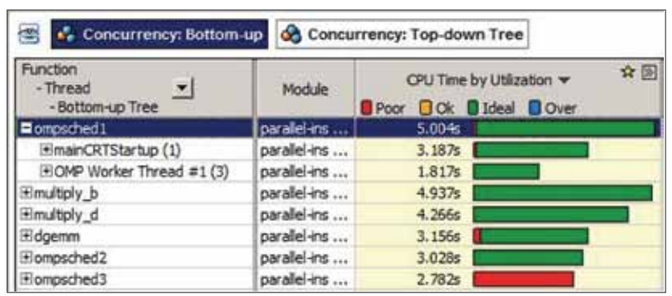


Figure 10: Intel® Parallel Amplifier concurrency analysis results. Granularity is set to Function-Thread

LOCKS AND WAITS ANALYSIS

While concurrency analysis helps developers identify where their application is not parallel or not fully utilizing the available processors, locks and waits analysis helps developers identify the cause of the ineffective processor utilization (Figure 12, p.15). The most common problem for poor utilization is caused by threads waiting too long on synchronization objects (locks). In most cases no useful work is done; as a result, performance suffers, resulting in low processor utilization.

During locks and waits analysis, developers can estimate the impact of each synchronization object. The analysis results help to understand how long the application was required to wait on each synchronization object, or in blocking APIs, such as sleep and blocking I/O.

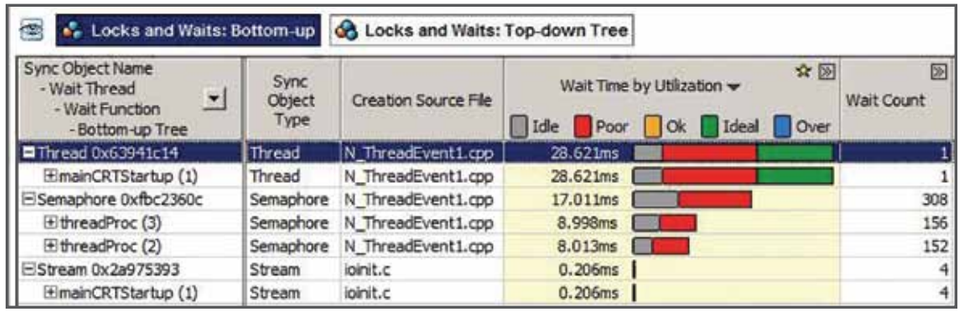


Figure 12: Locks and waits analysis results

The synchronization objects analyzed can be given as mutexes (mutual exclusion objects), semaphores, critical sections, and fork-joins operations. A synchronization object with the longest waiting time and high concurrency level is very likely to be a bottleneck for the application.

It is also very important to mention that for both Intel Parallel Inspector and Intel Parallel Amplifier, it is possible to drill down all the way to the source code level. For example, by double-clicking on a line item in Figure 13, I can drill down to the source code and observe which synchronization object is causing the problem.

VAST OPPORTUNITIES

Parallel programming is not new. It has been well studied and has been employed in the high-performance computing community for many years, but now, with the expansion of multicore processors, parallel programming is becoming mainstream. This is exactly where Intel Parallel Studio comes into play. Intel Parallel Studio brings vast opportunities and tools that ease the developers' transition to the realm of parallel programming and hence significantly reduce the entry barriers to the parallel universe. Welcome to the parallel universe.

Levent Akyil is Staff Software Engineer in the Performance, Analysis, and Threading Lab, Software Development Products, Intel Corporation.

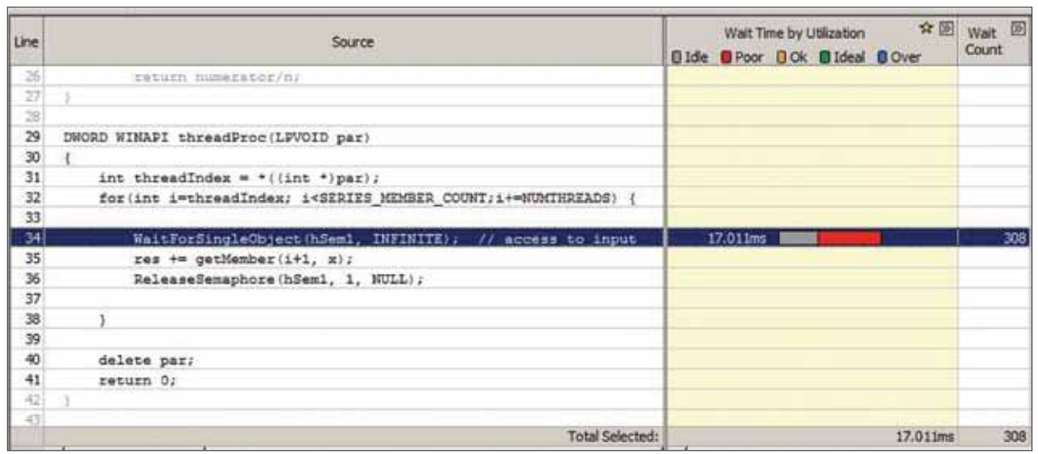


Figure 13: Source code view of a problem in locks and waits analysis

8 Rules for Parallel Programming for **Multicore**

By James Reinders

There are some consistent rules that can help you solve the parallelism challenge and tap into the potential of multicore.

Programming for multicore processors poses new challenges. Here are eight rules for multicore programming to help you be successful:

1 Think parallel. Approach all problems looking for the parallelism. Understand where parallelism is, and organize your thinking to express it. Decide on the best parallel approach before other design or implementation decisions. Learn to "Think Parallel."

2 Program using abstraction. Focus on writing code to express parallelism, but avoid writing code to manage threads or processor cores. Libraries, OpenMP*, and Intel® Threading Building Blocks (Intel® TBB) are all examples of using abstractions. Do not use raw native threads (Pthreads, Windows* threads, Boost threads, and the like). Threads and MPI are the assembly languages for parallelism. They offer maximum flexibility, but require too much time to write, debug, and maintain. Your programming should be at a high enough level that your code is about your problem, not about thread or core management.

3 Program in tasks (chores), not threads (cores). Leave the mapping of tasks to threads or processor cores as a distinctly separate operation in your program, preferably an abstraction you are using that handles thread/core management for you. Create an abundance of tasks in your program, or a task that can be spread across processor cores automatically (such as an OpenMP loop). By creating tasks, you are free to create as many as you can without worrying about oversubscription.

4 Design with the option to turn concurrency off. To make debugging simpler, create programs that can run without concurrency. This way, when debugging, you can run programs first with—then without—concurrency, and see if both runs fail or not. Debugging common issues is simpler when the program is not running concurrently because it is more familiar and better supported by today's tools. Knowing that something fails only when run concurrently hints at the type of bug you are tracking down. If you ignore this rule and can't force your program to run in only one thread, you'll spend too much time debugging. Because you want to have the capability to run in a single thread specifically for debugging, it doesn't need to be efficient. You just need to avoid creating parallel programs that require concurrency to work correctly, such as many producer-consumer models. MPI programs often violate this rule, which is part of the reason MPI programs can be problematic to implement and debug.

5 Avoid using locks. Simply say "no" to locks. Locks slow programs, reduce their scalability, and are the source of bugs in parallel programs. Make implicit synchronization the solution for your program. When you still need explicit synchronization, use atomic operations. Use locks only as a last resort. Work hard to design the need for locks completely out of your program.

6 Use tools and libraries designed to help with concurrency. Don't "tough it out" with old tools. Be critical of tool support with regard to how it presents and interacts with parallelism. Most tools are not yet ready for parallelism. Look for thread-safe libraries—ideally ones that are designed to utilize parallelism themselves.

7 Use scalable memory allocators. Threaded programs need to use scalable memory allocators. Period. There are a number of solutions, and I'd guess that all of them are better than malloc(). Using scalable memory allocators speeds up applications by eliminating global bottlenecks, reusing memory within threads to better utilize caches, and partitioning properly to avoid cache line sharing.

8 Design to scale through increased workloads. The amount of work your program needs to handle increases over time. Plan for that. Designed with scaling in mind, your program will handle more work as the number of processor cores increases. Every year, we ask our computers to do more and more. Your designs should favor using increases in parallelism to give you advantages in handling bigger workloads in the future.

I wrote these rules with implicit mention of threading everywhere. Only rule no. 7 is specifically related to threading. Threading is not the only way to get value out of multicore. Running multiple programs or multiple processes is often used, especially in server applications.

These rules will work well for you to get the most out of multicore. Some will grow in importance over the next 10 years, as the number of processor cores rises and we see an increase in the diversity of the cores themselves. The coming of heterogeneous processors and NUMA, for instance, makes rule no. 3 more and more important.

You should understand all eight and take all eight to heart. I look forward to any comments you may have about these rules or parallelism in general.

James Reinders is Chief Software Evangelist and Director of Software Development Products, Intel Corporation. His articles and books on parallelism include *Intel® Threading Building Blocks: Outfitting C++ for Multicore Processor Parallelism.* Find out more at www.go-parallel.com.

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