

Writing Parallel Code Safely

By Peter Varhol

Dek: Writing multi-threaded code to take full advantage of multiple processors and multi-core processors is difficult. Intel's new Parallel Studio should help us bridge that gap.

With the explosion of multi-core 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 multi-threading 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 multi-core processors.

But multiple processor systems and multi-core 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 sixteen 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 multi-core processors and multiple processors. For example, Intel Parallel Studio provides essential tools for working with multi-threaded 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 <http://software.intel.com/en-us/intel-parallel-studio-home/>. It consists of three components – Intel Parallel Composer, Intel Parallel Inspector, and Intel Parallel Amplifier. Together, these tools enable developers building multi-threaded 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

Parallel Inspector detects threading and memory errors and provides guidance to help ensure application reliability. Parallel Amplifier is a performance profiler that makes it straightforward to quickly find multi-core 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 multi-core 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 multi-processor and multi-core 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, then they 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.

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 – stumbling upon it because developers know that there is something wrong, yet 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 multi-threaded code.

Intel Helps Find Parallel Bugs

Intel Parallel Studio's three components enable developers to build multi-threaded 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 multi-threaded 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 multi-threaded programs can be a difficult and time consuming task. Without tools, it can be impossible to find, analyze, and fix these errors in any reasonable timeframe.

The 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.

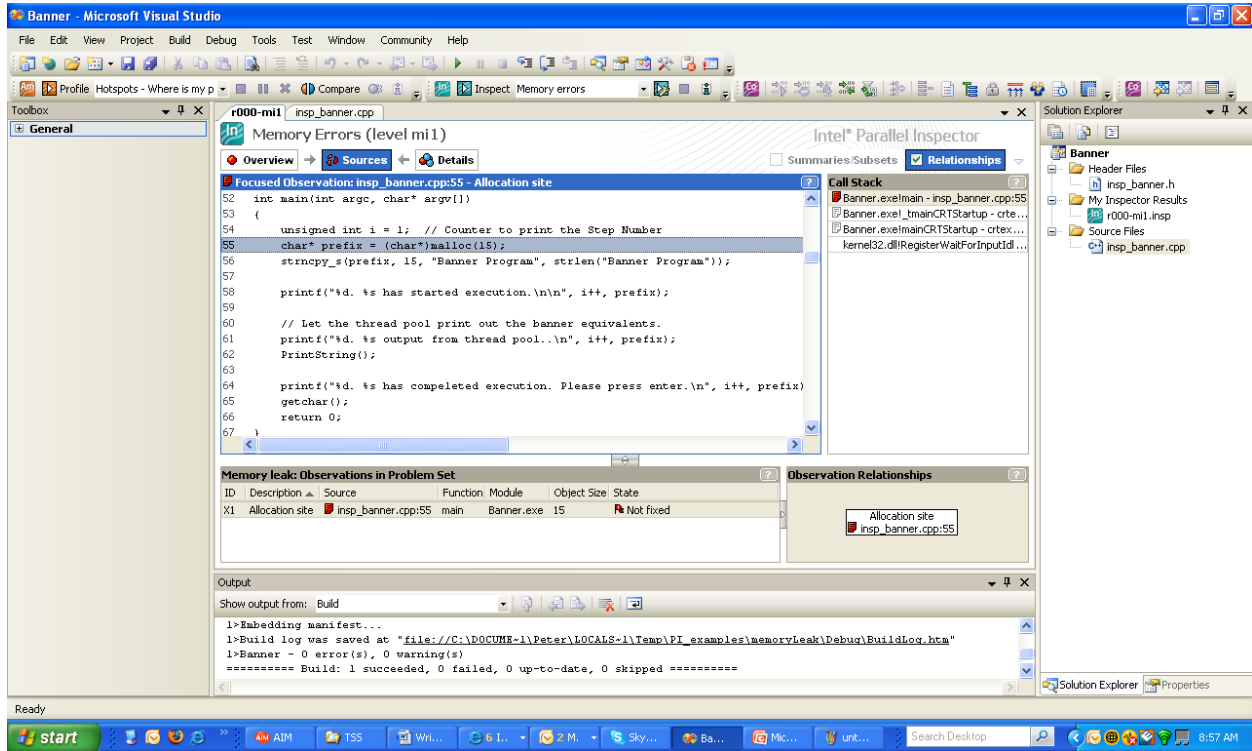


Figure 1. Intel Parallel Studio's Parallel Inspector enables developers to identify memory leaks and other memory errors in running multi-threaded applications.

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.

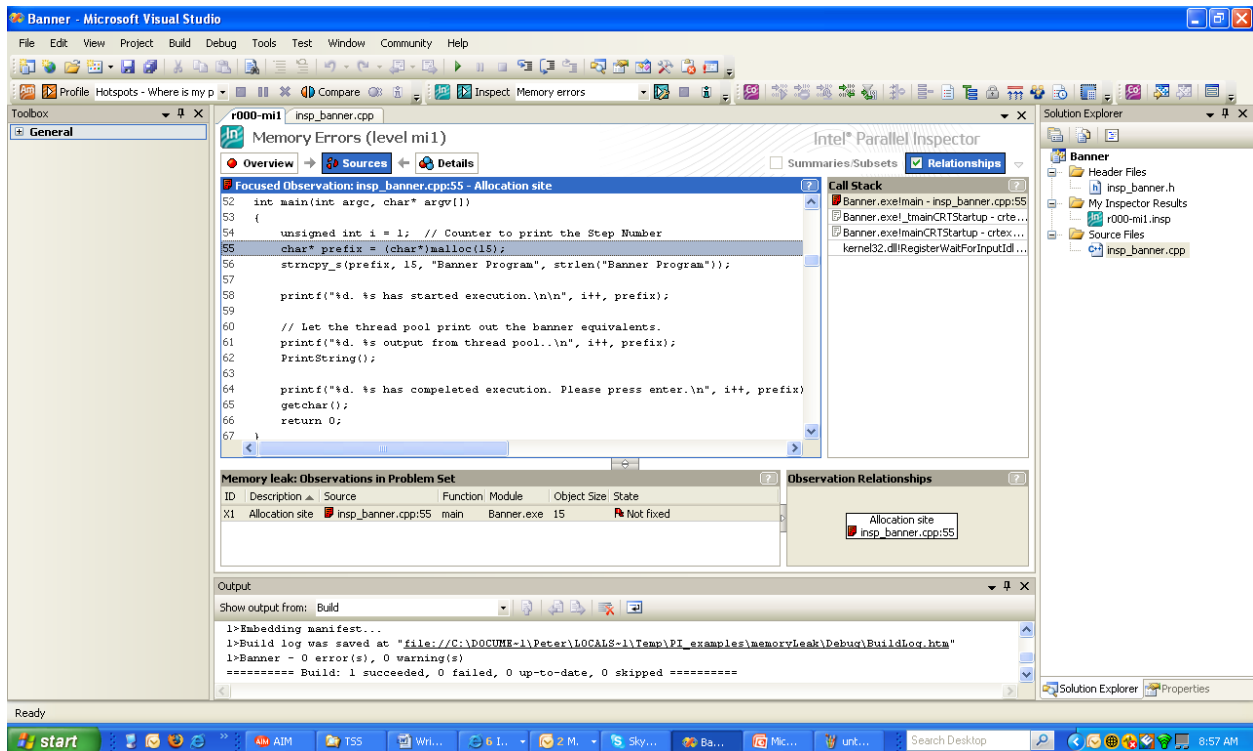


Figure 2. Intel Parallel Amplifier provides a way for developers to analyze the performance of their applications from the standpoint of multi-threaded execution. It provides a performance profiler, as well as the ability to analyze idle time and bottlenecks.

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

Parallel Composer also provides a debugger specifically designed for working with multi-threaded applications.

Any one of these tools, by itself, isn't sufficient to assist greatly in the development of multi-threaded code for parallel execution. Writing multi-threaded code can occur without existing, debugged libraries for parallel execution. Finding memory leaks is possible without a multi-threading memory tool. And analyzing deadlock and race conditions is conceivable without specialized tools for looking at thread execution. But to write, analyze, and debug multi-threaded code intended for high-performance operation on multi-core 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 Parallel Studio, developers can rapidly build, analyze, and debug multi-threaded applications for use in demanding environments for parallel execution. No other product lets developers work as comprehensively with multi-threaded applications.

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.