Welcome to the Parallel Universe

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

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++ 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 add parallelism for multicore that forward scales to manycore.

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/verify), 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 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 (SIMD3) 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.

Intel Parallel Composer is a high-performance tool that offers a simple mechanism to write scalable applications that take advantage of concurrent collections and parallel algorithms. Intel Parallel Composer 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++.

For more information, visit: https://software.intel.com/en-us/products/intel-parallel-studio-xe

---

**Figure 1:** The four steps in parallel software development

1. **Design:** Intel Parallel Advisor 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.
2. **Tune:** Intel Parallel Amplifier assists in fine-tuning parallel applications for optimal performance on multicore processors.
3. **Verify:** Intel Parallel Inspector detects and reports threading errors in multithreaded C/C++ Windows® applications.
4. **Code & Debug:** Intel Parallel Composer enables developers to express parallelism with ease, in addition to taking advantage of multicore architectures.

---

**EXAMPLE**

```c
void sum(int length, int *a, int *b, int *c)
{
    int i;

    for (i = 0; i < length; i++)
        c[i] = a[i] + b[i];

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

---

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.

---

**CODE & DEBUG**

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. 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++.

---

**VErify**

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

---

**Tune**

// Using concurrent functionality
__taskcomplete(__tasksum(500, a+500, b+500, c+500));
```
**Welcome to the Parallel Universe**

www.open-std.org/jtc1/sc22/wg21/docs/papers/2008/

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++ > 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](image)

**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 A. Harvey

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](image)

**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 access occurs 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 block 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](image)
Write-write data race condition occurs when two or more threads write to 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 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.

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 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 another thread concurrently reads from the same memory location.

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

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 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.

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. 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.

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.

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.

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