THE PARALLEL UNIVERSE

Issue 2
March 2010

Where Are My Threads?

Intel® VTune™ Performance Analyzer and Finding Threading and Parallelism Issues

by Levent Akyil

DEVELOPER ROCK STAR:
Levent Akyil

Letter from the Editor

by James Reinders

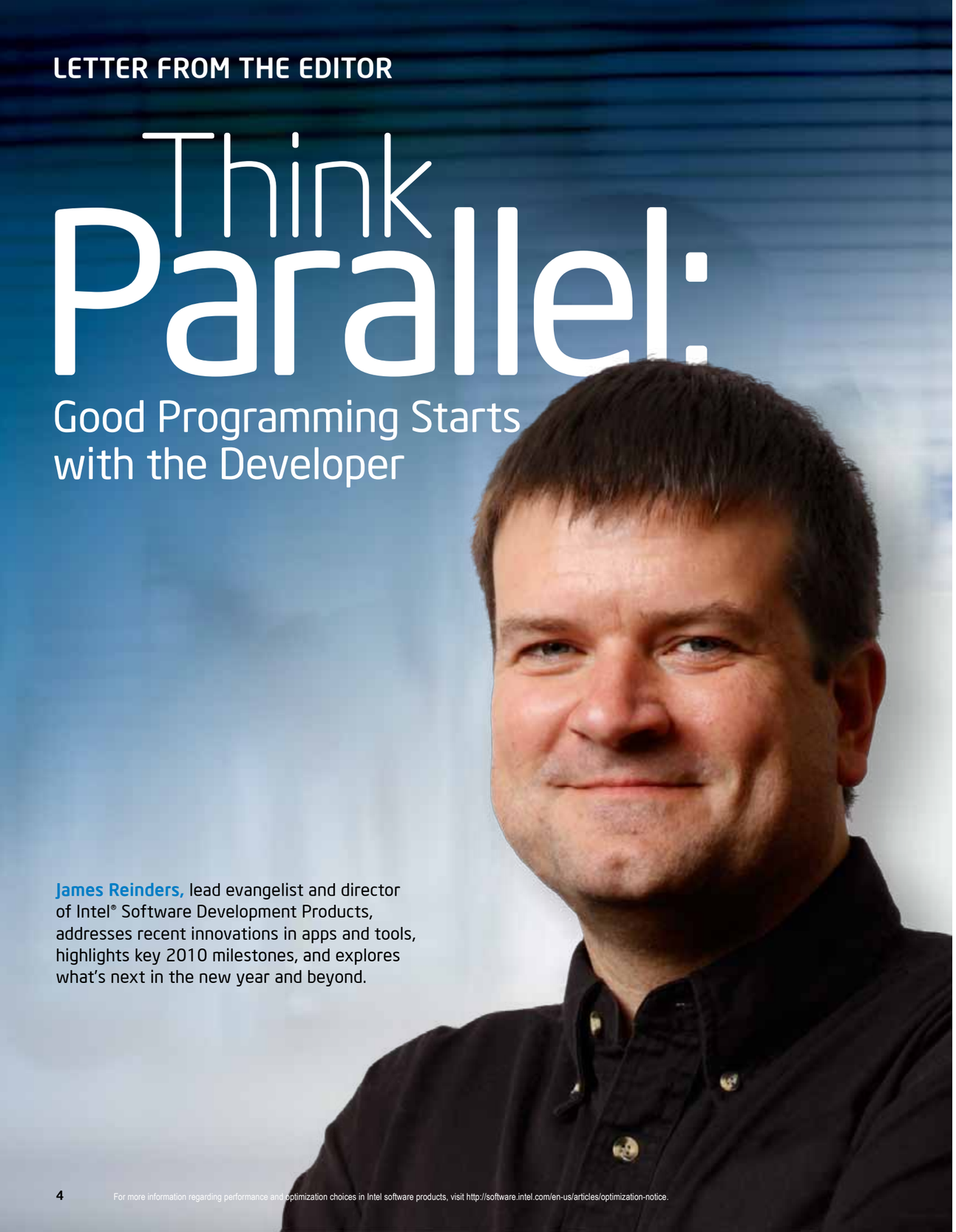
Advisor Origins

by Paul Petersen

CONTENTS

Think Parallel:

Good Programming Starts with the Developer



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.

It is no surprise that Intel® Parallel Studio is consistently cited in articles and analyst reports about software that helps with parallel programming. Of course, Intel has a long history with high performance computing (HPC) through efforts with OpenMP* and MPI*. Recently, Intel led non-HPC efforts on projects like Intel® Threading Building Blocks (Intel® TBB) and Intel® Parallel Studio.

Offering real assistance for multicore programming has meant addressing modification of existing applications, addressing ease-of-use issues, and supplying tools to aid in the entire process of designing a program. To these ends, Intel started with Intel TBB to address the many challenges of using C++—designed in an age of single-processor systems—in a multiprocessor and multicore processor world.

Encouraged by the success of Intel TBB, Intel addressed the whole design cycle with Intel Parallel Studio. By not just focusing on the alluring topic of language extensions for parallelism, Intel has made significant progress on debugging and tuning issues that have long perplexed parallel programming. Beyond helping debug the most common parallel programming issues, deadlocks, and data races, Intel found key advances to address memory leak errors, which have proven harder to debug in a parallel program.

Serious advances here have proven immensely helpful for development teams striving to have predictable schedules and results. It is not a stretch to say that without these new tools from Intel, a foray into parallel programming is much less likely to succeed.

Legacy, education, and tools were the three key needs identified in a recent joint Intel-Microsoft* customer roundtable on parallelism that I was fortunate enough to attend. Intel's alignment with our activities and these needs are the strongest that I see in the industry.

Efforts include:

- Helping with legacy applications by making it possible to add parallelism into existing programs without unreasonable changes
- Helping address multicore opportunities through software developer education when truly needed, while developing, where possible, better approaches to avoid the need for additional education

Encouraged by the success of Intel TBB, Intel addressed the whole design cycle with Intel® Parallel Studio.

Both the Go-Parallel site and the multicore portal on the Intel® Software Developer Network offer a rich set of forums and training materials used by many developers to hone their knowledge and skills. And Intel® software tools, long recognized for offering high-performance solutions, deliver strong support for parallel programming.

Meanwhile, Intel investments continue to yield results.

2010 marks the sixth year of both Intel shipping multicore processors and of Intel TBB. Given these milestones, in addition to Intel Parallel Studio entering its second year and the teams from Cilk Arts* and RapidMind* now at work in Intel, one might ask "What's next?"

Intel TBB has received awards and been cited as the most popular abstraction for parallelism in use. We are obviously pleased with the strong interest and many products that are produced using Intel TBB. In 2010, Intel TBB will support Microsoft's new Concurrency Runtime, a runtime designed for use by parallel models like Intel TBB to ensure a new level of compatibility. This move represents an important step forward in coordinating parallelism running on the Windows* operating system.

Intel Parallel Studio recently offered its first service pack, which added Windows 7* support and command line functionality for Intel® Parallel Inspector and Intel® Parallel Amplifier. It will also grow to include Intel® Parallel Advisor to help in the design and design evaluation steps of adding parallelism. We expect that the key insights into how to accelerate the error-prone and tedious step of deciding how to add parallelism will be put to good use by program architects.

Support for Microsoft Visual Studio* (VS) includes full support under VS 2005* and VS 2008*, and will expand to VS 2010* shortly after it is available from Microsoft. We will continue to offer developers a choice when it comes to deciding which version of VS they can use for parallel programming. We will also see tangible results from our acquisitions of a couple of great teams last year, specifically on Intel® compilers thanks to Cilk technology and on our Ct technology, available for field tests in beta form.

Of course, parallelism for high performance computing continues as well. Intel will offer the first and only OpenMP to support Microsoft® Concurrency Runtime, a new MPI library with split-rail support, MPI tools, a compiler, and an updated and expanded Intel® VTune™ Performance Analyzer. Parallel programming just keeps getting better and better.

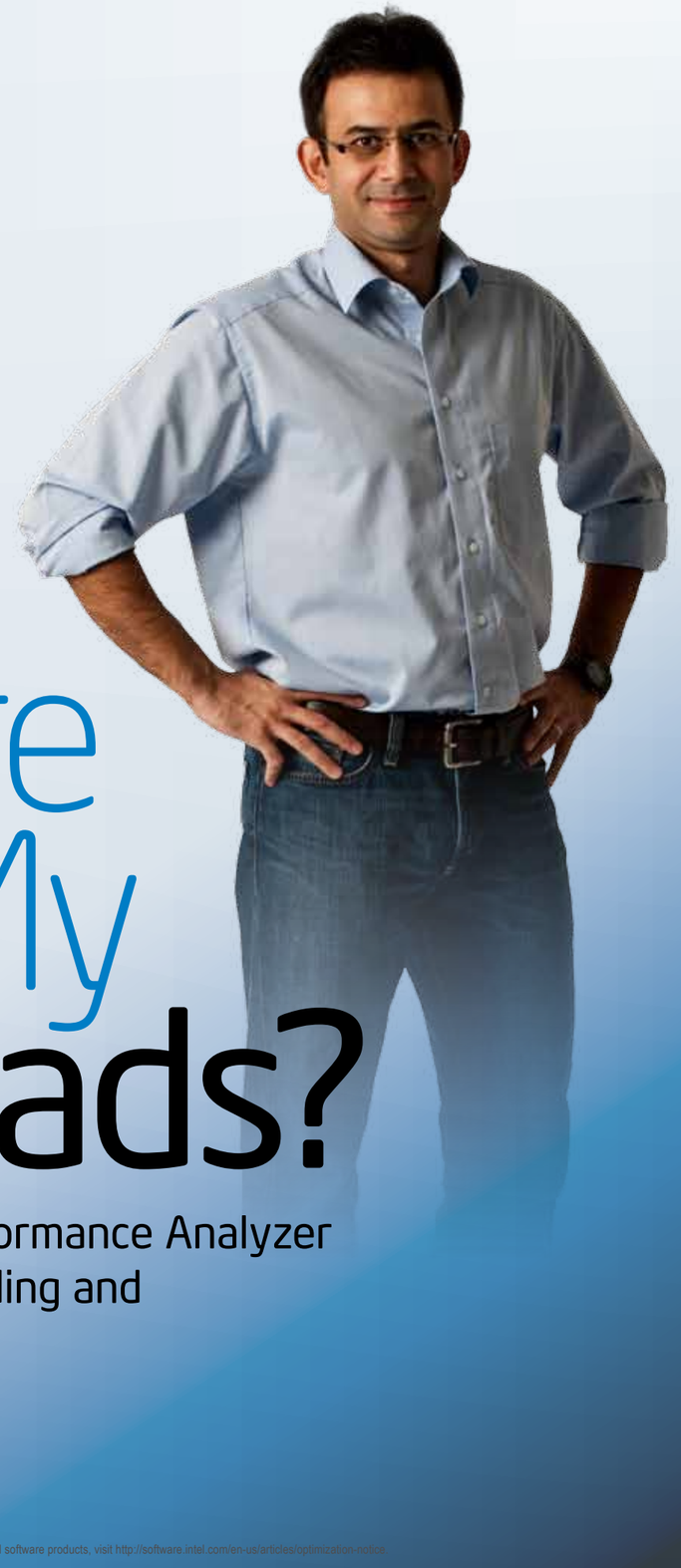
As I always like to point out, despite a great record with helping adapt legacy applications, educating developers, and providing really great help with tools, it is still up to us, as software developers, to know what to do with these wonderful tools. Just as before parallel programming, a good design comes from the human developer—not the tools. Parallel programming is no different. Therefore, we humans need to work on "Think Parallel."

JAMES REINDERS
Portland, Oregon
March 2010

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

By Levent Akyil

Do you ever wonder how your parallel workload is distributed or scheduled across the available cores/processors? Explore how Intel® VTune™ Performance Analyzer helps make such analysis easy.



Where Are My Threads?

Intel® VTune™ Performance Analyzer and Finding Threading and Parallelism Issues

The event-based sampling (EBS) technology identifies system-wide software performance problems by sampling processor events, such as clock ticks and cache misses.

One common question developers ask is how their parallel workload is distributed or scheduled across the available cores/processors.

Intel VTune Performance Analyzer helps make such analysis easy. The event-based sampling (EBS) technology identifies system-wide software performance problems by sampling processor events, such as clock ticks and cache misses (**Figure 1**). From the EBS data, you can determine which process, thread, module, function, and source line in a given application generated a particular event. By leveraging this technology you can see how many events were sampled on each core as well as which thread generated them.

The Show/Hide CPU Information button (**Figure 1**) in the sampling toolbar displays collected samples and events per processor in the Process, Thread, Module, and Hotspot sampling views (**Figure 2**).

We now know that this particular program (sort_mt1.exe) was executed on two cores and we can see the number of samples collected on each core. But what we don't know yet is how many threads this application created and how the threads executed on these cores. Selecting the Thread view (**Figure 2**) when the CPU button is also selected will show us the desired information. (**Figure 3**) tells us that sort_mt1.exe created two threads (thread18 and thread13) and that each thread was executed on both cores (OS scheduled these threads to run on each core) during the analysis. If you look at the clock ticks (i.e., CPU_CLK_UNHALTED.CORE) for thread18, it becomes clear that this particular thread was executed on each core while running most of the time on Processor 0.

If you are still curious and would like to see how these samples are distributed over time per thread and per core, the sampling over time (SOT) view can help. By selecting SOT view (**Figure 3**) in Thread view (or in any other view) the samples collected will be displayed per thread and/or core (**Figure 4**). The view seen in (**Figure 4**) is useful for many reasons.

The SOT view can help you:

- See how the operating system (OS) scheduled the threads to run.
- Identify scheduling problems (**Figure 5**).
- Identify load balancing issues among threads (**Figure 6**).
- Correlate microarchitectural problems.

The Show/Hide CPU Information button

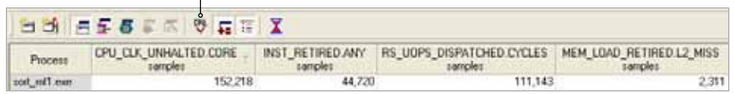


Figure 1

Thread view

Figure 2

SOT view

Figure 3

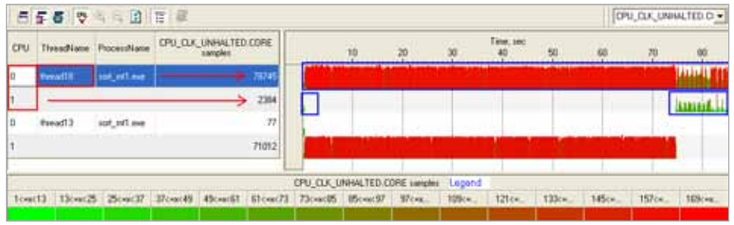


Figure 4

Selecting the Thread view button (**Figure 2**) when the CPU button is also selected provides insight into the execution and scheduling of these threads (**Figure 7**). A closer look at thread9 and thread59 and how they are executed on the cores, as shown in (**Figure 7**), reveals how the OS, Windows XP Sp3* in this particular case, is scheduling the threads on both cores. It also illustrates that each thread is running almost the same amount of time on each core.

Note: The RS_UOPS_DISPATCHED.CYCLES_NONE event shown in (**Figure 7**) counts the cycles where no μops were dispatched (stalls cycles). The overall formula is CPU_CLK_UNHALTED.CORE (clockticks) ~= RS_UOPS_DISPATCHED.CYCLES_ANY + RS_UOPS_DISPATCHED.CYCLES_NONE (stall cycles).

You can zoom in to any region on the timeline by identifying the region of interest with the mouse and selecting "Zoom In" from the context menu (right-click menu). **Figure 8**, which shows the zoomed region (0-1.8 secs), reveals how threads are actually tossed back and forth between the cores. The OS scheduler simply doesn't keep the threads on the same core (i.e., thread9 on core 0 and thread59 on core 1 or vice versa). This particular scheduling pattern might not be an issue for such a system since the cores share the same second-level cache. For multi-socket systems, however, such a scheduling pattern will be a problem.

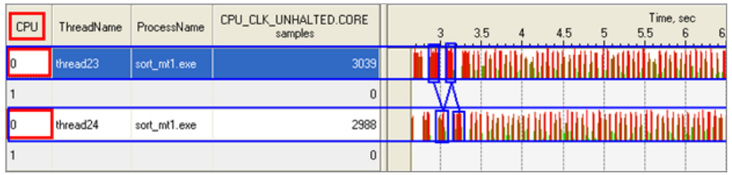


Figure 5: Manually setting thread affinity can create problems. Each thread is scheduled/pinned to the Core/Processor 0.

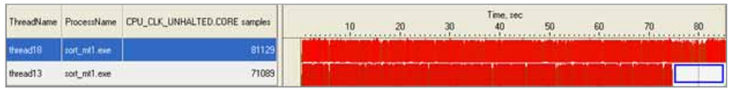


Figure 6: The SOT shows a load imbalance issue.

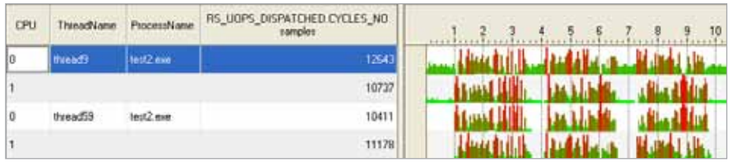


Figure 7: Identify the stall cycles per thread per cpu.

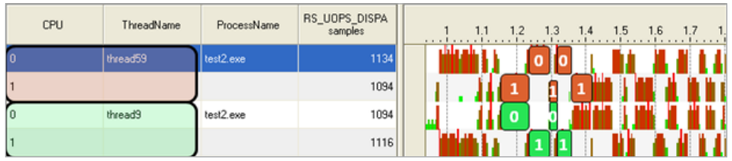


Figure 8: The SOT view shows how the OS scheduled the particular application.

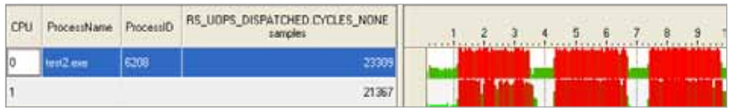


Figure 9: Note the SOT view after setting the thread affinity.

Figure 10: Results showing how setting thread affinity affects the application runtime.

You can zoom in to any region on the timeline by identifying the region of interest with the mouse and selecting "Zoom In" from the context menu (right-click menu).

Thread Affinity

At this point, it is important to introduce the concept of thread affinity. Thread affinity restricts execution of certain threads to a subset of the physical processing units in a multiprocessor computer. Depending on the topology of the machine, thread affinity can have a dramatic effect on the execution speed of an application.

However, you must have a good reason and be cautious before interfering with the OS scheduler's ability to schedule threads effectively across processors/cores. Most recent OSs and their schedulers have improved significantly; generally speaking, modern schedulers will perform efficiently.

The Intel® compiler OpenMP* runtime library has the ability to bind OpenMP threads to physical processing units. Thread affinity is supported on the Windows* OS systems and versions of the Linux* OS systems that have kernel support for thread affinity. There are three types of interfaces you can use to specify this binding. These are collectively referred to as the Intel® OpenMP* Thread Affinity Interface. For more information, click here. The affinity types supported by the Intel OpenMP runtime library are: none (default) / compact / disabled / explicit / scatter.

After Setting the Affinity

For this exercise and for this particular system, setting the affinity as "scatter" or "compact" will not make any difference. Please see the information provided in the link above for more details.

Figure 10 shows that both thread17 and thread64 remained on the same cores on which they were initially scheduled. Thread17 initially got scheduled to run on Core 0 and Core 1, but it stayed on Core 0 for the remainder of the run. □

BLOG highlights

Fun with Locks and Waits—Performance Tuning

BY DAVID MACKAY, PH.D.

At times threaded software requires some critical sections, mutexes or locks. Do developers always know which of the objects in their code have the most impact? If I want to examine my software to minimize the impact, or restructure data to eliminate some of these synchronization objects and improve performance, how do I know where I should make changes to get the biggest performance improvement? Intel® Parallel Amplifier can help me determine this.

Intel Parallel Amplifier provides three basic analysis types: hotspots (with call stack data), concurrency, and locks and waits. The locks and waits analysis highlights which synchronization objects block threads the longest. It is common for software to have too many or too few synchronization points. Insufficient synchronization points lead to race conditions and indeterminant results (if you have this problem you need Intel® Parallel Inspector, not Intel Parallel Amplifier. See this MSDN Web seminar for more on Intel Parallel Inspector: Got Memory Leaks?). If you have too many synchronization objects, you want to know which ones, if removed, would improve performance the most.

By Paul Petersen

As you change your application to make it ready to introduce parallelism, your test suite can be your biggest asset. Intel® Parallel Advisor is designed to be your assistant as you analyze your existing sequential implementations.

ADVISOR ORIGINS

A blank sheet of paper. It's frightening, and also empowering. It's a license for unlimited creative freedom. As software developers, how often do you get this opportunity? If your work is like mine, it turns out to be less often than you might think.

Most software development is adapting an existing solution to serve a new purpose. It is optimizing an existing algorithm. It is enabling new execution models for solutions customers already find valuable.

Maybe you are one of the developers who can afford to recreate your current source code base as you seek to exploit parallel execution. You likely will have the biggest payoff since you have unlimited freedom to design your algorithms and implementation to maximize the parallel execution benefit.

If you need to reuse large portions of an existing implementation, you still have a significant opportunity. In some ways, your job is much easier. Maybe you have a "diamond in the rough." You already know the "correct" definition you are trying to implement. The correct behavior is defined by the external behavior of your existing serial algorithms.

Your test suites validate your application against this correct behavior. As you change your application to make it ready to introduce parallelism, your test suite can be your biggest asset. After every transformation step you perform, the validity of the transformation can be checked by verifying your test suite application passes.

If you introduce changes to your algorithms that change the behavior of your application, it is important to determine early if these changes are desirable. In such a case, you need to update your test suite to allow this new behavior. If the behavior is in error, you need to revert these changes and go back to the prior version.

Intel® Parallel Advisor (part of Intel® Parallel Studio) is designed to be your assistant as you analyze your existing sequential implementations to discover how it can be refactored or redesigned to exploit parallel execution of your application. This article explains some of the principles upon which the design of Intel Parallel Advisor is based.

Refactoring to Enable Relaxed Sequential Execution

If your application already embodies the "correct" definition, you want to preserve it. This means that you can use refactoring techniques to uncover the latent parallelism in your application. This parallelism is latent because using a serial language to express your algorithm over-constrains the dependences that are necessary for correct execution. Refactoring is the process of changing your application's internal structure without modifying its external functional behavior. This allows you to express your intentions more clearly and eliminate the implicit serial dependences that may be present in the sequential implementation of your algorithms.

```
T1 = F1(A + B)
T2 = F2(C * D)
```

Figure 1

```
parallel {
      task { T1 = F1(A + B) }
      task { T2 = F2(C * D) }
}
```

Figure 2

In **Figure 1**, the serial semantics are that first you must add A+B, apply the function F1, and then store this result into T1. Only then do you multiply C*D, apply the function F2, and store the result into T2. Assuming F1 and F2 are pure functions, mathematically the semantics are equivalent if you calculate T2 first, and then calculate T1.

The act of writing down these two statements creates a constraint that did not exist before the two statements were written down. Optimizing compilers (and out-of-order execution hardware) try to understand the real and false dependences to enable faster execution. By declaring task boundaries (i.e., the parallel actions shown as tasks in **Figure 2**), you can use the same technique statically in the source code, indicating when implied control or data dependences do not need to be enforced to ensure correct execution.

Figure 3 shows a similar situation. This loop is shorthand for the set of assignment statements where the variable i is in the range of 0:N-1. The result generated by this loop produces each element of A containing the sum of the corresponding elements of B and C. In the serial program, this loop is over-constrained by specifying that the index variable i is calculated via the induction i=i+1. Declaring the task boundaries as shown in Figure 4 defines all iterations of the loop as logically separate tasks (capturing the value of i when each task is created).

Introducing parallelism via refactoring performs the primary task of identifying the places where the serial semantics are over-constraining the problem you need to solve. Designing your parallel application via refactoring creates the key property that the original serial execution is just one trivial execution of the parallel program. To see that this is true, consider a parallel program. What happens when you execute this program on a single thread? If you have only relaxed or removed artificial serial dependences, then adding them back in by executing the parallel program on a single thread preserves the same behavior.

When you don't care about the order of execution, the program can execute in parallel. When you do care about the order of execution (e.g., the next statement has dependences on the prior computation), then you retain a serial execution.

Multi-Level Parallel Task Execution

Refactoring to identify opportunities for relaxed sequential execution is typically implemented via fork-join parallelism. You fork when you want to relax the constraints of serial execution, and you join (using a barrier) when you want to enforce the constraints of serial execution. The barrier at the join point allows any dependence from before the barrier to after the barrier to be satisfied.

In simple fork-join parallelism the application is either executing serially or it is executing in parallel. From Amdahl's law you know that potential speedup is limited by the percentage of time the program is executing serially. Therefore, any time you transition from parallel to serial you are losing performance.

To overcome this problem, you can think hierarchically. The trick is to find a way to replicate the serializing algorithms at the inner level, and create multiple tasks at an outer level that each works independently. The QuickSort algorithm is a good example (**Figure 5**).

The algorithm has a serial phase (i.e., sorting small arrays, choosing the pivot, and partitioning the array) and a parallel phase (i.e., sorting the two halves of the now partitioned array). The hierarchy you can exploit is the recursive call to the QuickSort function.

Thinking hierarchically (sometimes recursively) expands your ability to specify independent work that can be exploited for parallel execution. If you only create parallelism from the two top-most independent QuickSort calls, then your speedup will be implicitly limited to, at most, a factor of 2x. Recursively subdividing these tasks into smaller tasks enables additional parallelism.

Another reason why hierarchical parallelism design is helpful is in increasing the number of tasks that can be launched. Problems best suited for parallelism have large collections of objects, each of which needs to be transformed independently by the application of a function. If you can transform your algorithm into this form you will achieve the best results.

Often, the objects are not this independent; you may only have a small collection of "top-level" objects that are independent. If you apply parallelism only to this "top-level" collection of objects you may get a large enough grain size to allow effective parallel execution, but your scalability will be limited to the number of "top-level" objects you have.

If your algorithm runs long enough to warrant the use of parallelism, you may find another level of nested algorithms that also does an independent computation over a different small set of independent data items. Applying parallelism to both levels increases the scalability of your parallel algorithm.

```
for (int i = 0; i < N; ++i)
      A[i] = B[i] + C[i]
```

Figure 3

```
parallel {
      for (int i = 0; i < N; ++i)
          task { A[i] = B[i] + C[i] }
}
```

Figure 4

```
void QuickSort( Value A[], int L, int H ) {
      if (H-L < TooSmallLimit) {
            SerialSort(A, L, H);
            return;
      }
      Value Pivot = A[ L+(H-L)/2 ];
      int L1 = L; int H1 = H;
      while (L1 < H1) {
            if (A[L1] < Pivot)
                  ++L1;
            else if (A[H1] >= Pivot)
                  --H1;
            else
                  Swap(A[L1], A[H1]);
      }
      parallel {
            task { QuickSort(A, L, L1-1); }
            task { QuickSort(A, L1, H); }
      }
}
```

Figure 5

Choosing Tasks

You can imagine that in the serial execution of the program tasks could exist at multiple levels. At one extreme, you could assume that the entire application is one task. At the other extreme, you could assume that every statement or expression is a task. Both extremes are usually incorrect when trying to pick where to add tasks to your application.

The potential concurrency of your application is bounded by the number of tasks you create. If you only create two tasks (your main program is also logically a task), your ideal parallel speed-up is only 2x faster than serial (ignoring serial cache or memory bandwidth effects), regardless of the number of processors you have available. Various forms of overhead will also reduce the potential benefit of creating parallel tasks, so you need to identify many more tasks than the available processors in order to achieve the best performance.

Having many tasks available also helps in another way. If you have two tasks, and one task is very short (e.g., one second), and the other task is very long (e.g., 59 seconds), then running these two tasks serially will take one minute. How fast will they run when executed in parallel? You might hope that you would achieve a 2x speedup and finish both tasks in 30 seconds.

Unfortunately, the answer is that you will finish both tasks in 59 seconds. The minimum time to execute both tasks is the maximum time it would take to execute either task. Since the duration of the second task is 59 seconds, you must wait at least this long. If possible, it would be better to break up the second task much finer into a larger collection of tasks. For example, if instead of having one task that takes 59 seconds, you could create 59 tasks that each take one second, you could potentially achieve a 2x speed-up and finish all 60 tasks in 30 seconds using two processors.

You can see that a larger number of smaller tasks are necessary to have enough potential concurrency to achieve the performance improvement you desire. Two effects serve to balance this and prohibit tasks that are too small.

First, scheduling a task onto a thread is inexpensive, but does have a cost. Scheduling 10 tasks, each with a single action, onto a thread will be slightly more expensive than combining all 10 of those actions into a single task and only scheduling that one task onto a thread. Most parallel frameworks have the capability to combine tasks to create larger chunks of work to be scheduled onto a thread for execution. This is a simple mechanical process. However, the opposite—splitting a monolithic large task into many smaller tasks—is very difficult for the parallel framework, but may be fairly easy for you to specify.

Second, a good task encapsulates a consistent computation on an object or set of objects. If you examine all of the variables accessed by a task you will find that they fall into three cases.

These cases are:

1. The variable is already private to a task or read-only by all tasks.
2. The variable can be made private to a task.
3. The variables communicate values between tasks.

In the second half of this article, we examine these cases by showing how the way you use variables in your application affects the choices you will make as you understand how to transform your application to exploit available latent parallelism. □

Understanding the

Features of Intel® Parallel Inspector by Example

By Bradley J. Werth

Intel® Parallel Inspector eases the correctness burden on programmers. Explore how it helps maintain memory and thread integrity.

Intel® Parallel Inspector is a combination of tools that performs two important functions: verifying memory integrity and verifying thread correctness. Memory integrity has been a key challenge for software development throughout the history of computing. When some high-profile program gets hacked through a buffer overrun, that is a breach of memory integrity. Even without malicious intent, it is all too easy for a programmer to make a mistake using the complex memory syntax of a language like C. Likewise, multithreaded semantics provide plenty of leeway for programming mistakes. Intel Parallel Inspector aims to ease the correctness burden on programmers by providing some assurance that memory and thread integrity are maintained.

Accompanying this article is a sample C++ project that contains examples of all the memory and threading problems that Intel Parallel Inspector can detect, with clear labels of each problem. This article describes how Intel Parallel Inspector can be configured to catch those problems, which will help you find and fix similar problems in your own projects.

The data for graphs in this article were measured on a Windows 7* PC with an Intel® Core™ i7 Extreme Edition processor using the referenced example project. The time dilation or slowdown caused by Intel Parallel Inspector inspection analysis will vary based on the platform, system configuration, and your project of analysis.

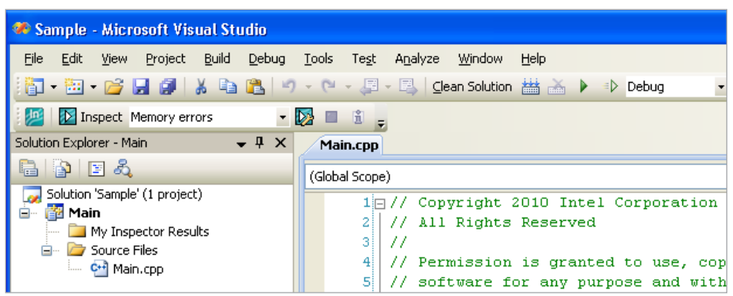


Figure 1: Intel Parallel Inspector appears as a toolbar in the Microsoft Visual Studio 2008* IDE.

Getting Started with Intel Parallel Inspector

Intel Parallel Inspector is installed into the Microsoft Visual Studio 2008* IDE and appears as a toolbar. **Figure 1** shows how Intel Parallel Inspector appears after a typical install.

The typical use of Intel Parallel Inspector is to build an application with debugging information included, and then run Intel Parallel Inspector memory analysis or threading analysis by choosing the type of analysis from the combo box and clicking the Inspect button.

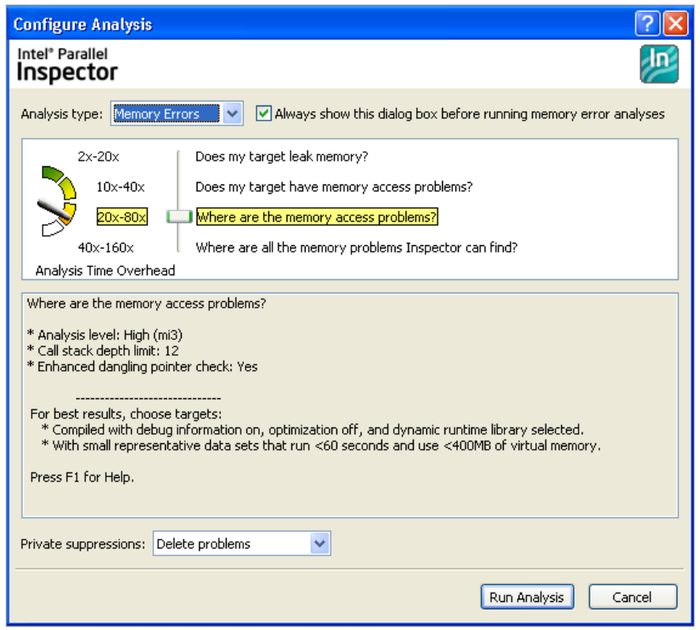


Figure 2: Intel Parallel Inspector allows configuration options for both memory and threading analysis.

Intel Parallel Inspector is a dynamic analysis tool, which means that it doesn't look at the source code but at the running program itself. This is both good and bad; good because many problems cannot be caught with just static analysis, but bad because this dynamic analysis can significantly slow down the execution of the program. Understanding this tradeoff, Intel Parallel Inspector provides "levels" of both memory and threading analysis. Levels range from 1 to 4 in order of increasing accuracy and decreasing performance. **Figure 2** shows the interface for specifying the level of memory analysis.

When the user clicks the Run Analysis button from the configuration screen, Intel Parallel Inspector will begin analysis and run the program to completion, or until the Stop button is pressed on the toolbar. Some problems will not be detected properly if the Stop button is pressed. When analysis is complete, Intel Parallel Inspector displays the analysis results in a tab in the Microsoft Visual Studio IDE. **Figure 3** shows the results of running Level 3 memory analysis on the sample project.

Double-clicking on one of the errors will display the source of the problem as well as additional detail to help diagnose the problem.

THE PARALLEL UNIVERSE

Figure 3: Analysis results appear in a tab in the IDE.

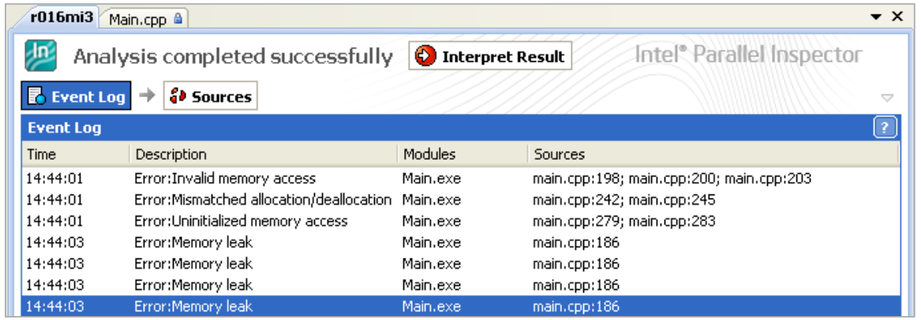
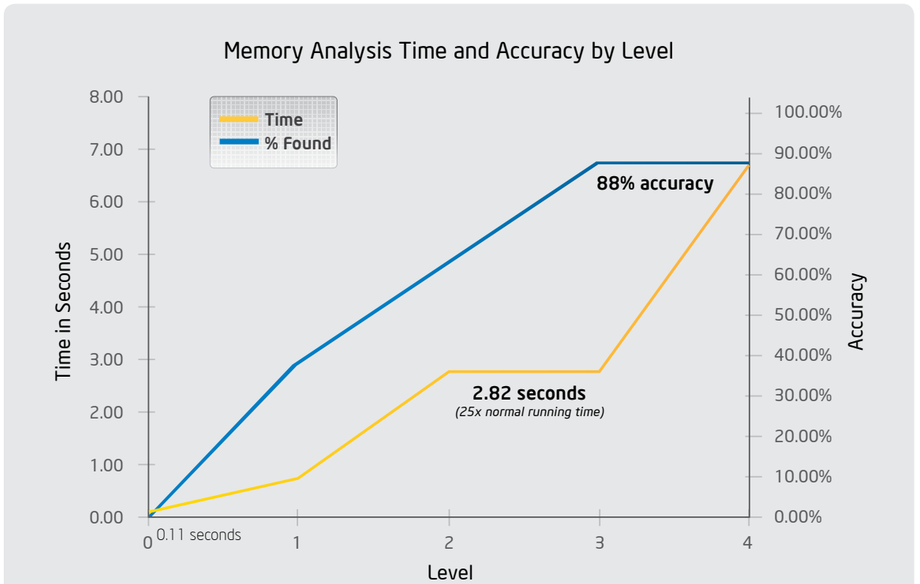


Figure 4: Memory analysis accuracy improves with analysis level—to a point.



Memory Analysis: Features, Accuracy, and Performance

The sample project contains 11 memory errors findable by Intel Parallel Inspector at the highest analysis level. Three of these errors are fundamentally unsafe in that they corrupt the heap or write to arbitrary memory. For this reason, those errors are only included as an optional preprocessor definition that is disabled by default. At level 1, Intel Parallel Inspector detects all of the memory leaks in the code, although deep memory leaks may not get the full call stack recorded. At level 2, read/write accesses to invalid memory are detected. At levels 3 and 4, the call stack depth for issues is progressively increased.

The following "safe" memory problems are present in the sample project:

- 4 memory leaks at different call stack depths
- 2 invalid memory reads
- 1 uninitialized memory access
- 1 mismatched allocation/deallocation

Figure 5: Threading analysis accuracy improves with analysis level—to a point.

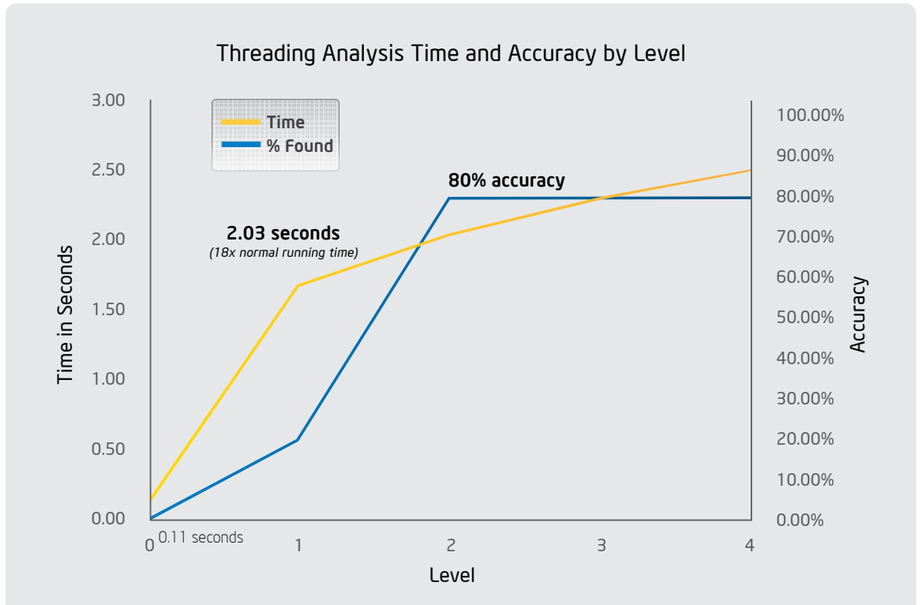


Figure 4 shows the tradeoff between accuracy and runtime for memory analysis of the sample program, when run on a Windows 7 PC with an Intel Core i7 Extreme Edition processor. Analysis levels 1 through 4 are shown alongside the percentage of problems found at each level. Analysis level 0 represents the case where Intel Parallel Inspector is not run on the code at all (i.e., the raw performance of the code).

The data indicates that level 3 memory analysis has an excellent accuracy rate for an acceptable increase in runtime. For memory analysis, 88 percent of the problems in this project are found with the runtime expanding to 25x of normal. Level 4 analysis provides additional information (e.g., deeper call stacks and thread stack analysis), but does not find additional memory problems in this project. Your project may have a more complex memory access pattern that necessitates level 4 memory analysis, but for most projects level 3 is sufficient and is certainly the right level for regular testing.

Threading Analysis: Features, Accuracy, and Performance

The sample project contains five threading errors findable by Intel Parallel Inspector at the highest analysis level. Unlike the memory analysis, all of the threading errors are safe in the sense that they do not corrupt memory. At level 1, Intel Parallel Inspector only detects potential deadlocks. At level 2, data races are also detected. Levels 3 and 4 increase the call stack depth of the reported data races.

The sample project contains the following threading problems:

- 3 heap data races
- 1 stack data race
- 1 deadlock

Figure 5 shows the tradeoff between accuracy and runtime for threading analysis of the sample program, when run on a Windows 7 PC with an Intel Core i7 Extreme Edition processor. Analysis levels 1 through 4 are shown alongside the percentage of problems found at each level. Analysis level 0 represents the case in which Intel Parallel Inspector is not run on the code at all (i.e., the raw performance of the code).

The data for threading analysis tells a similar story as the memory analysis. Level 2 analysis catches 80 percent of the problems with a runtime expanding to 18x of normal. Level 4 analysis will check for data races among stack variables. Stack variables are not meant to be shared, so data races are not common. A lower level analysis, such as level 2 or 3, are more commonly used for regular testing.

Further Reading and Resources

The help files installed with Intel Parallel Inspector (accessible in Microsoft Visual Studio* from the Help Menu as Intel® Parallel Studio—Parallel Studio Help—Inspector Help) give additional examples of detectable errors in the section "Problem Type Reference." □

THE PARALLEL UNIVERSE

The most important consideration is how to manage the calls between the managed .NET application and the unmanaged Intel® IPP Library.

Thread Your C# Code with Intel® Integrated Performance Primitives

By Naveen Gv

Investigate the basics of calling Intel® Integrated Performance Primitives (Intel® IPP) from .NET Framework languages such as C#.

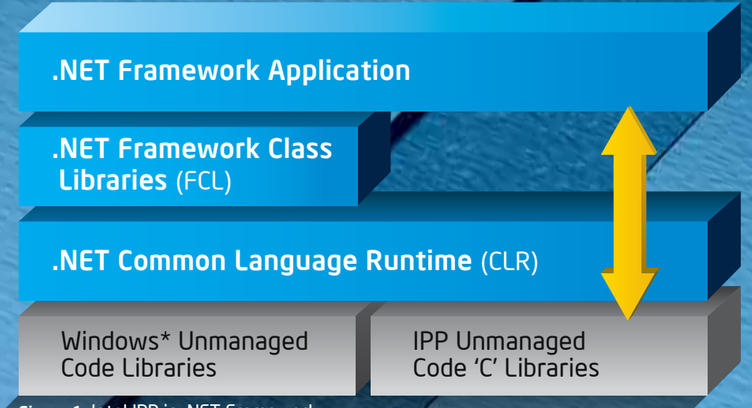


Figure 1: Intel IPP in .NET Framework

This article is intended to educate Intel® Integrated Performance Primitives (Intel® IPP) users on the basics of calling Intel IPP from .NET Framework languages such as C#. The most important consideration is how to manage the calls between the managed .NET application and the unmanaged Intel® IPP Library. **Figure 1** provides a high-level perspective.

Intel® IPP is an unmanaged code library written in native programming languages and compiled to machine code that can run on a target computer directly.

In this article, we consider the Platform Invocation services (P/Invoke) .NET interoperability mechanism specifically for calling the Intel IPP from C#. We describe the basic concepts such as passing callback functions and arguments of the different data types from managed to unmanaged code and the pinning of array arguments.

Microsoft* .NET Framework Overview

.NET Framework Terminology

.NET Framework: The Microsoft* .NET Framework is a managed runtime environment for developing the applications that target the common language runtime (CLR) layer. This layer consists of the runtime execution services needed to develop various types of software applications, such as ASP .NET, Windows* forms, XML Web services, distributed applications, and others. Compilers targeting the CLR must conform to the common language specification (CLS) and common type system (CTS), which are sets of language features and types common among all the languages. These specifications enable type safety and cross-language interoperability. It means that an object written in one programming language can be invoked from another object written in another language targeting the runtime.

C#: C# is a programming language designed as the main language of the Microsoft .NET Framework. It compiles to a common intermediate language (CIL) like all the other languages compliant with the .NET Framework. CIL provides a code that runs under control of the CLR. This is managed code. All codes that run outside the CLR are referred to as unmanaged codes. CIL is an element of an assembly.

Assembly: An assembly is a collection of types and resources that are built to work together and form a logical unit of functionality. They are the building blocks of the .NET Framework applications, are stored in the portable executable (PE) files, and can be a DLL or EXE. Assemblies also contain metadata, the information used by the CLR to guarantee security, type safety, and memory safety for code execution.

Garbage collector: The .NET Framework's garbage collector (GC) service manages the allocation and release of memory for the managed objects in the application. The GC checks for objects in the managed heap that are no longer used by the application and performs the operations necessary to reclaim their memory. The data under control of the GC is managed data.

Unsafe code: C# code that uses pointers is called unsafe code. The keyword "unsafe" is a required modifier for the callable members such as properties, methods, constructors, classes, or any block of code. Unsafe code is a C# feature for performing memory manipulation using pointers. Use the keyword fixed (pin the object) to avoid movement of the managed object by the GC. Note that unsafe code must be compiled with the /unsafe compiler option.

Figure 2

```
DllImport("custom.dll")]
static extern double
foo(double a, double b);
```

.NET Framework Interoperability Mechanisms

The CLR supports the Platform Invocation Service (P/Invoke) that allows mapping a declaration of a managed method to unmanaged methods. The resulting declaration describes the stack frame but assumes the external method body from a native DLL.

P/Invoke Service

P/Invoke enables managed code to call C-style unmanaged functions in native DLLs. P/Invoke can be used in any .NET Framework-compliant language. It is important to be familiar with the attributes DllImport, MarshalAs, StructLayout, and their enumerations to use P/Invoke effectively.

When a P/Invoke call is initiated to call an unmanaged function in the native DLL, the P/Invoke service will perform the following steps:

1. Locate the DLL specified by the DllImport attribute by searching either in the working directory or in the directories and sub-directories specified in the PATH variable, and then load the DLL into the memory.
2. Find the function declared as static extern in the DLL loaded to memory.
3. Push the arguments on the stack by performing marshalling and, if required, using the attributes MarshalAs and StructLayout.
4. Disable pre-emptive garbage collection.
5. Transfer the control to the unmanaged function.

Declare Static Extern Method with the DllImport Attribute

An unmanaged method must be declared as static extern with the DllImport attribute. This attribute defines the name of a native shared library (native DLL) where the unmanaged function is located. The attribute DllImport and function specifiers static extern specify the method that is used by the .NET Framework to create a function and to marshal data.

The DllImport attribute has parameters to specify correspondence rules between managed and native methods, such as CharSet (Unicode or Ansi), ExactSpelling (true or false), CallingConvention (cdecl or StdCall), EntryPoint.

In the simplest case, the managed code can directly call a method foo() as illustrated in **Figure 2**.

Marshalling for the Parameters and Return Values

The .NET Framework provides an interoperability marshaller to convert data between managed and unmanaged environments. When managed code calls a native method, parameters are passed on the call stack. These parameters represent data in both the CLR and native code. They have the managed type and the native type.

Some data types have identical data representations in both managed and unmanaged code. They are called isomorphic, or blittable, data types. They do not need special handling or conversion when passed between managed and unmanaged code. Basic data types of this kind are: float/double, integer, and one-dimensional arrays of isomorphic types. These are common types in Intel IPP.

However, some types have different representations in managed and unmanaged code. These types are classified as non-isomorphic, or non-blittable, data types and require conversion, or marshalling. **Table 1** presents some non-isomorphic types commonly used in the .NET Framework.

In some cases, the default marshalling can be used. But not all parameters or return values can be marshalled with the default mechanism. In such cases, the default marshalling can be overridden with the appropriate marshalling information. Marshalling includes not only conversion of the data type, but other options such as the description of the data layout and direction of the parameter passing. There are some attributes for these purposes: MarshalAs, StructLayout, FieldOffset, InAttribute, and OutAttribute.

MarshalAsAttribute and Marshal class in the System.Runtime.InteropServices namespace can be used for marshalling non-isomorphic data between managed and unmanaged code.

MANAGED	UNMANAGED
Boolean	BOOL, Win32 BOOL, or Variant
Char	CHAR, or Win32 WCHAR
Object	Variant, or Interface
String	LPStr, LPWStr, or BStr
Array	SafeArray

Table 1: Mapping of several data types, unmanaged-managed environment

Figure 3: Pinning Arrays and Other Objects: Declarations are similar for both methods

```
// fixed statement in unsafe code
[DllImport("custom.dll")]
unsafe static extern float
foo(float *x);

// automatic pinning
[DllImport("custom.dll")]
static extern float
foo(float[] x);
```

Pinning Arrays and Other Objects

All objects of the .NET Framework are managed by the GC. The GC can relocate an object in memory asynchronously. If managed code passes to native code a reference to a managed object, this object must be prevented from being moved in the memory. Its current location must be locked (or pinned) for the life of the native call.

Pinning can be performed manually or automatically. Manual, or explicit, pinning can be performed by creating GCHandle: GCHandle pinnedObj = GCHandle.Alloc(anObj, GCHandleType.Pinned).

Pinning can also be performed using the fixed statement in unsafe code. An /unsafe compilation option is required in this case.

The other method is automatic pinning. When the runtime marshaller meets managed code that passes an object to the native code, it automatically locks the referenced objects in memory for the duration of the call. Declarations are similar for both methods. See **Figure 3**.

Automatic pinning enables calling native methods in a usual manner and does not require the /unsafe option during compiling.

Note that aggressive pinning of short-lived objects is not good practice because the GC cannot move a pinned object. This can cause the heap to become fragmented, which reduces the available memory.

Callback Function

A callback function is a managed or unmanaged function that is passed to an unmanaged DLL function as a parameter. To register a managed callback that an unmanaged function uses, declare a delegate with the same argument list and pass an instance of it through P/Invoke. **Figure 4:** On the unmanaged side, it appears as a function pointer.

The CLR automatically pins the delegate for the duration of the native call. Moreover, there is no need to create a pinning handle for the delegate for the asynchronous call: In this case, the unmanaged function pointer actually refers to a native code stub dynamically generated to perform the transition and marshalling. This stub exists in fixed memory outside of the GC heap. The lifetime of the native code stub is directly related to the lifetime of the delegate.

The delegate instance, which is passed to unmanaged code, employs the StdCall calling convention. An attribute to specify the Cdecl calling convention is available only in the .Net Framework v2.0+.

BLOG highlights

Structured Parallel Programming

BY MICHAEL MCCOOL

One way of looking at parallel patterns (sometimes called algorithmic skeletons) is through an analogy with "structured programming." The premise of structured programming is that a small number of control flow and data management patterns can be composed to implement the necessary control flow and data access logic in most serial programs. There is some evidence (see, for example, work by Skillicorn, Campbell, Cole, and the Berkeley Dwarfs) that a relatively small number of patterns can also express the necessary task and data organization in a large fraction of parallel programs.

Back in the '70s there was a heated argument about structured vs. unstructured control flow. Basically, you can obviously do anything you want with a conditional goto, which is usually the only control flow construct made available in machine language. From the point of view of completeness, this is all a programming language really needs to support. However, many computer scientists noted that there were certain maintainability advantages to restricting control flow to the composition of a small number of patterns supporting iteration (do/while, repeat/until, for) and selection (if/then/else, switch) in high-level languages.

```
public delegate void MyCallback();
[DllImport("custom.dll")]
public static extern void
MyFunction(MyCallback callback);
```

Figure 4

```
          // ipps.cs
          namespace ipp {
              public class sp {
              …
              };
          };
// ippi.cs
          namespace ipp {
  public class ip {
  …
  };
};
```

Figure 5

```
public enum IppRoundMode {
ippRndZero = 0,
ippRndNear = 1,
ippRndFinancial = 2,
};
```

Figure 6

Intel IPP Overview

Intel IPP is a low-level software library. It provides a set of basic functions highly optimized for the IA-32, IA-64, and Intel® 64 architectures. The use of the library significantly speeds up a wide variety of software in different application areas: signal and image processing, speech (G.728, GSM-AMR, Echo Canceller), audio (MP3, AAC), and video (MPEG-2, MPEG-4, H.264, VC1) coding, image coding, data compression (BWT, MFT, RLE, LZSS, LZ77), cryptography (SHA, AES, RSA certified by NIST), text processing, and computer vision. The Intel IPP software runs on different operating systems: Windows* OS, Linux* OS and Mac OS* X.

Intel IPP is a C-style API library. However, due to the StdCall calling convention, the primitives can be used in applications written in many other languages. For example, they work in applications written in Fortran, Java*, Visual Basic, C++, and C#. The Intel IPP functions can be used in the Microsoft .NET Framework managed environment to speed up the performance of applications on Intel® processors and compatible platforms.

```
[StructLayout(LayoutKind.
Sequential,CharSet=CharSet.Ansi)]
public struct Ipp64fc {
      public double re;
      public double im;
      public Ipp64fc( double re, double im ) {
          this.re = re;
          this.im = im;
      };
};
```

Figure 7

```
using System;
using System.Runtime.InteropServices;
namespace ipp {
  public enum IppStatus {
      …
      ippStsNoErr = 0,
      …
      };
  [StructLayout(LayoutKind.Sequential,CharSet=CharSet.Ansi)]
public struct IppiSize {
      public int width;
      public int height;
      public IppiSize( int width, int height ) {
          this.width = width;
          this.height = height;
      };
  };
  unsafe public class ip {
      [DllImport("ippi-6.1.dll")] public static extern
      IppStatus ippiAndC_8u_C1R( byte* pSrc, int srcStep, byte value,
                                 byte* pDst, int dstStep, IppiSize roiSize );
  };
};
```

Figure 8

C# interface to Intel IPP Namespace and Structures

The Intel IPP software, in addition to the libraries, provides special wrapper classes for the functions of several Intel IPP domains: signal and image processing, color conversion, cryptography, string processing, data compression, JPEG coding, and math and vector math. The wrappers allow Intel IPP users to call Intel IPP functions in C# applications. These classes, in the case of signal sp and image processing ip functions, are declared as in **Figure 5**.

Enumerated data types that are used in the Intel IPP library must be declared in the wrapper classes with a keyword "enum". For example, the enumerator IppRoundMode is declared in ippdefs.cs as illustrated in **Figure 6**.

Many Intel IPP functions use structures as parameters. To pass structures from a managed environment to an unmanaged environment, the managed class struct must comply with the corresponding library in the unmanaged code. The attribute StructLayout is used with the value LayoutKind.Sequential. **Figure 7** shows how to use the structure Ipp64fc, which is a type of double complex number.

Almost all Intel IPP functions have pointers as parameters. The functions with pointers must be declared in the unsafe context. For example, the function ippiAndC_8u_C1R is declared as in **Figure 8**.

This code must be compiled with the /unsafe option.

Note that working with pointers in the C# application requires using the operator fixed. The fixed statement sets a pointer to a managed variable, and this variable is used during execution of the statement. Without the fixed statement, pointers to managed variables may be relocated unpredictably by the GC.

For the class Bitmap, the methods LockBits and UnlockBits must be used.

Almost all Intel IPP functions have pointers as parameters.

```
namespace ExampleIP
{
      using System;
      using System.Windows.Forms;
      using System.Drawing;
      using System.Drawing.Imaging;
      using ipp;

      public class tip : System.Windows.Forms.Form {
          private System.Drawing.Bitmap bmpsrc, bmpdst;

private BitmapData getBmpData( Bitmap bmp ) {
              return bmp.LockBits( new Rectangle(0,0,bmp.Width,bmp.Height),
                  ImageLockMode.ReadWrite, PixelFormat.Format24bppRgb );
          };

          unsafe private void FilterBoxFunction() {
              …
              BitmapData bmpsrcdata = getBmpData( bmpsrc );
              BitmapData bmpdstdata = getBmpData( bmpdst );
              IppiSize roi = new IppiSize( bmpsrc.Width*3/4, bmpsrc.Height*3/4 );
              const int ksize = 5, half = ksize/2;
              // the three-channels images
              byte* pSrc = (byte*)bmpsrcdata.Scan0+(bmpsrcdata.Stride+3)*half,
                     pDst = (byte*)bmpdstdata.Scan0+(bmpdstdata.Stride+3)*half;
              IppStatus st = ip.ippiFilterBox_8u_C3R(pSrc,bmpsrcdata.Stride,
                                                     pDst,bmpdstdata.Stride,
                                                     roi,
                                                     new IppiSize(ksize,ksize),
                                                     new IppiPoint(half,half)});
              …
          };
      };
  };
```

Figure 9

```
namespace ExampleIP
{
  using System;
  using System.Windows.Forms;
  using System.Drawing;
  using System.Drawing.Imaging;
  using System.Reflection;
  using System.Collections;
  using ipp;

  public class tip : System.Windows.Forms.Form {
      private Assembly assembly;
      private object ippi;
      private Type ippiType;
      private Hashtable hash;
      …
      public tip(Bitmap bmp) {
          assembly = Assembly.LoadFrom("ippi_cs.dll");
          ippi = assembly.CreateInstance("ipp.ip");
          ippiType = ippi.GetType();

  …
      };
      void CreateMenu() {
          hash = new Hashtable();

MenuItem miBlur =
              new MenuItem("Blur", new EventHandler(MenuFilteringOnClick));
          hash.Add(miBlur, "ippiFilterBox_8u_C3R");
          MenuItem miMin =
              new MenuItem("Min Filter",
                           new EventHandler(MenuFilteringOnClick));
          hash.Add(miMin, "ippiFilterMin_8u_C3R");
          …
      };

      private void MenuFilteringOnClick(object sender,System.EventArgs e) {
          FilteringFunction((string)hash[sender]);
      };

      unsafe private void FilteringFunction(string func) {…
          MethodInfo method = ippiType.GetMethod(func);
          const int ksize = 5, half = ksize/2;
          byte* pSrc = (byte*)bmpsrcdata.Scan0+(bmpsrcdata.Stride+3)*half,
                pDst = (byte*)bmpdstdata.Scan0+(bmpdstdata.Stride+3)*half;
          IppStatus st = (ipp.IppStatus)method.Invoke(null, new object[]
              {(IntPtr)pSrc,bmpsrcdata.Stride,(IntPtr)pDst,bmpdstdata.Stride,
                  roi, new IppiSize(ksize,ksize), new IppiPoint(half,half)});
      …};
    };
  };
```

Figure 10

INTEL IPP FUNCTION	PERFORMANCE OVERHEAD
ippiFilterMedian_8u_C3R	14%
ippiRotateCenter_8u_C3R	14%
ippiWarpAffine_8u_C3R	39%
ippiMirror_8u_C3R	47%

Table 2: Cost of C# call—the lighter an operation, the bigger the overhead.

When calling functions with the same set of parameters, you can use the dynamic method of function search and execution.

Intel IPP Components—Image Processing Sample

Intel IPP Samples include C# interface and a demo application illustrating how C# developers can build applications with Intel IPP calls. The demo application performs filtering and morphological and geometric operations.

Additionally, the image compression functions are used in the demo to read and write JPEG files. The application uses the wrapper classes for the image processing (ippi.cs) and image compression (ippj.cs) domains. The application launches the P/Invoke mechanism for unmanaged code in ippi-6.1.dll and loads the dispatcher of the processor-specific libraries. This dispatcher loads the most efficient library for a given processor. For example, the library ippiv8-6.1.dll is loaded on a system with an Intel® Core™2 Duo processor, and the library ippiw7-6.1.dll is loaded on a system with an Intel® Pentium® 4 processor.

Image ROI Processing

Special attention must be paid when working with the functions that require border pixels (e.g., image filtering functions). The Intel IPP functions operate only on pixels that are part of the image. Therefore, a region of interest (ROI) is implied to be inside the image in such a way that all neighborhood pixels necessary for processing the ROI edges actually exist. For example, for filtering functions, the width of the border outside the ROI must not be less than half of the filter kernel size with the centered anchor cell.

When processing an image ROI, the developer has to perform two additional operations: shifting the pointer to the data and specifying the ROI size that is less than the image size. In **Figure 9** the sample code illustrates how to work with ROI (using the example of the Intel IPP function ippiFilterBox that performs image blurring).

Runtime Function Invocation

The code example in **Figure 9** shows how an Intel IPP function is launched via a direct call, and how static code is generated when the application is compiled. This method is rather simple and obvious.

When calling functions with the same set of parameters, you can use the dynamic method of function search and execution. This method is called reflection and can noticeably reduce the size of the executable code.

Figure 10 shows the demo application, the reflection method is used to launch filtering and morphological functions from the menu.

Intel IPP Performance from C#

Intel IPP functions are optimized for Intel® and compatible processors. This optimization can speed up performance of various applications. However, specific features of using these libraries in the .NET Framework can decrease performance of the application.

Two possible reasons for a decrease in performance:

- Managed C# code calls unmanaged code through P/Invoke. Every P/Invoke call requires from eight to 27 CPU clocks.
- When a function is called from a DLL for the first time, the corresponding DLL must be loaded into memory (e.g., using LoadLibrary() on Windows* OSs), which takes 1000 to 2000 CPU clocks. More CPU clocks are needed to create the entry points for all functions that are exported by this DLL.

Table 2 shows the performance overhead numbers for Intel IPP image processing functions. Because Intel IPP functions are several times faster at corresponding C# implementations, it still makes sense to call Intel IPP despite the overhead of a C# call. To decrease the overhead effect, developers can create a component- or application-level interface in which one C# call leads to the execution of many IPP functions.

An example of this approach is the .NET interface for DMIP. Also, we can compare the performance of the C# implementation and C# call of Intel IPP functions. For example, the C# .NET library Mirror costs 5.7 CPU cycles per pixel; the Intel IPP-based C# call of Mirror function costs 1.7 cycles per pixel.

Conclusion

The managed-unmanaged code interoperability provided in the .NET Framework environment can be performed in different ways. The best way to call C functions residing in a native DLL is to use the P/Invoke service, which is available in any managed language. P/Invoke provides a powerful and flexible interoperability with inherited codes. The DllImport attribute declares the external entry point. Marshalling attributes allow describing various options for the data conversion and data layout in addition to the default data marshalling.

The use of SuppressUnmanagedCodeSecurityAttribute, InAttribute, and OutAttribute may significantly reduce the performance overhead. Automatic pinning of the objects passed prevents them from being garbage collected for the duration of the call. Manual pinning is also available if the pointer to the object is kept and used in native code after the call returns. A managed delegate type allows implementing callback functions.

References

To download an Intel IPP evaluation package, sample code, and other documents, visit http://software.intel.com/en-us/intel-ipp/. □

RESOURCES AND SITES OF INTEREST

The mission of Dr. Dobb's Go Parallel is to assist developers in their efforts toward "Translating Multicore Power into Application Performance." Robust and full of helpful information, the site is a valuable clearinghouse of multicore-related blogs, news, videos, feature stories, and other useful resources.

For its fall 2009 webinar series Intel invited Microsoft Visual Studio* C/C++ developers from a range of industry-leading companies to share the secrets behind their real-world successes using Intel® Parallel Studio. All webinars, including those from previous series, are available for immediate, on-demand download.

Check out a range of resources on a wide variety of software topics for a multitude of developer communities ranging from manageability to parallel programming to virtualization and visual computing. This content-rich collection includes Intel® Software Network TV, popular blogs, videos, tools, and downloads.

What if you could experiment with Intel's advanced research and technology implementations that are still under development? And then what if your feedback helped influence a future product? It's possible here. Test drive emerging tools, collaborate with peers, and share your thoughts via the What If blogs and support forums.

The Intel® Software Evaluation Center makes 30-day evaluation versions of Intel® Software Development Products available for free download. For High Performance Computing Products, you can get free support during the evaluation period by creating an Intel® Premier Support account after requesting the evaluation license, or via Intel® Software Network Forums. For evaluating Intel® Parallel Studio, you can access free support through Intel® Software Network Forums ONLY.

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