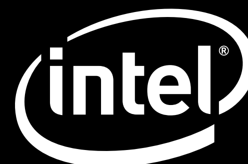


The State of Software Development Today: A Parallel View

June 2012

What is Parallel Programming?

When students study computer programming, the normal approach is to learn to program sequentially. We code our software such that the steps happen one after another. Only after one step is finished does the next begin. For example, we learn various algorithms for solving problems, such as sorting, and we code the individual steps of our sort algorithm sequentially, step by step.

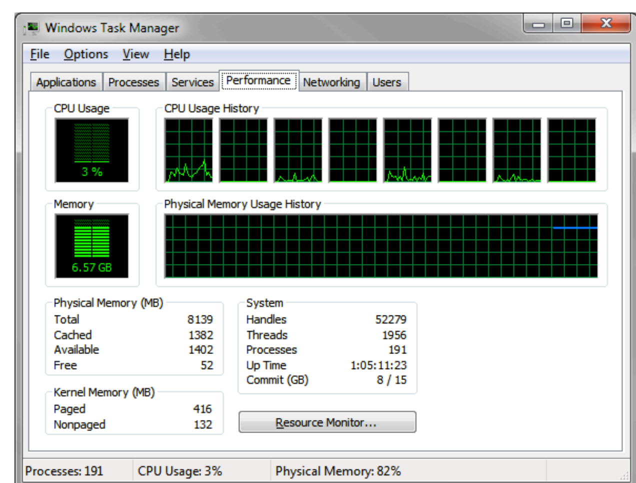
For many software applications, this approach seems to work fine. But years ago, researchers discovered that, given two computers – or three, or more – it was possible for a single program to divide up its tasks between the computers, in the same way a team of people might divide up the labor to complete a single project.

Of course, in order for a team to function the most effectively, their labor needs to be divided in such a way that one person doesn't have to stop and wait for another person to finish before beginning his or her task. In the same way, software needs to be carefully coded such that one process, or thread, can complete its task without necessarily having to wait for another task to finish. This results in the most optimal use of the separate computers.

Students learn to create software that models an actual entity. For example, a checking account program would model an actual checking account. It would include features such as account balance, the ability to deposit money, and the ability to withdraw money. Real-life entities such as the checking account are easy to model. But other entities are much more complex. For example, a weather system model would require knowing the individual temperature and pressure at hundreds, thousands, possibly millions of locations in the three dimensions of the earth's atmosphere. Determining what the weather does next requires large, sophisticated algorithms. If these algorithms are coded in a strictly sequential manner, it could take hours or days to complete just one step in the model – possibly even after the actual storm system has passed.

This is where parallel programming comes in. In the past, coding in parallel required computers with multiple processors, and these were found primarily in research labs. The cost was simply too prohibitive. So while computer science students might learn the basics of parallel programming, their education would be primarily hypothetical. Or, if they had access to a supercomputer, as soon as they graduated and found work in industry, they would likely go back to strictly sequential programming.

But all that changed with the advent of multi-core processors. Almost all desktop and most notebook computers sold today feature at least two cores; four cores are extremely common. Further, each core can usually run two simultaneous threads, doubling the effective number of cores. As such, for instance, the Task Manager program in Microsoft Windows running on an

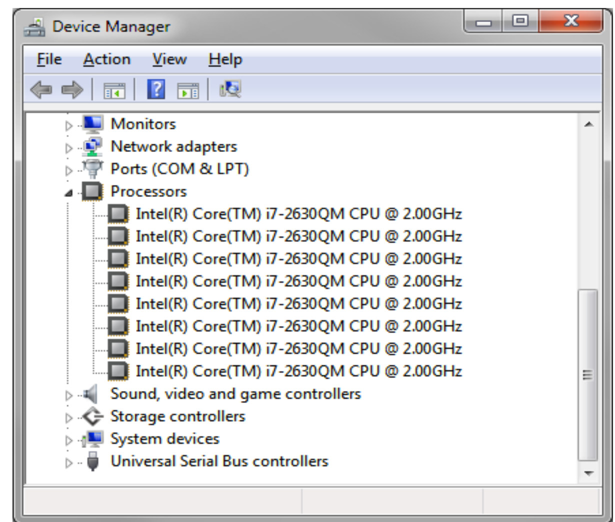


Task Manager shows eight separate CPU threads

Intel® i7 Quad Core processor-based machine will show what appears to be eight individual processors. The Device Manager will similarly show eight processors.

Why Use Parallel Programming?

With the right tools, the computer programmer can easily write code that is split up into multiple threads running simultaneously. And with a good understanding of parallel techniques, the programmer can divide up algorithms effectively. The resulting executable can even detect how many cores and threads area available at runtime. If the program runs on a single-core, single-thread machine, it will degrade gracefully down to a sequential algorithm. On the other hand, if the same program, with no changes, runs on a multiple-core computer with dozens or even hundreds of cores, the program will make use of as many cores as possible and finish the task as quickly as possible.



Device Manager shows eight separate processers, even though there's actually only a single Quad-Core i7 processor installed in this computer

And it doesn't stop there. Today's processors also feature registers that span many bytes, typically 16 or even 32 bytes. But integers and single-precision floating point numbers typically take up only four bytes. A program running on a computer with 16-byte-wide registers can simultaneously place four integers or four single-precision numbers in one of these single registers, and perform math simultaneously on all four numbers. The processors include support for such calculation with individual instructions capable of doing mathematical functions on all four numbers at once. With careful coding, loops can place four numbers into the registers at once and do four calculations, effectively finishing a task in a quarter of the time. This is called vectorization, and today's processors support both parallel algorithms and vectorization. These registers, called Single Instruction Multiple Data (SIMD) registers, have been available since the original MMX technology and Streaming SIMD Extensions (SSE) in the early Pentium processors, and have evolved to today's SSE4.2 and Advanced Vector Extensions (AVE) within current processors.

One might ask, "Why use parallel programming?" Perhaps a better question is, "Why not?" If you're creating software, why not get the most out of the processor? Why not take advantage of as much processing as is available? Why not finish the calculations for the entire storm system before the storm even arrives? Why not provide the scientist with maximum efficiency in calculations? Why not use those processors to calculate all the points in a 3D game so that the game moves as smoothly as scenes in a movie?

Components of Parallel Programming

Traditional, sequential programming usually involves creating both data structures and algorithms. The algorithms operate on the data structures, and the structures and algorithms can be combined in various ways (for example, object-oriented programming packages up the algorithms together within a data structure).

Programming in parallel, then, requires making your algorithms parallel-friendly, as well as your data structures.

For example, suppose an algorithm needs to calculate a sum. If the algorithm is split up into four separate threads, and there's only a single variable holding the running total, the four threads might clash, resulting in an incorrect sum. Two threads might simultaneously grab the current total; one thread adds on its current sum, and saves it. But immediately after that, the second thread would add its sum to the first total as well, and then save it right after the first. The end result is the wrong total. The first thread's sum was ignored.

One solution might be to let each thread work on the total variable individually. But that somewhat defeats the purpose of breaking the summation up into a parallel algorithm. Instead, the correct way is to make use of what's called a *reducer*, whereby each thread operates on a single sum, and then at the end the individual sums are combined. With the help of a good threading library, the programmer doesn't even need to write the reduction code. Instead, the programmer can declare the total variable as a reducer variable, and write the loop that performs the sum. The library will not only split the loop into parallel, but will also provide local copies of the total variable for the individual threads, and then combine them together. The end result is code that is extremely simple: one total variable, one loop. But under the hood, the loops run in parallel.

Industry Needs

Virtually every industry can benefit from an increase in computational power. Some of the most obvious are scientific industries such as meteorology, as previously noted. But there are many more that are worth mentioning.

For example, the motion picture industry is already making heavy use of parallelism with three-dimensional (3D) graphics. These movies are created using 3D software tools that use parallel programming, and state directly in their software requirements the need for advanced SIMD registers for vectorization. Without such parallel software, the rendering of 3D movies would take much longer to complete, and filmmakers would still be forced to create graphics that were more simplistic and less realistic.

Healthcare is another industry that already is benefiting from parallel programming. Again, 3D imagery comes into play here, but other areas of healthcare software also benefit. Software could process large data sets from an MRI system, for example, and find a diagnosis in a much shorter time than otherwise would be possible.

Industrial automation and robotics benefit as well. The enormous robots used in automotive factories are some of the most advanced examples of robotics. These robots need to be able to make rapid calculations on multiple data points simultaneously so they can maximize the work, and move in three-dimensional space to move accordingly, often with unforeseen circumstances. Without parallel programming, the robots would not be able to function as effectively.

Perhaps the most obvious example is the gaming industry. Sophisticated 3D scenes can be brought to motion only by using processors to their fullest potential, which includes the use of parallel programming and especially vectorization. 3D game programming requires solving simultaneous algebraic equations, which is an ideal use of vectorization.

The list of industries benefitted by parallel programming is virtually endless. The military industry needs to make rapid calculations with weapons, even making modifications to calculations at the last moment. The space industry needs software that can respond quickly as a space vehicle is launched and as it moves through space.

Generally speaking, where computers and calculations are needed, an industry can benefit (and already is benefiting) from parallel programming.

Difficulties in Older Methods

Although parallel programming is powerful, writing the code can be cumbersome. Trying to split off the loops into multiple threads requires either making calls into the operating system using its own threading library or, for maximum effectiveness, embedding assembly code directly in the higher-level code, to make use of features such as vectorization and the spawning of the code onto multiple cores. Coding becomes even more complex if the single program is to detect how many cores and threads are available, and try to make use of all of them.

As with software engineering in general, certain aspects should be easily available to the programmer through tools and libraries, so that the programmer doesn't have to concern him- or herself with the nuts and bolts of some lower-level tasks. For example, most high-level languages include libraries for container and collections so that the programmer doesn't have to recode a linked list every time one is needed.

By the same token, today's programming tools should offer as many features to make parallel programming happen at best automatically, or at least with minimal effort on behalf of the programmer. That's where Intel® Parallel Studio XE comes in.

How Intel Can Help

Intel Parallel Studio XE provides the right tools for parallel programming. Intel Parallel Studio XE runs directly inside Microsoft® Visual Studio® 2010, which means you can continue using the tools you already know. Parallel Studio XE includes multiple technologies such as Intel® Cilk™ Plus, and Intel® Threading Building Blocks.

Introducing Intel® Cilk Plus

Intel® Cilk Plus is an extension to the C/C++ languages that allows for easy conversion from serial to parallel loops and functions. It is based on the original Cilk technology created by Charles E. Leiserson, a pioneer in

parallel computing at the Massachusetts Institute of Technology. The original Cilk technology included a language that was based in C, and included additional keywords to support parallel programming. From there, designers developed Cilk++, a commercial form of Cilk. Cilk Arts, the makers of Cilk+, was acquired by Intel, and the technology was given a new name, Intel Cilk Plus.

For the programmer, Intel Cilk Plus is easy to use yet very powerful. There are only three new keywords, and the rest of the language is the same ANSI-standard C and C++ programmers already know.

The first Intel Cilk Plus keyword is _Cilk_for. This keyword uses the same syntax as the familiar for keyword in C and C++. By replacing a for keyword with a _Cilk_for keyword, a loop can be recoded to run in parallel. Once executing, the runtime library will attempt to launch as many simultaneous copies of the loop as possible based on the number of cores and threads available.

The next keyword is _Cilk_spawn. Place this keyword immediately before a function call, and the runtime will attempt to launch the function as a separate thread, again based on the number of available cores and threads.

The final keyword is _Cilk_sync. This keyword is used as a single statement, and it allows completed threads to pause and wait until all threads are finished. This allows for code synchronization so that all threads can complete before additional code is run.

Using these three keywords, a programmer can easily write parallel code without having to drop down to assembly language, and without having to make operating systems calls. Here’s an example for-loop written in traditional C++:

```
for (int i=0; i<1000; i++) {
      res[i] = ar[i] * 2;
}
```

Now here is the same loop using Intel Cilk Plus. Notice the only change is the replacement of the for keyword with the _Cilk_for keyword:

```
_Cilk_for (int i=0; i<1000; i++) {
      res[i] = ar[i] * 2;
}
```

Introducing Auto-vectorization

In addition to providing Intel Cilk Plus keywords, the compiler can perform auto-vectorization to target the advanced features of today’s processors. Using auto-vectorization, the compiler automatically analyzes the loops in the code and chooses the ones that are good candidates for vectorization.

The compiler then compiles the loop to at least two different forms of assembly code, including a simple, non-vectorized version for processors that don't support vectorization, as well as an advanced SIMD-based version to make use of vectorization. When running the code on processors with SIMD registers that are 16 bytes in size, the vectorization will happen automatically, resulting in as much as a four-times increase in speed. This feature happens automatically by the compiler, although the programmer can include special pragmas in the code to turn off auto-vectorization, or make suggestions to assist the compiler in the vectorization of the code. Additionally, the compiler will provide a message when a function or loop is auto-vectorized such as this:

```
SIMD LOOP WAS VECTORIZED
```

But as mentioned earlier, parallel programming requires care to certain data structures. Further, programmers shouldn't have to re-invent the wheel, so to speak. That's where Intel® Threading Building Blocks helps.

Introducing Intel® Threading Building Blocks

Whereas Intel Cilk Plus consists of three new keywords added to the C and C++ languages, Intel® Threading Building Blocks (Intel® TBB) is a complete library offering many classes and functions to assist in the coding of parallel code. Intel TBB was created by Intel as an open source project, and does not rely on the Intel Cilk Plus keywords. As such, Intel TBB can be used without special compilers.

Intel TBB primarily includes two parts: algorithms and classes. The classes include a rich set of container classes modeled after the C++ Standard Library, but with parallel features. For example, one such container is called concurrent_vector, which works similarly to the C++ standard Vector class. The algorithms include, for instance, a parallel_reduce template function that helps with the reductions described earlier in this article. There is also a parallel_sort template function that offers parallel support for sorting.

Additionally, Intel TBB includes thread local storage classes, memory allocation and deallocation classes, timing and synchronization, task scheduling, and even special exception classes that support the propagation of exception outward from spawned threads to calling threads.

Dealing with Existing Software

Existing serial software can be upgraded to run in parallel using Intel® Parallel Studio XE. With a simple menu click, a Visual Studio C++ project can be modified to make use of the Intel C/C++ Compiler, which offers full support for parallel programming and vectorization.

Once a project is switched over to the Intel compiler, the Project Properties dialog box that programmers are already familiar with now includes additional entries for configuring support for today's processors. By default, the final executable will make use of the most advanced processor features available on the computer the executable runs on, including vectorization through the SIMD features, as well as the use of as many cores and threads as possible. The compiler will automatically attempt to create vectorized forms of loops. And Intel Composer XE can help the programmer find places where loops and functions can be made to run in parallel.

Walkthrough with Intel Parallel Studio XE

Intel Parallel Studio XE is a comprehensive tool suite that provides C/C++ and Fortran developers a simplified approach to building future-proof, high-performance parallel applications for multicore processors.

Intel® Composer XE combines optimizing compilers with powerful parallel models and libraries including Intel TBB and Intel Cilk Plus.

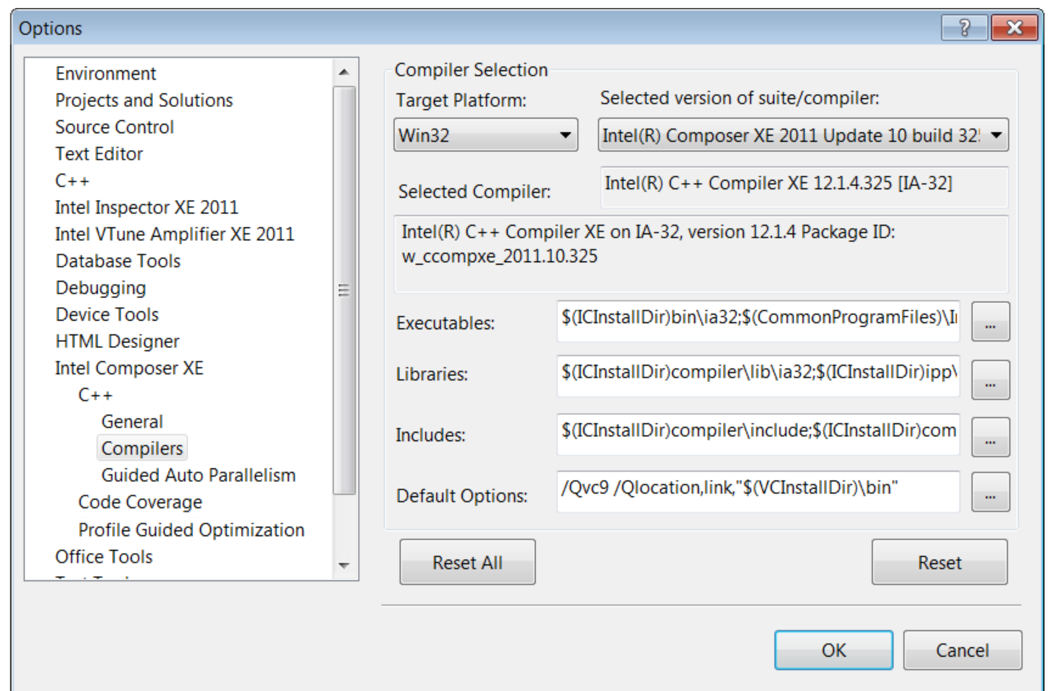
Intel® VTune™ Amplifier XE helps find performance bottlenecks in both serial and parallel code.

Intel® Inspector XE is a powerful thread and memory error checker.

Intel® Parallel Advisor is a wizard-style tool that can analyze your code and look for opportunities to parallelize your code.

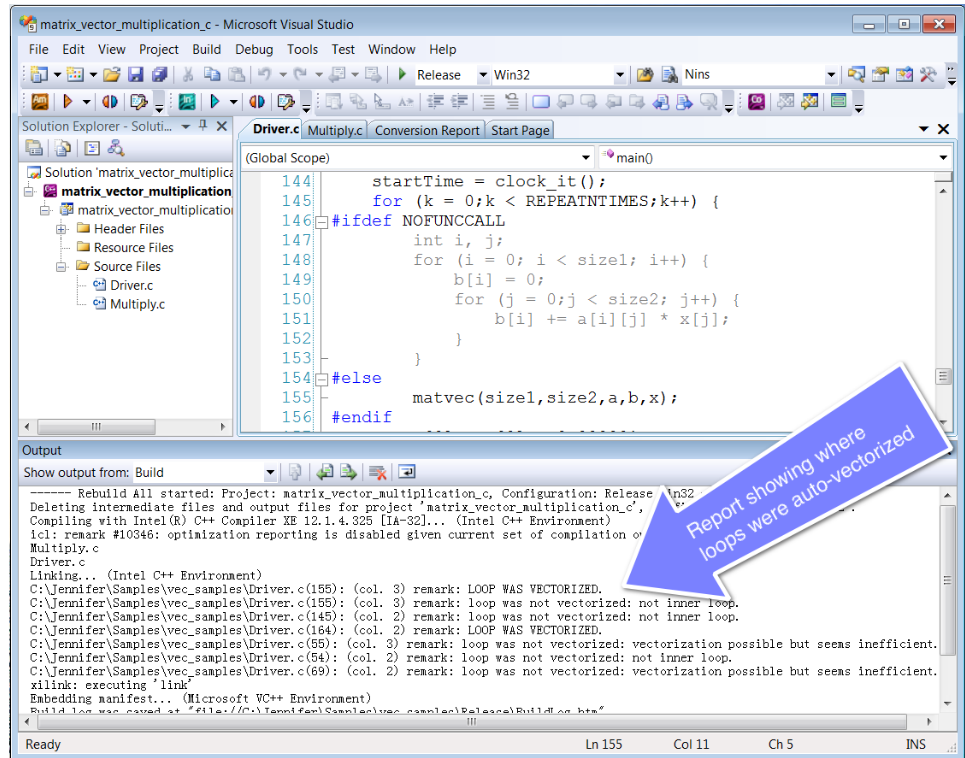
Here are some sample sessions demonstrating these features.

Part of **Intel Composer XE** is the Intel® C++ compiler. The compiler can be controlled from the new options in the project's Property Pages dialog box, as shown here:

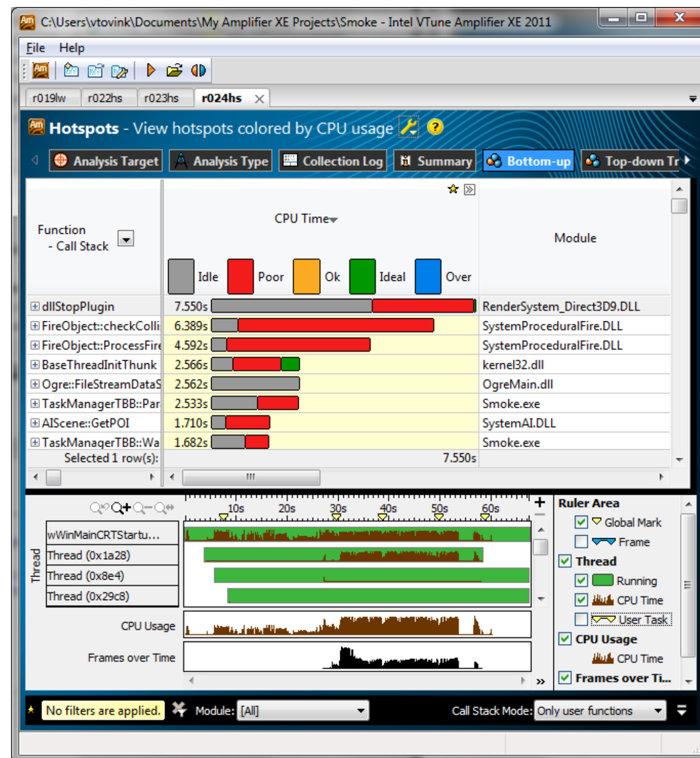


Geeknet Slashdot sourceforge free(code): intel

When you compile your application using the Intel C++ Compiler, you can see places where the auto-vectorization took place, as well as places it did not, along with a reason why it did not, as shown here:

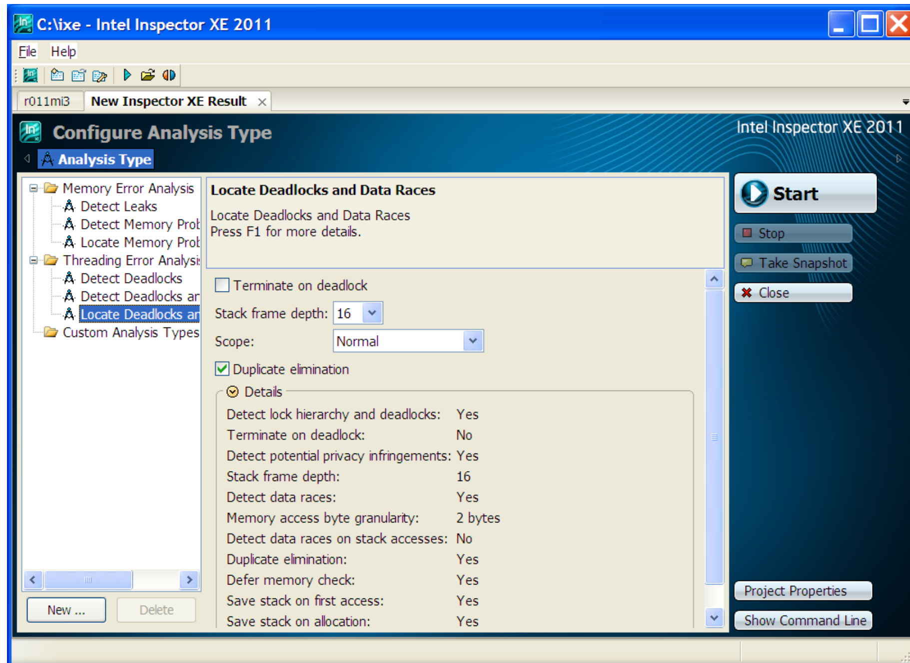


Intel VTune Amplifier XE can be used to run an analysis on your code and determine hotspots, poor concurrency, and locks and waits, as shown here:

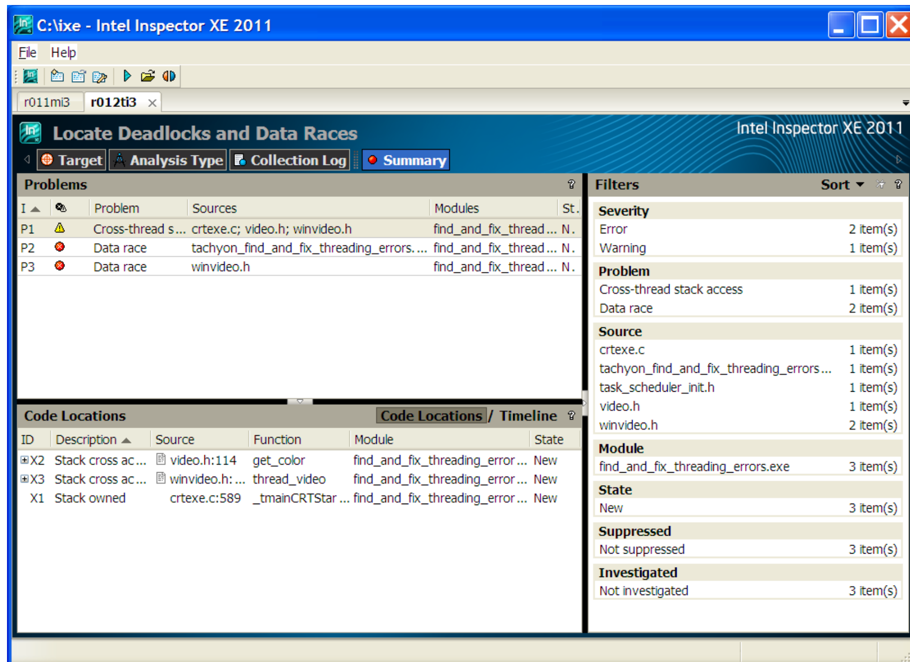


Geeknet Slashdot sourceforge free(code): intel

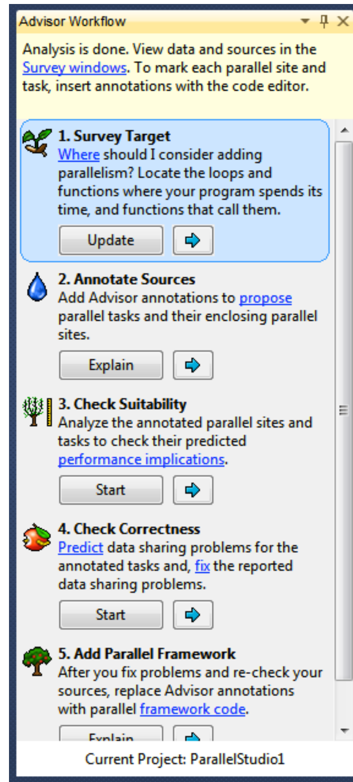
Intel Inspector XE can do a full analysis of your code to identify threading and memory errors:



After running its analysis, **Intel Inspector XE** provides results such as this:



Intel Parallel Advisor includes an entire workflow to guide you along the path of converting a serial application into a parallel application, as shown here:



Conclusion

Parallel programming has evolved substantially over the years. Programmers are still learning new ways to program in parallel after spending much time coding in serial. Programming for multiple-core processors can be difficult, between writing parallel code, maintaining data structures in parallel, and attempting to vectorize the software. Intel Parallel Studio XE is just the right tool for the job, providing parallel extensions to C++ in the form of Intel Cilk Plus, a solid library called Intel Threading Building Blocks, and a powerful suite of tools: Intel Composer XE, Intel VTune Amplifier, Intel Inspector XE, and Intel Parallel Advisor.

To learn more about Intel Parallel Studio XE, click here.

Jeff Cogswell is a Geeknet contributing editor, and is the author of several tech books including C++ All-In-One Desk Reference For Dummies, C++ Cookbook, and Designing Highly Useable Software. A software engineer for over 20 years, Jeff has written extensively on many different development topics. An expert in C++ and JavaScript, he has experience starting from low-level C development on Linux, up through modern web development in JavaScript and jQuery, PHP, and ASP.NET MVC.

Geeknet Slashdot sourceforge free(code): intel