WILL YOUR APP RUN FASTER WITH MORE CORES?

Two Tools Measure the Performance Scalability of Your Application

WHITE PAPER

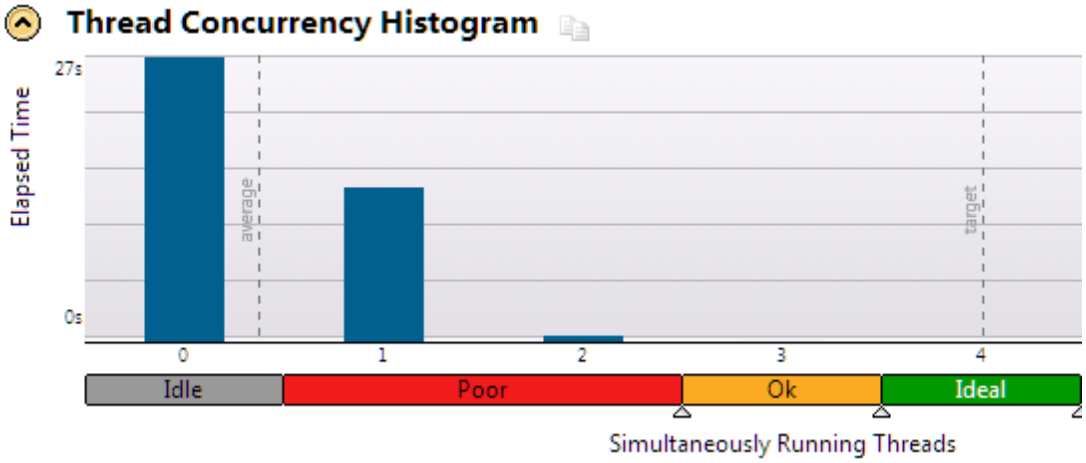
Q: Will my software performance scale if it is run on a large number of cores? How do I know if my code is or isn't parallel enough to take advantage of more cores?

Measure your application's concurrency

Don't guess, measure. Measurement is a key part of any performance plan. Applications don't always behave as we expect. Accurately measuring what your app is really doing is required to develop an effective plan for scalability.

Intel® VTune™ Amplifier XE (included as part of Intel® Parallel Studio XE) has multiple profiling tools. Running just two of these will give you an accurate picture of how you are doing on today's multicore systems and highlight scaling bottlenecks.

The first tool, concurrency analysis is one of the key views in Intel VTune Amplifier XE. It answers the question: For a given workload, what amount of time is the app running serially and what amount is in parallel? This gives a measure of best case performance scaling. Parallel code may scale, serial code will not. Concurrency analysis tells you the maximum performance you can achieve if all your parallel code scales perfectly.

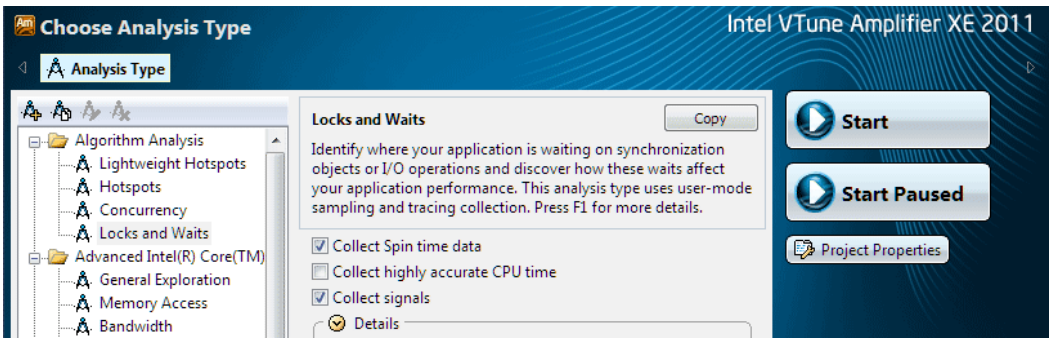


Intel VTune Amplifier XE Concurrency Summary

This is a breakdown of elapsed time of an application running on a 4 core processor. It shows the amount of wall clock time where a specific number of threads are running simultaneously. Threads are considered running if they are either actually running on a CPU or are in a runnable state in the OS scheduler.

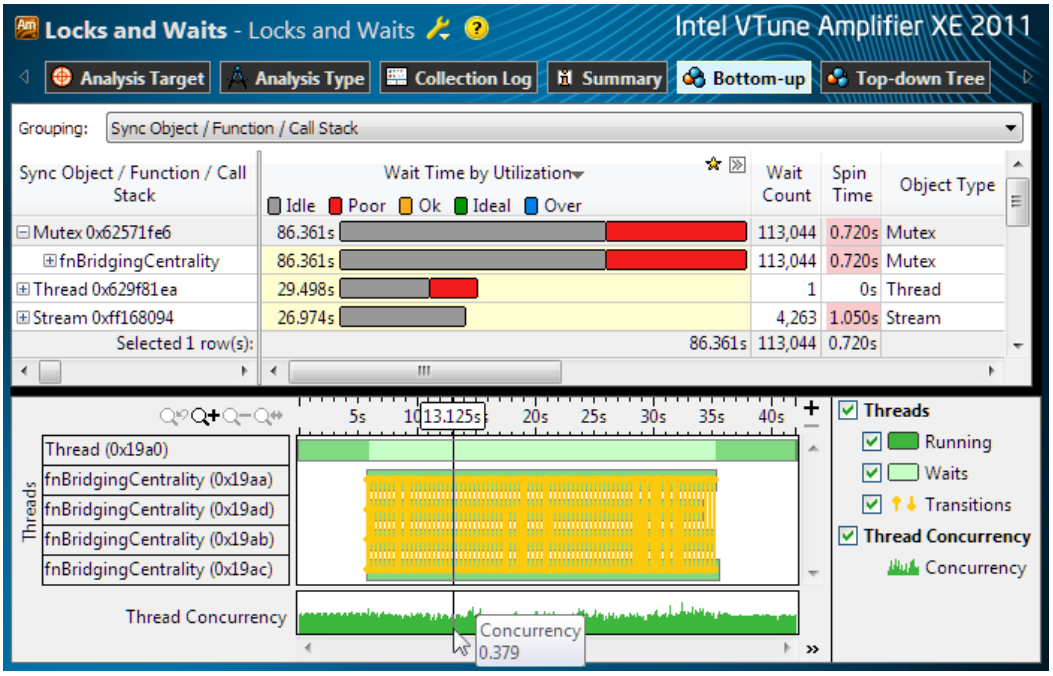
This particular app will not scale well. The locks and waits analysis in the next figure will help us see why.

The second tool, locks and waits analysis, is also in Intel VTune Amplifier XE. It will tell you if the threads you do have are doing a good job of coordinating. If you have a lot of contention, then you want to reduce it. This will both improve performance on current multicore systems and increase performance scalability on systems with more cores.



Running the Analysis is Easy

To run a Lock & Waits Analysis in Intel VTune Amplifier XE, just select "Locks and Waits" from the column of analysis types on the left, then the "Start" button.



Locks & Waits Analysis

Here we can clearly see the problem. It was designed so that 4 threads would be running most of the time, but there are two issues. First, there are 26.97 seconds of idle time because a lock (stream) is being held during I/O. Second, there is a lot of synchronization overhead.

The grid at the top is a list of synchronization objects sorted by wait time. The color coding shows the processor utilization during the wait. A long wait is ok if the processors are well utilized (green) during the wait. In this example, the processors are poorly utilized (red) or idle (grey) during the waits. The bottom row in the grid (stream) shows the idle time during I/O.

When you look for efficient scalable code it is important to look at the overhead and wait time. It is also important to understand whether threads are waiting a long time at a few locks or small increments of time on numerous locks. The timeline in the bottom half shows the synchronization as many yellow transitions between the running threads. Consequently, the actual concurrency (bottom row of timeline) is very low. Numerous transitions may show excessive synchronization calls and may merit a review of the code or algorithm to find ways to reduce the number of synchronizations or barriers.

Will it scale?

Now that you have a picture of how your app is behaving it is time to reflect on a few basic design issues.

Does the number of threads adjust to match the hardware?

Have you designed your parallelism so that the number of threads can be automatically or easily adjusted to match the optimum number of threads the hardware can run? If there are too few threads, you won't take advantage of the hardware. Too many and execution is inefficient. If you have used a high level model for parallelism like Intel® Threading Building Blocks (Intel® TBB) this is automatic. If you have done your own OS level threading, it is something you must manage.

Will the granularity be correct for a large number of cores?

The workload for each thread needs to be heavy enough that you get payback for the thread overhead. You can measure this using the concurrency analysis in VTune Amplifier XE, or you can just time program execution. Let's say that the measurements you make indicate that your workload is appropriate when you run it on a 16 core system. (Appropriate scaling is your decision – are you comfortable with 10X scaling on 16 cores? Or do you expect 12X or 14X? Different projects have different scaling expectations.) Your application is performing well, but you want to know what will happen when you move to a 64 core system. Try re-analyzing your application on today's system, but re-size the workload so it matches the size per core you will use on the future system with more cores. If the threading overhead is still reasonable, then your granularity will most likely be fine in the future.

Does the synchronization overhead increase?

When the number of threads increases, does the number of synchronization points increase? If so, what is the degree of synchronization? If the number of synchronization points does not increase or if the increase ispair-wise or low degree of interation and not global (all threads), then your synchronization overhead is likely to be fine. If you are unsure, run an experiment with more threads and view the synchronization using Intel® VTune™ Amplifier XE.

As you work through these design issues, it is a good time to think about your model for parallelism. One of the easiest ways to design for scalability is to adopt a higher level model for parallelism.

Express parallelism at a higher level

We are all familiar with the productivity advantages of writing software in high level languages instead of in assembly. The same is true for parallelism. Using low level constructs like Pthreads and Windows* Threads is equivalent to writing assembly. You have maximum control, but productivity suffers and performance optimizations may need to change with new hardware.

No one high-level construct for parallelism is best for all applications. There are a number of high level constructs depending upon your needs.

OpenMP* is a popular choice for C and Fortran programmers, and is well supported by many companies including Intel. If it works well for you, there is probably no reason to change, however programs without OpenMP* today have better options for parallelism. OpenMP* lacks the composability, in particular the ability to nest parallelism well, that Intel TBB and Cilk Plus can offer. Still, it is a good solution if you already have a major investment in it.

Intel® Threading Building Blocks is a template library based solution for C++. It uses task-based abstractions that make it easier to get scalable and reliable parallel applications. With Intel® TBB as your parallel development model you have a known scalable solution that will be optimized for new generations of hardware. Commercial versions of Intel® TBB are available for Windows*, Linux* and Mac OS*. Or use the open source version. TBB offers the ability to express a wide range of algorithms. In addition it offers a scalable memory allocator as well as highly optimized thread controls for the advanced user.

Intel® Cilk™ Plus is an extension to C and C++. The three Intel Cilk Plus keywords provide a simple yet surprisingly powerful model for parallel programming, while runtime and template libraries offer a well-tuned environment for building parallel applications. Intel® Cilk™ Plus is now available in Intel Parallel Studio XE and a subset of Cilk Plus is currently added to a development branch of the GCC 4.7 project. Cilk™ Plus is the most efficient and requires the fewest code changes and this makes it a popular choice.

Coarrays as defined in the Fortran 2008 standard allow Fortran developers to program in parallel. As an extension to the Fortran language, coarrays offer one method to use Fortran as a robust and efficient parallel programming language. Coarray Fortran uses a single-program, multi-data programming model (SPMD) and is supported by Intel Parallel Studio XE.

These are just a few examples of the parallel development choices available in Intel Parallel Studio XE. Pick the one that is the best fit for your app. No matter which construct you pick, you see the benefits of moving to a higher level of abstraction. Productivity will increase and the compiler or library that implements the higher level parallelism will optimize performance for new hardware platforms.

Summary

It is a multicore / manycore world. Concurrency and scalability are vital to future performance growth. Using tools to characterize your application's performance is the key to predicting performance on future hardware with increased core counts. Expressing parallelism at a higher level of abstraction improves productivity and increases your chances for performance scalability.

Try it yourself

Download a free evaluation copy of Intel® Parallel Studio XE. It contains:

- Intel® VTune™ Amplifier XE performance analyzer
- Parallel development models including Intel® Threading Building Blocks (Intel® TBB)
- Intel® Inspector XE memory and thread checker
- Static security analysis to find coding errors and harden software security.
- Optimizing compilers and performance libraries

Suggested Reading

- Intel Guide for Developing Multithreaded Applications
- Webinar – "The Key to Scaling Applications for Multicore"
- Intel® VTune™ Amplifier XE – a short overview movie and detailed step-by-step getting started tutorials.
- Intel® TBB white papers:
  - Enable safe, scalable parallelism with Intel® TBB concurrent containers
  - Demystify scalable parallelism with Intel® TBB generic parallel algorithms
  - Intel® TBB: Scalable programming for multi-core

About the Author

Dick Kaiser is an Intel Product Marketing Engineer specializing in software analysis tools.

Notices