به نام خدا

برنامه‌نویسی چندهسته‌ای

دستور کار آزمایشگاه ۳

دانشگاه صنعتی امیرکبیر
(پلی تکنیک تهران)

مقدمه

در این آزمایش شما برنامه‌ی سریال مربوط به آزمایش اول را در محیط Visual Studio و به کمک ابزارهای Parallel Studio به صورت اصولی موازی می‌کنید.

آزمایش

❖ **ساخت پروژه در ویژوال استودیو، تنظیم پروژه، کامپایل و اجرای کد**

- کد برنامه به نام Lab-1-1.cpp در فایل آزمایش ۳ قرار دارد.
- به تنظیمات اجرا و کامپایل توجه کنید. برای اطمینان از صحت تنظیمات می‌توانید از تابع قرار داده شده در moodle استفاده کنید.

THE FOUR-STEP METHODOLOGY[1]

Initially, parallelizing a serial program may seem fairly simple, with the user following a set of simple rules and applying common sense. But this may not always achieve the most efficient parallel program running at the expected speeds. Indeed, it is possible that faulty attempts at parallelization will actually make a program run more slowly than the original serial version, even though all parallel cores are running. The four-step methodology, as shown in Figure 1, is a tried and tested method of adding parallelism to a program.

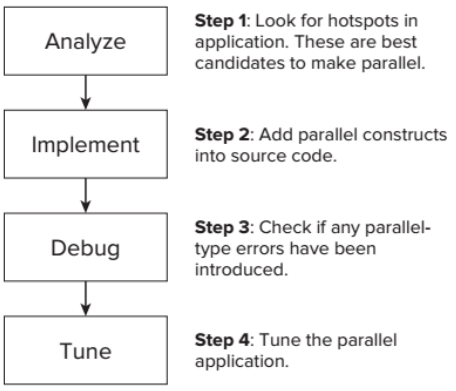


Figure 1 The four-step methodology

1. Analyze the serial program for opportunities to parallelize. This is probably the most important step; decisions made here will affect the final parallelized program.

2. Implement the parallelism using constructs from the parallel model you have chosen to use.

3. Debug or check if any parallel-type errors have been introduced. Is the program running correctly? Does it have threading or memory errors?

4. Tune the parallel application. Are all the threads doing equal amounts of work? Has an excessive overhead been introduced into the program as a result of adding parallelism?

With the exception of the debug step, you should carry out the steps on an optimized version of the application.

Step 1: Analyze the Serial Program

[1] Blair-Chappell, Stephen, and Andrew Stokes. Parallel programming with intel parallel studio XE. John Wiley & Sons, 2012

The purpose of this step is to find the best place to add parallelism to the program. In simple programs you should be able to spot obvious places where parallelization might be applied. However, for any program of even just moderate complexity, it is essential that you use an analysis tool, such as Intel Parallel Amplifier XE. Although this is a rather trivial programming example, you can use the steps on more complex programs.

Using Intel Parallel Amplifier XE for Hotspot Analysis

When Intel Parallel Studio XE was installed into Microsoft Visual Studio, it set up a number of additional toolbars, one of which is for Intel Parallel Amplifier XE. Amplifier is a profiling tool that collects and analyzes data as the program runs. This example uses Amplifier XE to look for parts of the code that are using the most CPU time; referred to as *hotspots*, they are prime candidates for parallelization. Because Amplifier XE does slow down the execution of the program considerably, it is recommended that you run an application with reduced data. Provide data input and reduce loop iterations, where possible, to reduce the run time. For this example, the outer loop is reduced to 1. This will not prevent Amplifier from finding the hotspots, because the outer loop merely runs through the same work loop several times. Hotspots found in the first iteration of the work loop will be the same in any further iterations of it. Also, leave VERYBIG set as 10000. You will need to rebuild with these new settings before using Amplifier.

Starting the Analysis

To start the analysis, follow these steps:

1. Select New Analysis from the Amplifier XE part of the toolbar, as shown in Figure 2. This brings up the start-up page.

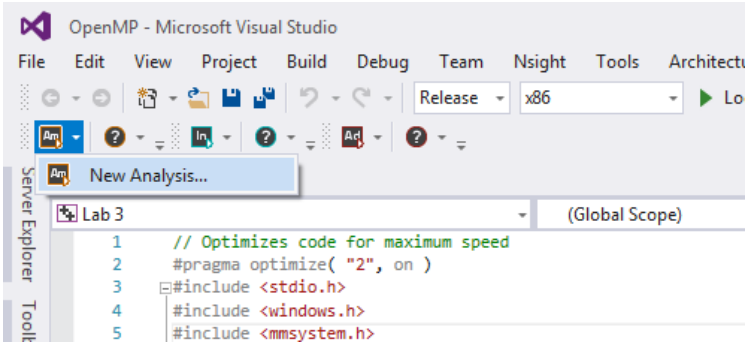


Figure 2 Selecting a new Amplifier analysis

2. Select the analysis type Basic Hotspots, as shown in Figure 3. Hotspot analysis looks for code that is consuming the most CPU activity.

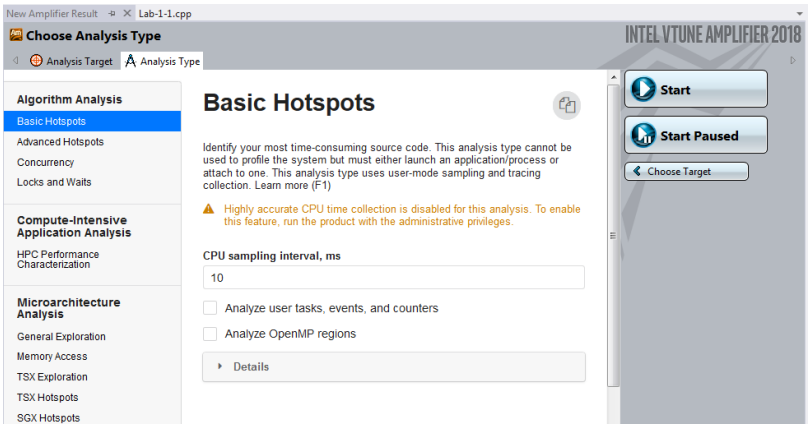


Figure 3 Start-up page of the Amplifier XE

3. Click the Start button. Amplifier runs a hotspot analysis on your program. Because there is no pause at the end of your program, Amplifier will both start and finish your program itself. Figure 4 shows the results.

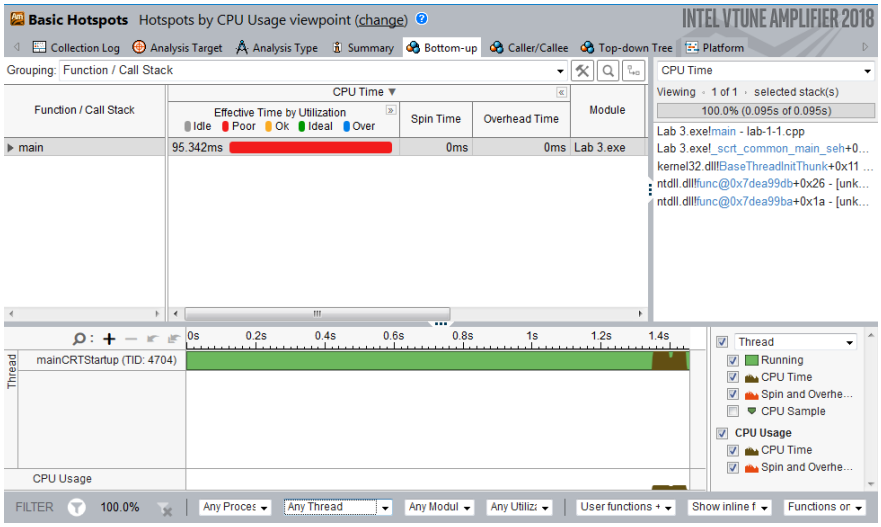


Figure 4 Hotspot analysis using Amplifier XE

Drilling Down into the Source Code

Figure below shows how much CPU time was spent in each function. In this example, because there is only a single function, main, only this one entry is present. To examine the source code of the hotspot, double-click the entry for function main. This reveals the program code, with the hotspots shown as bars to their right with lengths proportional to CPU time spent on each line (Figure 5). Note that the code pane has been expanded within the Amplifier window, and that line numbers within Parallel Studio have been turned on. In the code shown, Amplifier automatically centers on the line of code that consumes the most CPU time — in this case, line 36. Two groups of three lines are using most of the CPU time; these involve the loops calculating the two arithmetic series. The remaining code lines consume little CPU time in comparison and so show nothing. These arithmetic loops are the hotspots within your program. Your own computer system may give different times, but it should follow a similar pattern. You should also note that the Amplifier results for this run are placed in the Amplifier XE folder under the project solution. You can see this to the left of the screen.

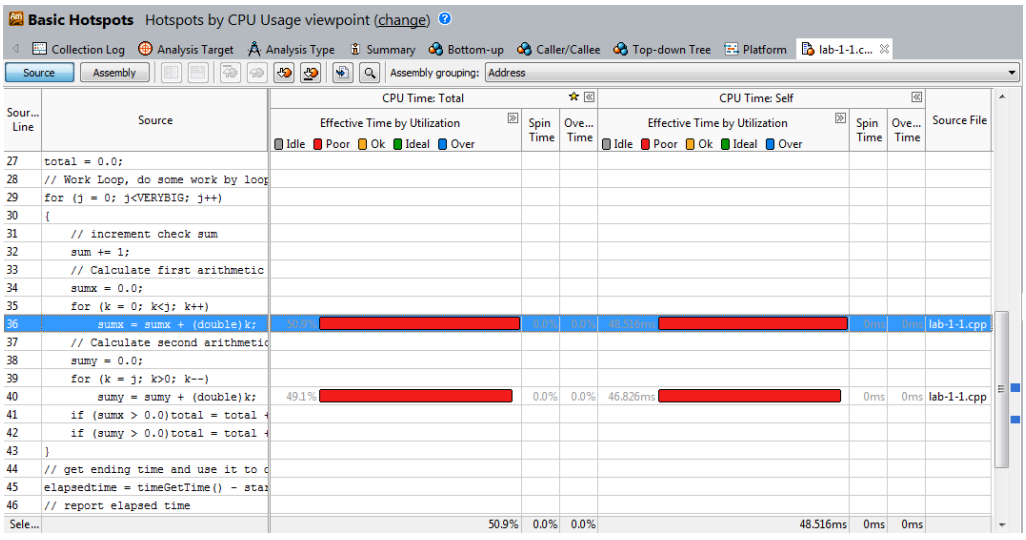


Figure 5 Hotspot analysis using Amplifier XE, showing hotspots

Parallelization aims to place hotspots within a parallel region. You could just attempt a parallelism of each of the arithmetic loops. However, parallelization works best if the largest amount of code can be within a parallel region. Parallelizing the work loop places both sets of hotspots within the same loop.

Step 2: Implement Parallelism using OpenMP

To add parallelism to the original serial program, follow these steps:

1- Add an OpenMP directive immediately before the loop to be parallelized. We apply parallelism to the work loop:

```
// Work loop, do some work by looping VERYBIG times
#pragma omp parallel for
for( int j=0; j<VERYBIG; j++ )
{
```

2- Add an additional include file:

```
#include <omp.h>
```

When the OpenMP directive is encountered, a parallel region is entered and a pool of threads is created. The number of threads in the pool usually matches the number of cores. Execution of the *for* loop that follows is parallelized, with its execution being shared between the threads.

Step 3: Debug and Check for Errors

With the introduction of parallelism into the program, the program no longer runs correctly. The problem is, by introducing parallelism, you also introduced problems caused by concurrent execution. In the next few steps you investigate how to fix these problems by enhancing both the speed and performance of the application.

Checking for Errors

You can find data races and deadlocks by using Intel Parallel Inspector XE. It is recommended that you perform any error checking on the debug version of the program, not the Release version. Building in the Release version will carry out optimizations, including in-lining, which may accidentally hide an error. Using the debug build also means that the information reported by Inspector is more precise and more aligned with the actual code written. Running an Inspector analysis is a lot slower than just running the program normally. As with Amplifier, you should reduce the running time by reducing loop counts and using small data sets.

To check for errors, follow these steps:

1. Change the solution configuration to be a debug version, but don't rebuild just yet.

2. Because Inspector is slow, reset VERYBIG to just 1000, and reduce the outer loop to be just 1:

```
// repeat experiment several times

for (i=0; i < 1; i++)
```

Errors found in the first iteration of the loop will just be repeated in further loops, so there is no point in having more loops.

3. Rebuild the application.

4. Launch Inspector XE from the toolbar, and select New Analysis (Figure 6).

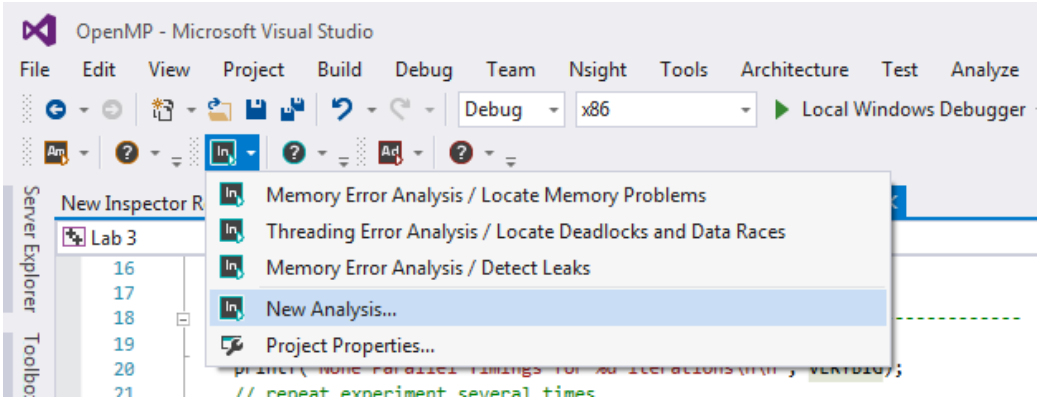


Figure 6 Selecting a new Inspector analysis

4. In the opened window (Figure 7), from Threading Error Analysis select the analysis type Locate Deadlocks and Data Races and check Use maximum resources.

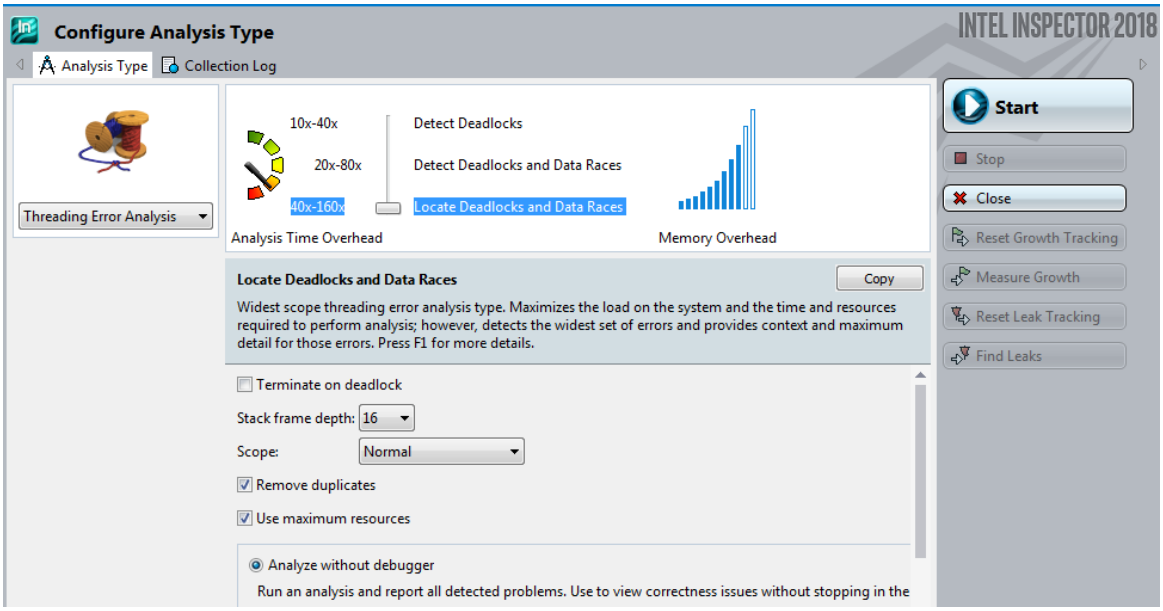


Figure 7 Selecting for locating deadlocks and data race

5. Inspector runs your program, carrying out an analysis as it does so. Inspector targets the analysis to find deadlocks and data races, with the results as shown in Figure 8. The top-left pane of the Inspector window summarizes the problems. The bottom-left pane shows the events associated with any selected problem. Try clicking on the various problems. In Figure 8 problem P1, a data race, is selected; its associated events are listed in the lower pane. Altogether, four problems, P1 to P4, are shown as data races. These are marked with an *x* in a red circle.

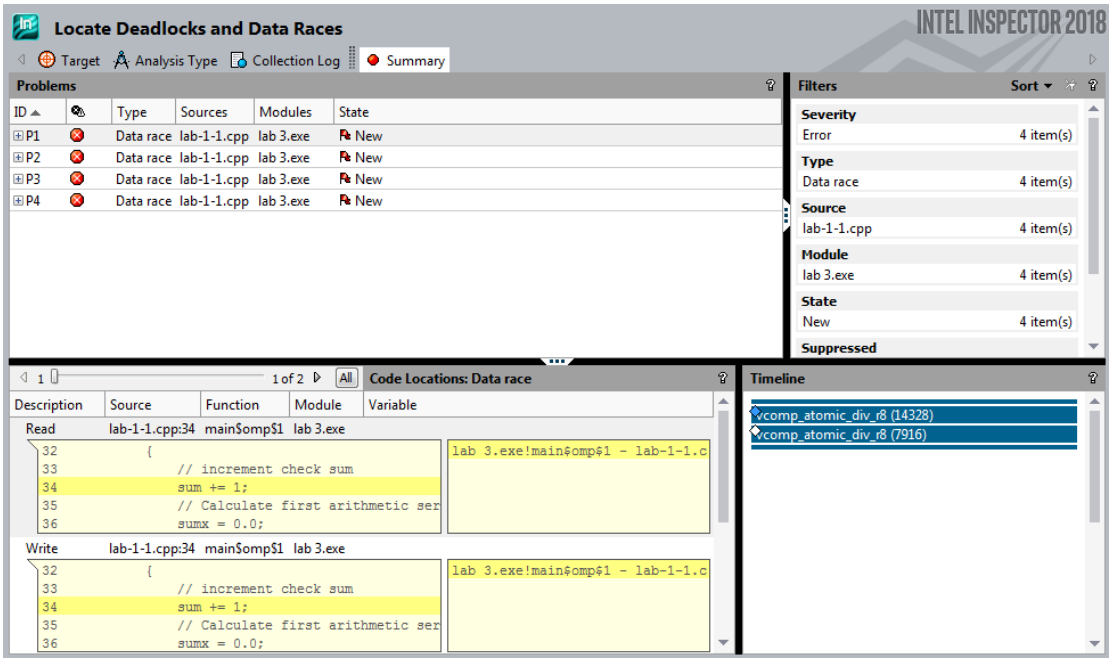


Figure 8 Summary of threading errors detected by the Inspector XE analysis

6. Double-click the P2 problem to reveal the code associated with it (Figure 9). Two code snippets are shown, with the location of one of the events in each. The top code pane, Focus Code Location, shows where the data race was detected during a write event. The lower of the two code snippets, Related Code Location, shows the read event that was involved in the data race.

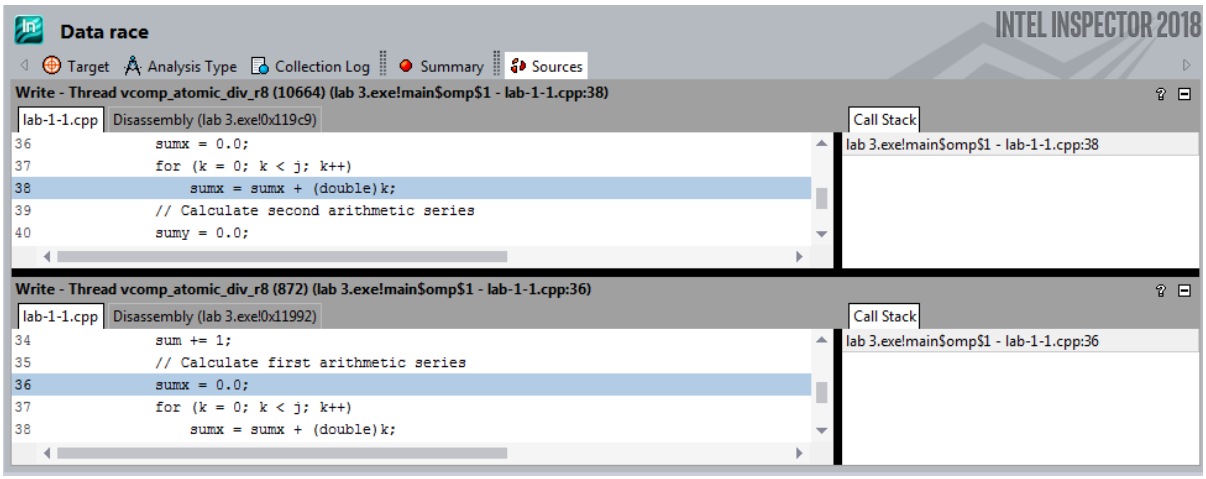


Figure 9 A data race exposed in the source code

From these two code snippets you can determine that variable sumx is the problem. The Focus Code Location pane shows the variable being changed (write), at line 36 of the code. The Related Code Location pane shows the variable being read, at line 38 of the code. When multiple threads are running there arises the danger of one thread changing the value of sumx (resetting to 0), while a second concurrently running thread is still using it, thereby making the second thread have an incorrect value. This is referred to as a *data race*. Examining the other data race problems, you should be able to determine which variables they involve.

The full list of variables causing data races is sum, total, sumx, sumy, and k. All five of these variables were created at the start of the function, and their scope is that of the function. However, the arguments that follow would be the same regardless of whether these variables are global or static. During parallel execution all the threads are competing to read from and write to these function-scoped variables. These are referred to as *shared variables*, and they are all shared by the concurrently executing threads. For future reference, they will be referred to as *nonlocal variables*.

Making the Shared Variables Private

You can fix the data races caused by variables *sumx*, *sumy*, and *k* by creating private variables for each thread by adding a private clause to the parallel pragma directive, as follows:

```
#pragma omp parallel for private( sumx, sumy, k )
```

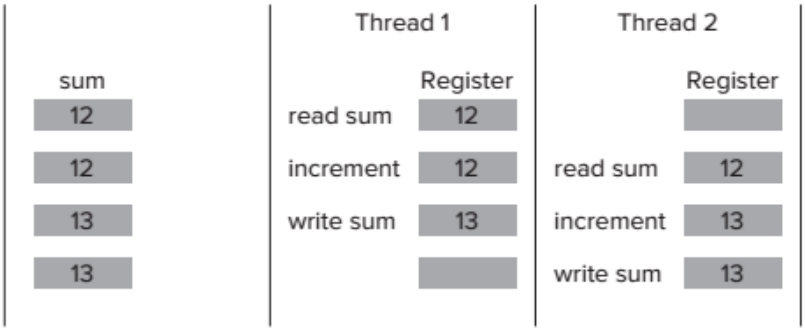
In an OpenMP parallelized *for* loop, its loop counter is, by default, always made private. The variables in the list must already exist as declared nonlocal variables — that is, as automatic, static, or global variables. When the worker threads are created, they automatically make private versions of all the variables in the private list. During execution each thread uses its own private copy of variables, so no conflicts occur and the data races associated with these variables are resolved.

Adding a Reduction Clause

If you run the program you will notice that the Total and Check Sum values are incorrect and inconsistent between the runs. These errors are also caused by data races, but making thread-private copies of the offending variables total and sum locally within the loop will not help in this case, because these variables must be shared between all the threads. Figure 3-18 illustrates what happens when more than one thread attempts to increment the check sum in the code line:

```
sum += 1;
```

As you can see from Figure 3-18, if sum starts with a value of 12, after both threads have incremented the result is 13, instead of the expected 14.



Variables within the loop that must be shared by the threads cannot be made private. This is handled in OpenMP by adding a reduction clause to the OpenMP directive:

```
#pragma omp parallel for private( sumx, sumy, k ) reduction( +: sum, total )
```

When the parallel section is reached, each participating thread creates and uses private copies, or versions, of the listed reduction variables. When the parallel section ends, the private thread versions of the variables are operated on according to the operator within the reduction brackets — in this case, they are added together. The resultant value is then merged back into the original nonlocal variable for future use. Other operators, such as multiply and subtract (but not divide), are also allowed. Note that it is up to the programmer to ensure that the operation on the variables within the loop body matches the operator of the *reduction* clause. In this case, both *sum* and *total* are added to each iteration of the work loop, which matches the *reduction* operator of +.

Step 4: Tune the OpenMP Program

To use Intel Parallel Amplifier XE to check for concurrency and efficiency within the OpenMP program, follow these steps:

1. Start a new analysis from the Parallel Studio menu bar, as before.

2. Select Concurrency from the list of analysis types.

3. Run the Amplifier for concurrency analysis by clicking its Start button. Amplifier will now run your program and generate a new output of results.

4. Click the drop-down button (Figure 10) to obtain a list of alternative ways to display the information, and select /Thread/Function/Call Stack. The result should be as shown in Figure 11. The various panes have had their borders moved to produce the display shown. Your actual results may look different to that shown, depending on the number of cores your machine has.

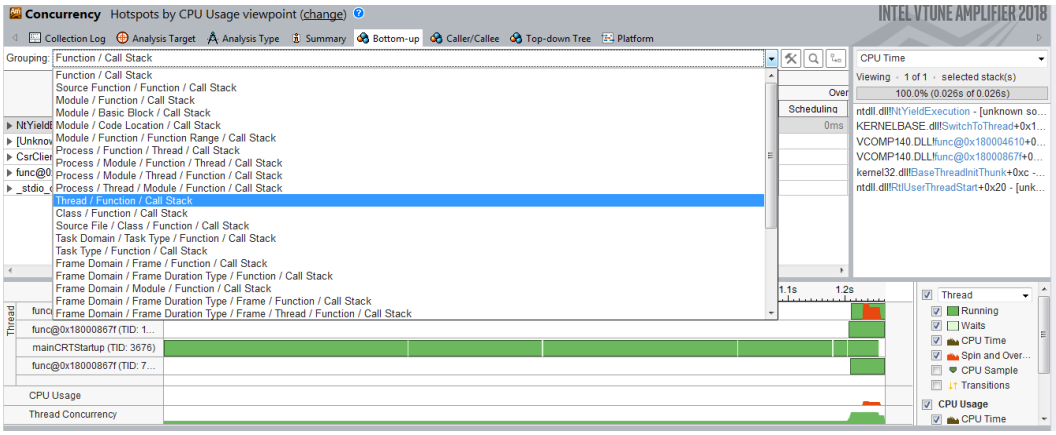


Figure 10 Selecting for viewing concurrency information

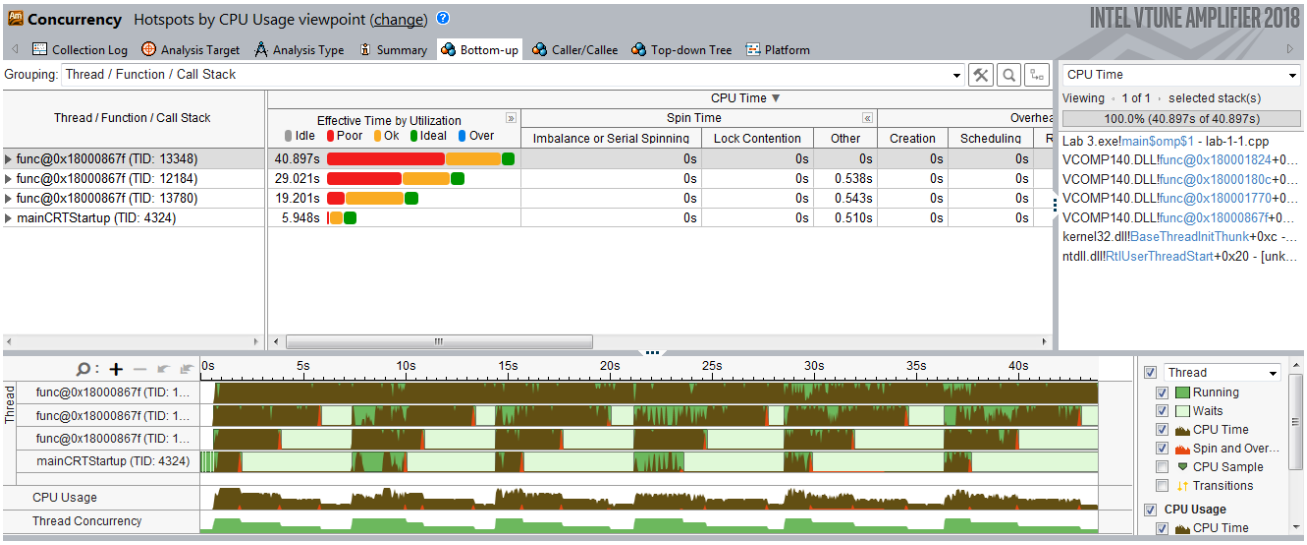


Figure 11 Concurrency information showing unbalanced loads

The top pane shows the mainCRTStartup thread, but now there are only three other OMP Worker Threads — because the main (serial) thread is also used as one of the parallel threads. The top pane, which shows CPU utilization time for each thread, clearly indicates an imbalance between the threads, with the mainCRTStartup thread doing less than a third of the work of OMP Worker Thread 3. This imbalance is reflected in the lower pane, where the timelines of each of the parallel threads are given. Brown shows when a thread is doing work, and green shows when it is idle. You can clearly see that the mainCRTStartup thread is doing little work compared to the OMP Worker Thread 3. Also notice the sixfold pattern to the work caused by the outer loop running six times.

The CPU Usage and Thread Concurrency timelines both show poor performance. This is demonstrated in Figure 12, which can be obtained by clicking the Summary button. This shows the amount of time spent with 0, 1, 2, 3, and 4 threads running concurrently. Only a small part of the time are four threads running together. Overall, an average of 2.26 threads were running concurrently, which agrees almost exactly with the increase in speed.

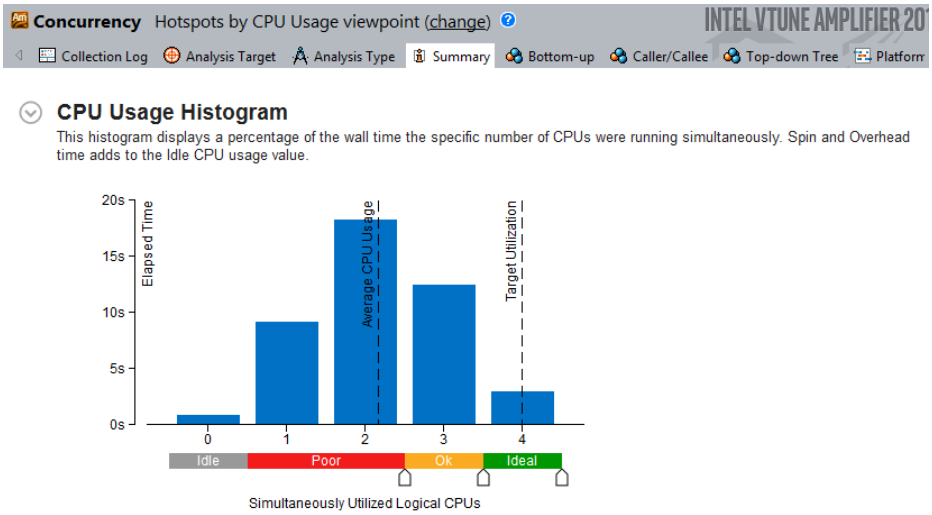


Figure 12 Amplifier showing thread concurrency for the OpenMP program before tuning

The problem occurs because the arithmetic series required more terms as the iteration count of the work loop grew larger. As the work loop counter increases, so too does the amount of work required to calculate the arithmetic series. However, the default scheduling operation of OpenMP is to simply divide the execution of the work loop between the available threads in a straightforward fashion. Each thread is given the task of iterating the work loop a fixed number of times, referred to as the *chunk size*. For example, on a 4-core machine, with an iterative count of 100,000, OpenMP simply divides the iterations of the loop by four equal ranges:

- One thread is given iterations for loop counter 0 to 24,999.
- The next thread is given iterations for loop counter 25,000 to 49,999.
- The next thread is given iterations for loop counter 50,000 to 74,999.
- The final thread is given iterations for loop counter 75,000 to 99,999.

For most purposes, this would be a balanced workload, with each thread doing an equal amount of work. But not in this case. Threads working through higher-value iterations encounter arithmetic series with greater numbers of terms, the number of terms being dependent on the iteration value. This means that the threads working on the higher iterations have to do more work, creating unbalanced loading of the threads. Figure 13 demonstrates a simplified problem with unbalanced loads. In this example a serial program enters a loop with a count of 12, where each iteration of the loop carries out work whose execution takes longer. This is indicated by the blocks marked 1 through 12 on the top bar, labeled Single Thread (78 time units). This bar represents the running of the serial program, where the width of each block is the time taken to execute each of the 12 loops. In this example the first loop takes 1 time unit, the second loop takes 2 time units, and so on, with the final loop taking 12 time units — making a total time of 78 time units to run the serial program.

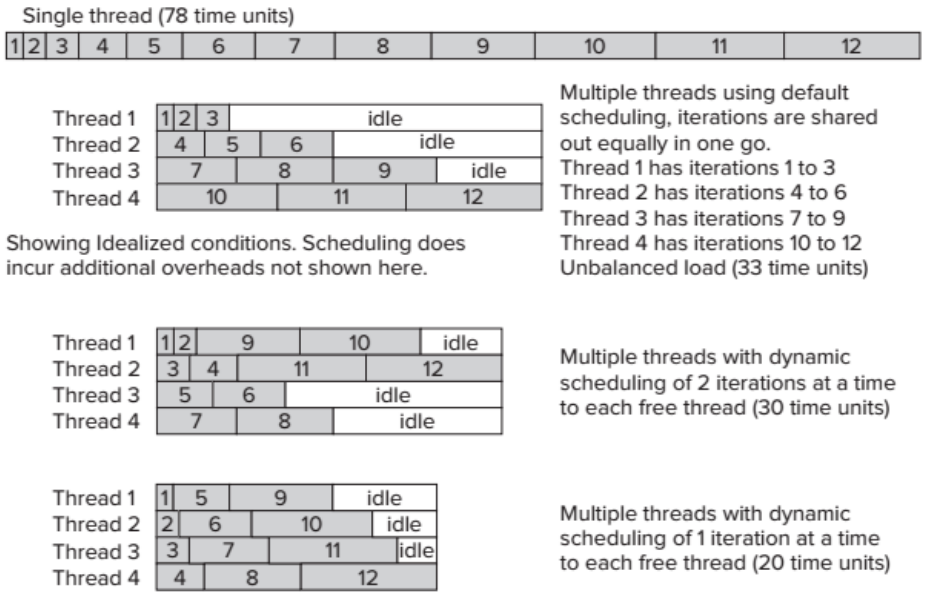


Figure 13 Example demonstrating unbalanced loading

Ideally, the parallelized time to run the same loop on a 4-core machine should be 78/4 = 19.5 time units; however, this is not the full story. The default scheduling behavior for OpenMP is to distribute the loops equally to all the available threads — in this case, as follows:

- Thread 1 is allocated iterations 1 to 3.
- Thread 2 is allocated iterations 4 to 6.
- Thread 3 is allocated iterations 7 to 9.
- Thread 4 is allocated iterations 10 to 12.

When you run the parallel program under these default conditions, the result is shown by the second block of Figure 13. Thread 4 has all the high iterations, which take more time. Thread 4's running time is 10+11+12=33 time units, which is clearly shown in the diagram. Concurrently running thread 3 takes only 7+8+9=24 time units to execute its allocated work. Thread 2 takes 15 time units, and thread 1 takes only 6 time units. Since all threads must synchronize at the end of the parallelized loop before continuing, it means that threads 1 to 3 must wait for thread 4 to complete. A lot of time that could be otherwise used is wasted. You can alter the scheduling preferences of OpenMP by using the scheduling clause. This clause enables you to set how many iterations each thread will be allocated — referred to as the chunk size. Each thread will execute its allocated iteration of the loop before coming back to the scheduler for more.

The third block of Figure 13 shows what happens when a new scheduling preference of 2 is used. At the start of the loop each available thread receives two iterations, as follows:

- Thread 1 is allocated iterations 1 and 2.
- Thread 2 is allocated iterations 3 and 4.
- Thread 3 is allocated iterations 5 and 6.
- Thread 4 is allocated iterations 7 and 8.

In this case thread 1 quickly executes its 2 allocated iterations, taking only 3 time units to do so. It then returns to the scheduler for more, and is given iterations 9 and 10 to execute. Thread 2 also finishes its allocated work, taking 7 time units, before returning back to the scheduler to be given iterations 11 and 12 to execute. When threads 3 and 4 finish, they also return back to the scheduler, but since there is no more iterations to be executed they are given no more work. These threads must idle until the other threads finish their work. The first thread must also idle for a time since the execution of its allocated iterations finishes before the second thread. All this is clearly shown by the third block of Figure 13, where the second thread dictates the overall time of execution — in this case, 30 time units. The fourth block of Figure 13 demonstrates what happens if the chunk size for the scheduler is reduced to just 1. The overall time of execution, decided by the fourth thread, reduces to just 20 time units and minimal idle time. There is an overhead because

scheduling chunks takes time; too small of a chunk size could end up being detrimental to the operation. Only by trying various values can you find the correct chunk size for your particular program.

Improving the Load Balancing

To obtain a balanced load, you need to override the default scheduling behavior. In this case the loop iterates 100,000 times, so as a first attempt use a chunk size of, say, 2000. You can override the default scheduling algorithm for a *for* loop by using the schedule clause on the directive:

```
#pragma omp parallel for private( sumx, sumy, k ) reduction( +: sum, total ) schedule( dynamic, 2000 )
```

This causes the OpenMP directive to use the fixed chunk size given in parentheses — in this case, 2000. After each thread finishes its chunk of work (2,000 iterations), it comes back for more. This divides the work more evenly. Adjusting the size of the chunk fine-tunes the solution further. After rebuilding the solution with these changes, run Parallel Amplifier again to check for concurrency. Again, select Thread/Function/Call Stack. Figure 14 shows the result.

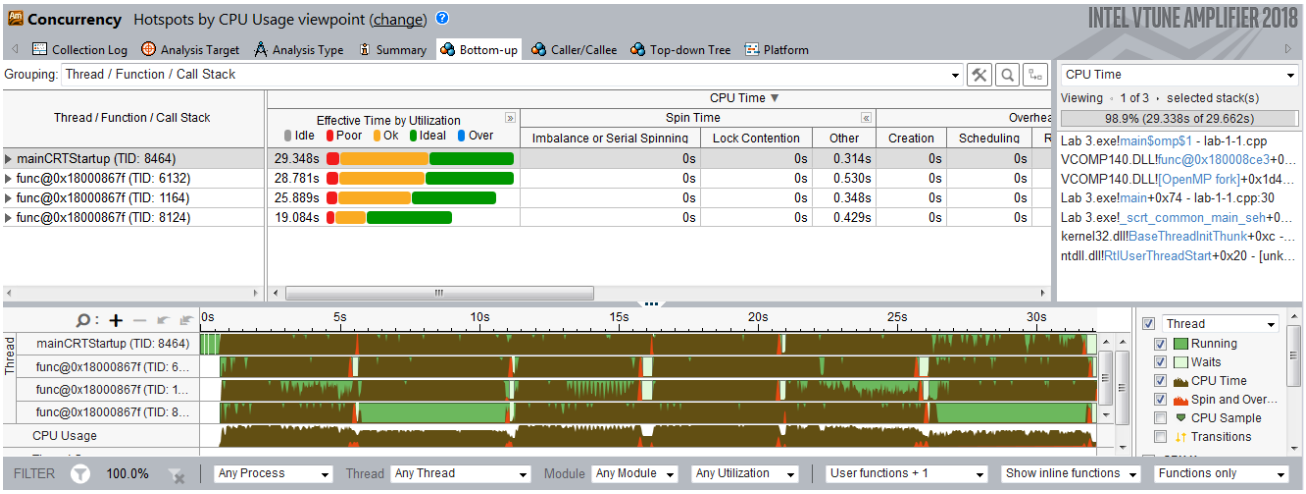


Figure 14 Concurrency showing balanced (but still not ideal) loads

The figure shows nicely balanced loads, but the load bars show mainly Ok (orange), not Ideal (green). Figure 15 shows the Thread Concurrency Histogram, with the average number of threads running concurrently still only 2.76.

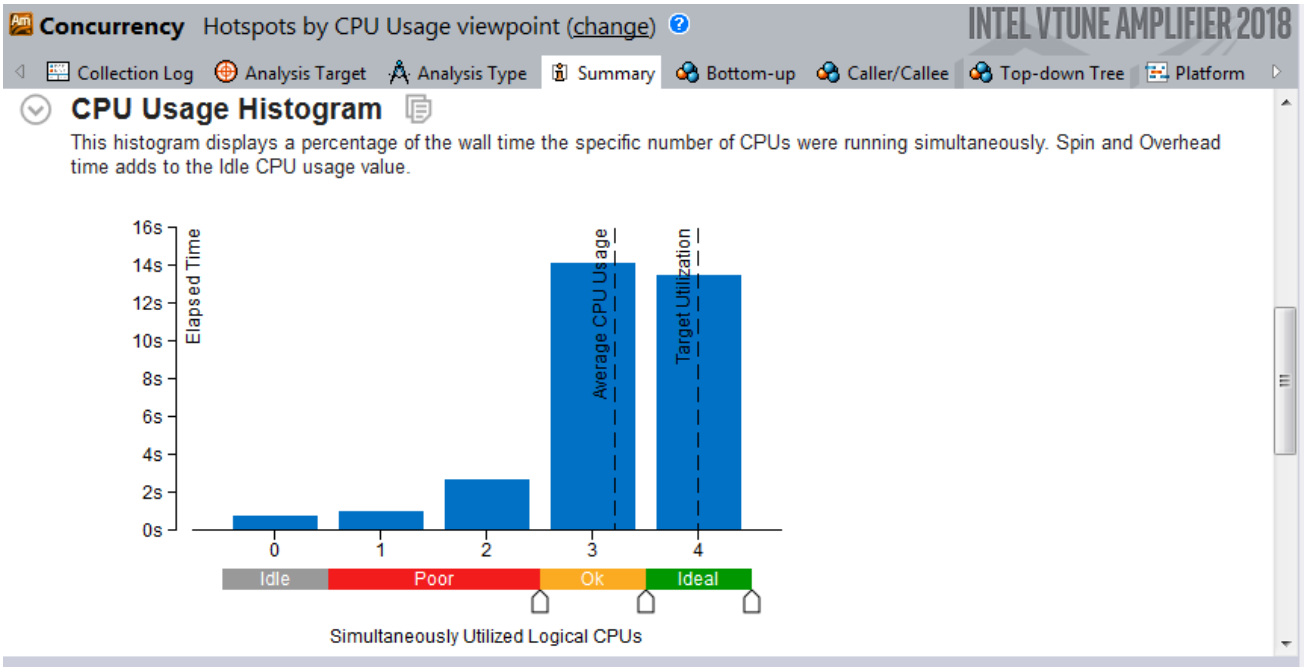


Figure 15 Concurrent information after first tuning attempt

You can try tuning further by changing the chunk size to 1000. Figure 16 shows the results of the concurrency; now, the program spends more time using four threads (Ideal state (green)) and a better average thread concurrency is achieved. You can further try different values to achieve the best speedup.

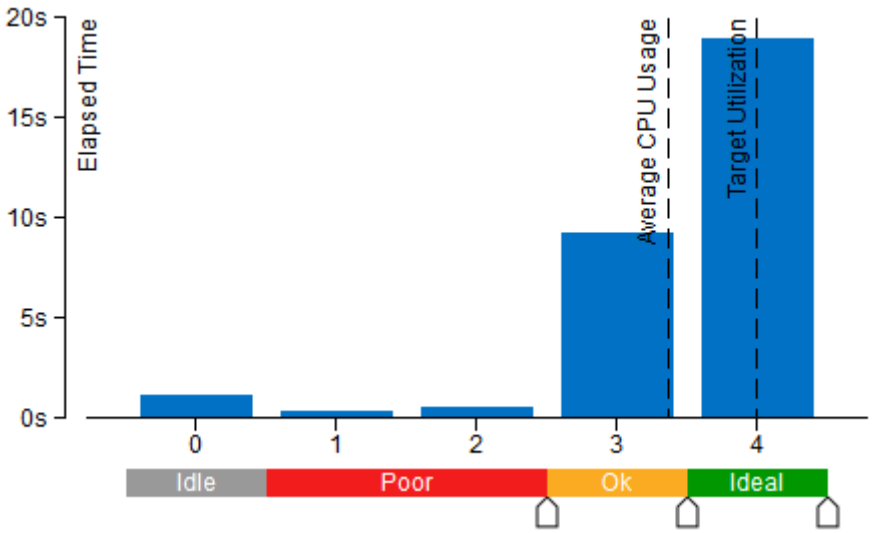


Figure 16 Concurrent thread information after final tuning

The timings were generated on a 4-core computer system and may differ from your timings, depending on what system you are running.