# Getting Started with the Intel® Parallel Amplifier

The Intel® Parallel Amplifier provides information on the performance of your code. The Intel Parallel Amplifier shows you the performance issues, enabling you to focus your tuning effort and get the best performance boost in the least amount of time.

The goal of this guide is to introduce you to the basic features of the Amplifier.

After completing this guide, you will be able to use the Amplifier to analyze your code and understand where to focus your tuning efforts to gain the most performance improvement.

This document will step you through the iterative process of tuning a sample application and step you through the stages of performance tuning:

- Locate a performance issue
- Revise the code to remove the issue
- Compare the performance of the new code with the initial code

For a more graphical getting started experience, try the Show Me video demonstrations offered at http://software.intel.com/en-us/articles/intel-parallel-studio.

## Contents

# Disclaimer and Legal Information

# 1 Build the Application

Before you start, you need to build the sample application in the Microsoft* Visual Studio* environment. The matrix application, used as a sample application in this guide, calculates matrix transformations. To facilitate the analysis and quickly estimate your optimization efforts, the application includes a timer and prints the amount of time it takes to calculate matrix transformations.

To build the application:

1. From the Visual Studio, go to **File > Open > Project/Solution** and navigate to `<install_dir>\samples\matrix\matrix.vcproj`.
   The project is added to Visual Studio and shows up in the Solution Explorer.
2. Go to **Build > Build Solution**.
   The matrix.exe application is built.

# 2 Where is Your Program Spending Time?

After building the application, you can go through the process of analyzing the performance of the code provided in the `<install-dir>/samples/matrix` directory. The Intel® Parallel Amplifier provides several types of analysis to collect different types of performance data. In this step, you will run the Hotspot analysis to collect data, view the results, and zoom into the specific problem areas of the source code. The Hotspot analysis helps you understand where your application spends time running and identify the most time-consuming functions.

***NOTE:*** To make sure the performance of the application is repeatable, go through the entire tuning process on one particular system with a minimal amount of other software executing.

## 2.1 Create a Benchmark

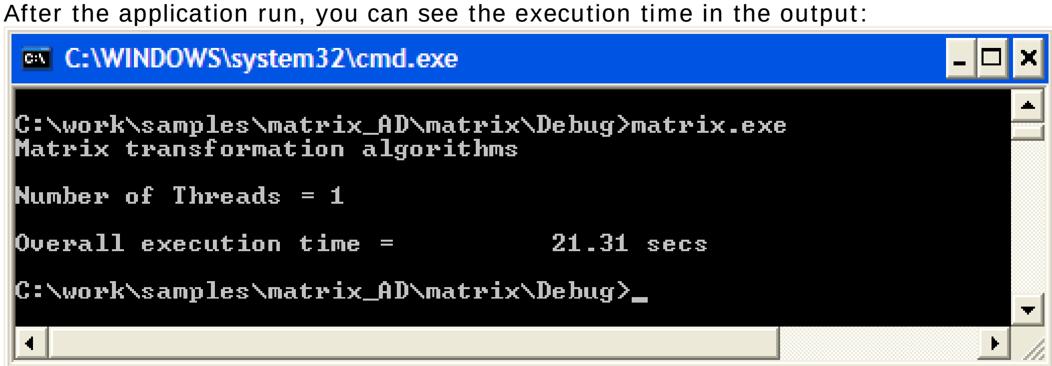
Create a *benchmark* of the original performance:

1. Start the matrix application outside of Visual Studio to get the most accurate numbers.

***NOTE:*** Before running the application to analyze, you are recommended to minimize the number of other software running on your computer to get more accurate results.

A benchmark must be measurable and reproducible so that it can be used as a basis for comparison of future revisions.

2. After the application run, you can see the execution time in the output:



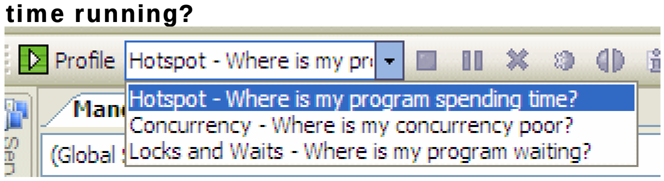

The execution time is your benchmark for this phase of tuning the application.

***NOTE:*** You can run the application several times and use the average number. This helps to minimize transient system activity skewing.

## 2.2 Find a Hotspot

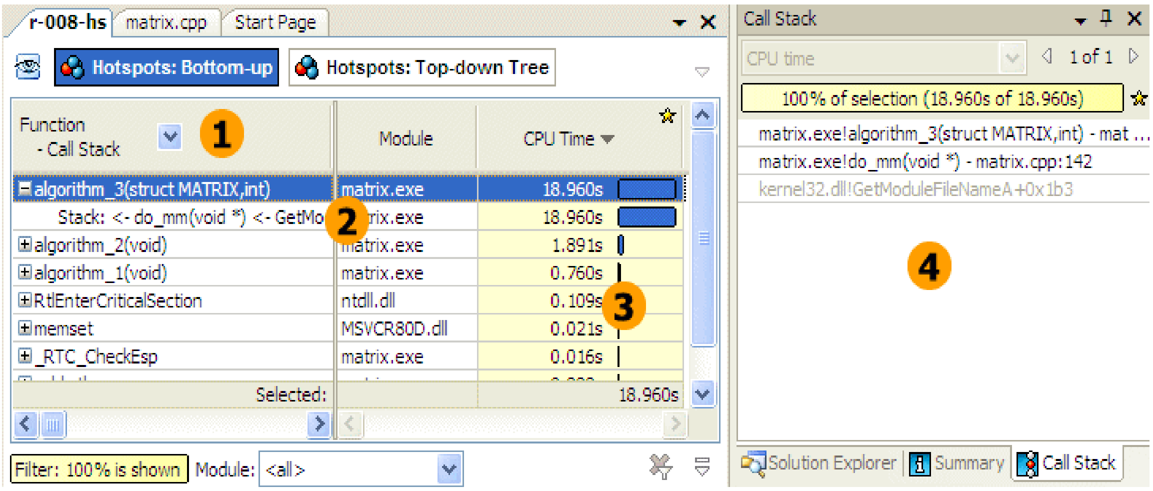
Run the Hotspot analysis to identify the *hotspots* - functions/code sections that took much time to execute.

1. From the Amplifier toolbar, select **Hotspots – Where is my program spending time running?**

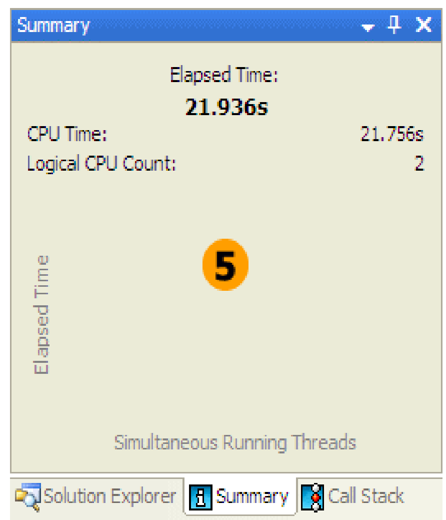



2. Click the **Profile** button.
   The Amplifier launches the matrix application that calculates matrix transformations and exits.

When data collection completes, the **Hotspots: Bottom-up** window opens:



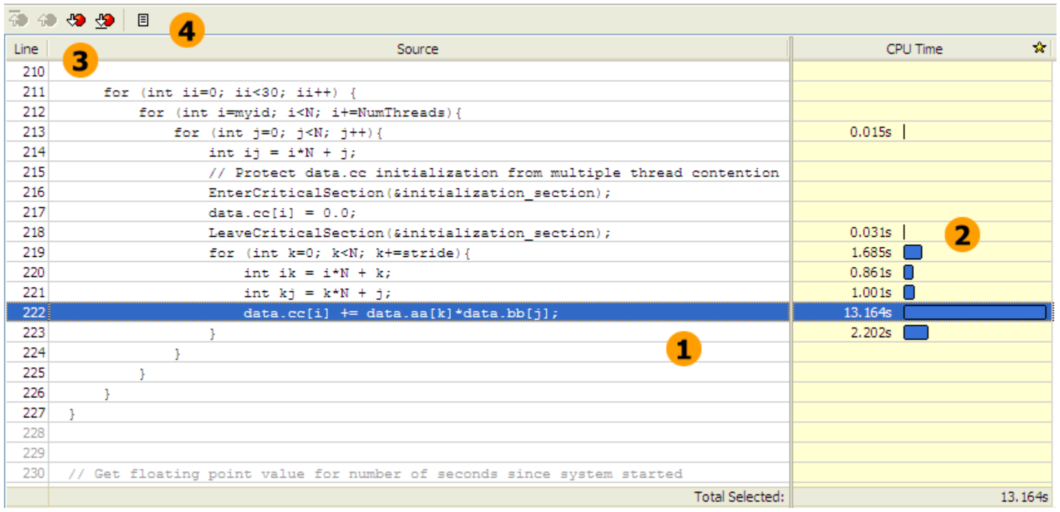

| | |
|---|---|
| 1 | **Function – Call Stack** is the default grouping level for hotspot data. Click the arrow button to change the grouping level. |
| 2 | Click the plus sign in front of the function name to view call stacks for the selected function. Callers of the selected function are displayed, then callers of the first caller(s), and so on. |
| 3 | **CPU Time** is the Data of Interest column for the hotspot analysis results. CPU time is calculated based on running time only. For multiple threads, CPU time is summed up. |
| 4 | Full stack information for the function selected in the grid. Yellow bar shows the contribution of the selected stack to the hotspot function CPU time. |
| 5 | Summary data on the analysis run. CPU Time is the sum of hotspot functions' CPU time. Elapsed time is the application execution time from start to termination. |




# 2.3 Analyze the Results

The first function listed in the **Hotspots: Bottom-up** window and the one that takes the most time, is `algorithm_3`. Focus on this function to see if you can find a way to improve its performance.

Double-click the `algorithm_3` function to view its source code. Notice that line #222 consumed the most CPU time.




This is a good opportunity to get to know some features of the Source pane. The table below explains some of the features available in the Source pane when viewing the Hotspot analysis data.

| | |
|---|---|
| 1 | Non-editable source code of the application. It opens if the function symbol information is available. The code line that took the most CPU time to execute is highlighted. |
| 2 | Processor time is attributed to a particular code line. If the hotspot is a system function, its time, by default, is attributed to the user function that called this system function. |
| 3 | Hotspot navigation buttons to switch between code lines that took a long time to execute. |
| 4 | Source file editor button to open and edit your code. |

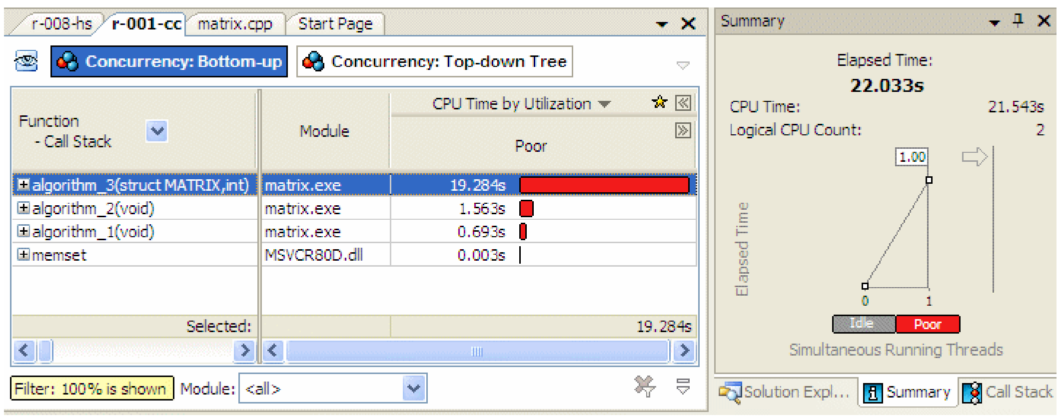
To optimize the sample code, you can consider adding threads to the application so that it could perform well on multicore processors. You have to determine the best place in the application to break up the code into multiple threads.

# 3 Where is Your Concurrency Poor?

In this step, you will run the Concurrency analysis to understand whether your application effectively utilizes all available cores and identify the most serial code to parallelize.

# 3.1 Check for Concurrency

To run the Concurrency analysis, from the Amplifier toolbar select **Concurrency – Where is my concurrency poor?** and click **Profile**. When the matrix application exits after calculations, the Amplifier finalizes the results and opens the **Concurrency** window:




Both the Concurrency window and Summary tab show that the entire matrix application is serial.  The red bars in the CPU Time by Utilization column indicate that processor cores were underutilized. The Summary tab shows only CPU time with 0 or 1 running thread.

Notice that the method with the most serial time is `algorithm_3` as you saw in the **Hotspots: Bottom-up** window. Potentially this module is the best opportunity to parallelize. Double-click `algorithm_3` to see the source code and identify the lines with the most serial time.

# 3.2 Rebuild the Application

In this step, you will rebuild the matrix application for parallelism.

1. From Visual Studio, open `matrix.cpp`.

2. At line 22, uncomment the macro that defines `USE_MULTIPLE_THREADS` as `TRUE`.

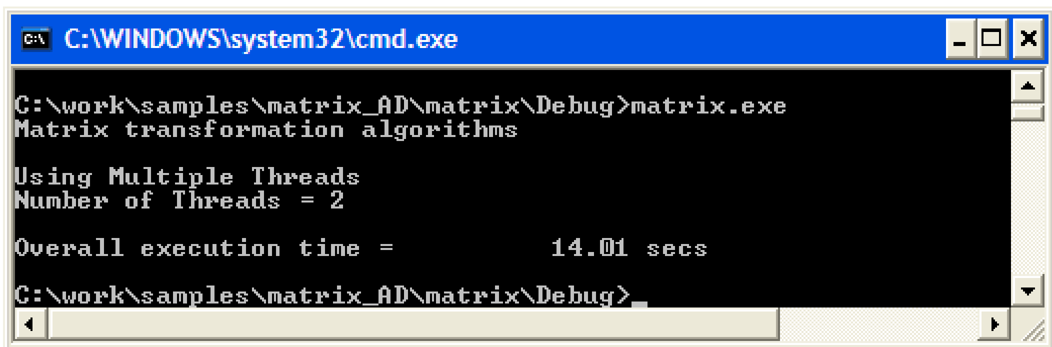
3. At line 23, comment out the macro that defines `USE_MULTIPLE_THREADS` as `FALSE`.

4. In the `algorithm_3` procedure, at lines 216 and 218, uncomment the `Enter` and `LeaveCriticalSection` calls to keep the initialization safe from multiple thread access.

5. Rebuild the application with a debug build.
   Make sure you see 0 errors and 0 warnings in the Visual Studio output pane.

## 3.3 Compare Performance with the Benchmark

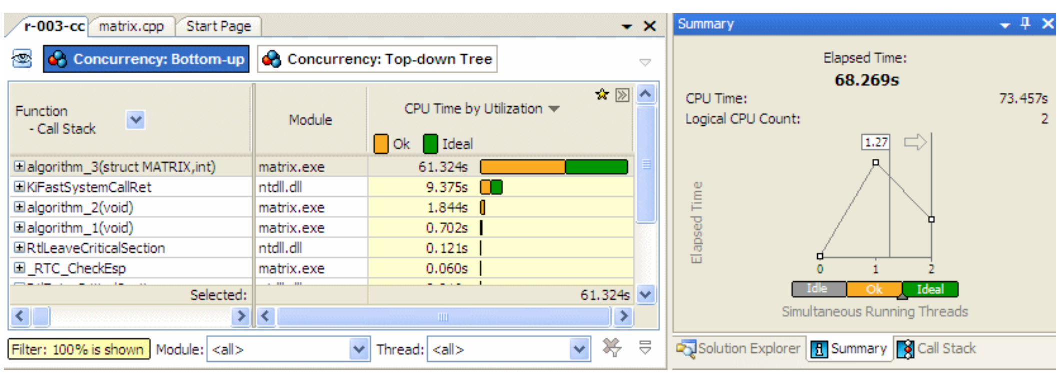
Run the newly built application again from the command window.




Notice that the execution time decreased from 21.31 to 14.01 seconds.

## 3.4 Re-check for Concurrency

In this step, you will run the concurrency data collection again on the modified matrix application.



Notice that `algorithm_3` now does not have under utilized time anymore. But there is still some serial time (Ok type of CPU time utilization indicated with orange bar) that you can try to optimize.

***NOTE:*** The modified version of the application uses cross-thread synchronization primitives. When running the Concurrency or Locks and Waits analysis, the Amplifier analyzes these primitives, which makes the result finalization stage longer and increases the application elapsed time.

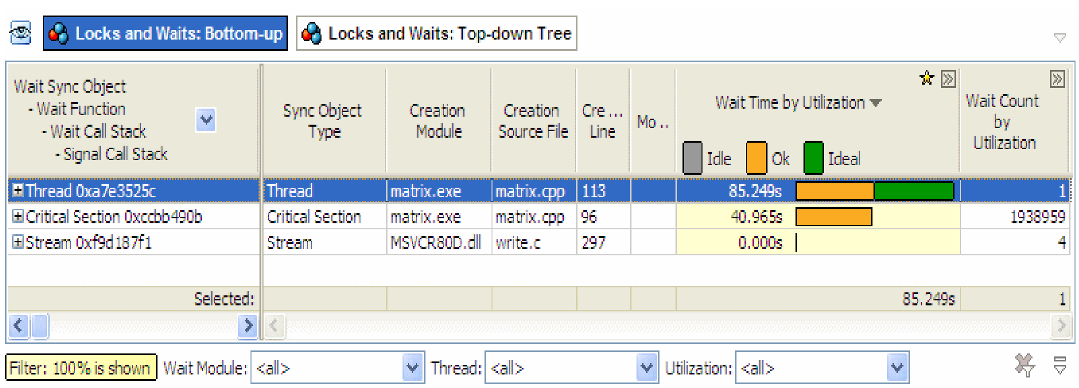
# *4 Where is Your Program Waiting?*

In this step, you will run the Locks and Waits analysis to understand the cause for the serial Ok CPU time left in `algorithm_3`.

## 4.1 Analyze Locks and Waits

To run the Locks and Waits analysis, from the Amplifier toolbar select **Locks and Waits – Where is my application waiting?** and click **Profile**.

The **Locks and Waits: Bottom-up** window provides the following data:




Notice that the synchronization object with the most wait time is a thread. Double-click that thread to go to the source code of the wait.

| Line | Source | Wait Time by Utilization (Idle / Ok / Ideal) | Wait Count by Utilization |
|---|---|---|---|
| 115 | `printf("CreateThread %d failed %d\n",myid[i],GetL` | | |
| 116 | `exit(1);` | | |
| 117 | `}` | | |
| 118 | `}` | | |
| 119 | | | |
| 120 | `// Wait for all "algorithm_3" threads to finish` | | |
| 121 | `int done = WaitForMultipleObjects(NumThreads, h, TRUE, IN` | 85.249s | 85.2487 |
| 122 | | | |
| 123 | `// Display overall execution time` | | |
| 124 | `double overall_end_time = GetSeconds();` | | |
| 125 | `printf("Overall execution time =      %10.2f secs\n",` | | |
| 126 | `overall_end_time - overall_start_time);` | | |
| 127 | | | |
| 128 | `DeleteCriticalSection (&initialization_section);` | | |

You see that it is just the main thread waiting for the matrix transformation thread to complete. This is not a problem because the matrix transformation thread is doing its calculations while the main thread is waiting for it to complete.

Consider the second item in **the Locks and Waits: Bottom-up** window that is more interesting. It is a Critical Section that shows serial only the time while the critical section is causing a wait. Double-click the Critical Section to see the source code for the wait.

| Line | Source | Wait Time by Utilization (Idle / Ok / Ideal) | Wait Count by Utilization |
|---|---|---|---|
| 210 | | | |
| 211 | `for (int ii=0; ii<30; ii++) {` | | |
| 212 | `for (int i=myid; i<N; i+=NumThreads){` | | |
| 213 | `for (int j=0; j<N; j++){` | | |
| 214 | `int ij = i*N + j;` | | |
| 215 | `// Protect data.cc initialization from multip` | | |
| 216 | `EnterCriticalSection(&initialization_section)` | 40.965s | 40.965 |
| 217 | `data.cc[i] = 0.0;` | | |
| 218 | `LeaveCriticalSection(&initialization_section)` | | |
| 219 | `for (int k=0; k<N; k+=stride){` | | |
| 220 | `int ik = i*N + k;` | | |
| 221 | `int kj = k*N + j;` | | |
| 222 | `data.cc[i] += data.aa[k]*data.bb[j];` | | |
| 223 | `}` | | |

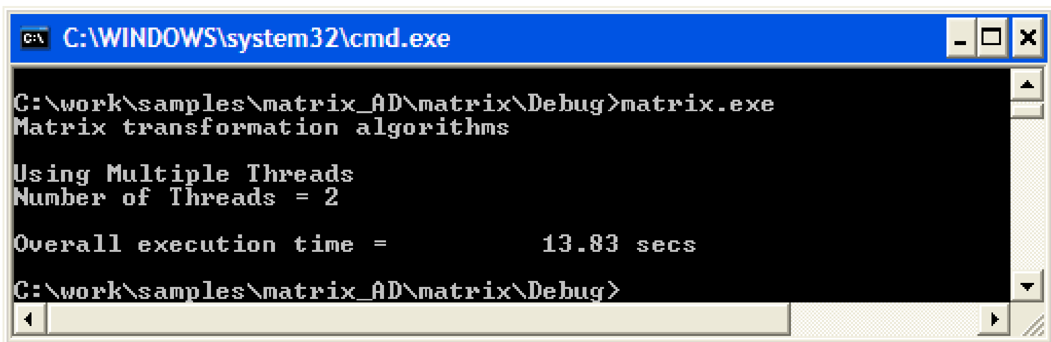
It is the critical section you created when you threaded `algorithm_3`. This critical section is causing some significant wait time. If you examine the code more thoroughly, you can see the critical section is not required. The index `i` used in the `data.cc[i] = 0.0;` statement is already protected from multithread access because of the `for` loop above in which `i` is set. The `for` loop induction variable `i` is set differently for each thread because of the `for` loop iterator `i+=NumThreads`. Thus, you can delete the Critical Section reference and rerun the application.

## 4.2 Rebuild the Final Application

Go back to Visual Studio software and comment out the `EnterCriticalSection` call (line 216) and the `LeaveCriticalSection` call (line 218) and rebuild the app.

## 4.3 Run the Final Benchmark

Run the newly built `matrix.exe` again from the command window.




The execution time of the application decreased from 14.01 to 13.83 seconds.

# 5 *What Optimization Did You Get?*

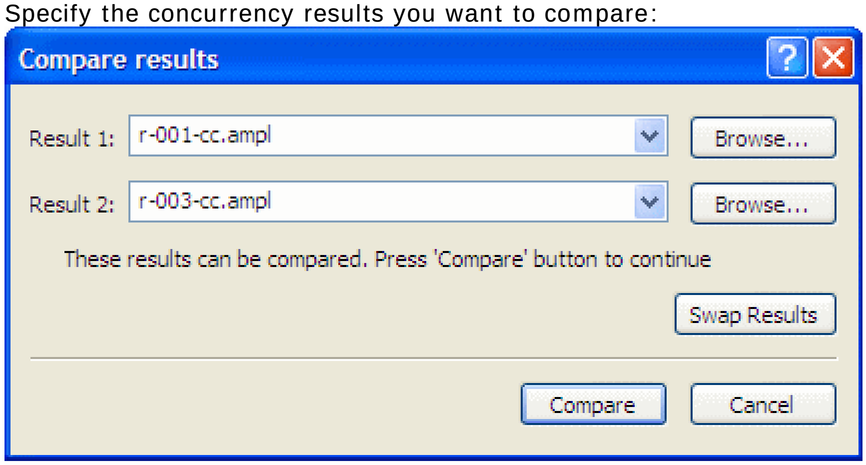
In this step, you will compare concurrency analysis results. You will be able to view performance changes function by function. By comparing the results before and after optimization you made, you can estimate how your changes have changed the performance and how much.

To compare the concurrency results:

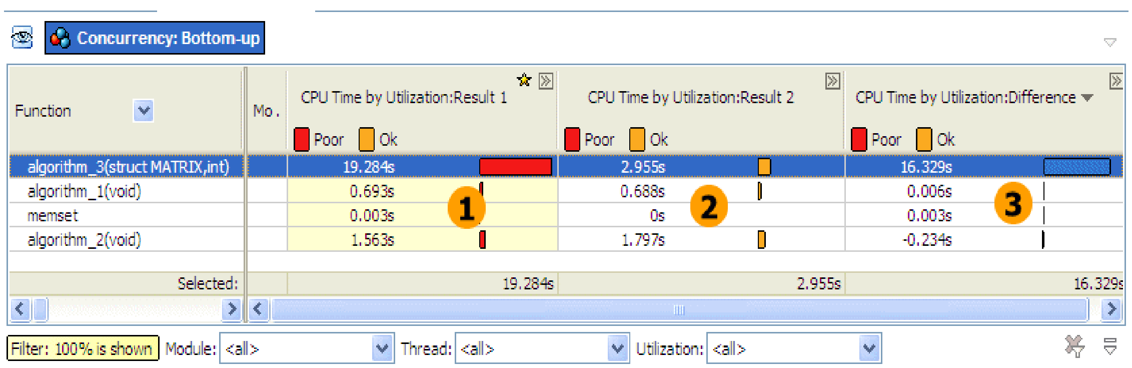
1. Run the Concurrency analysis on the code modified after the Locks and Waits analysis.

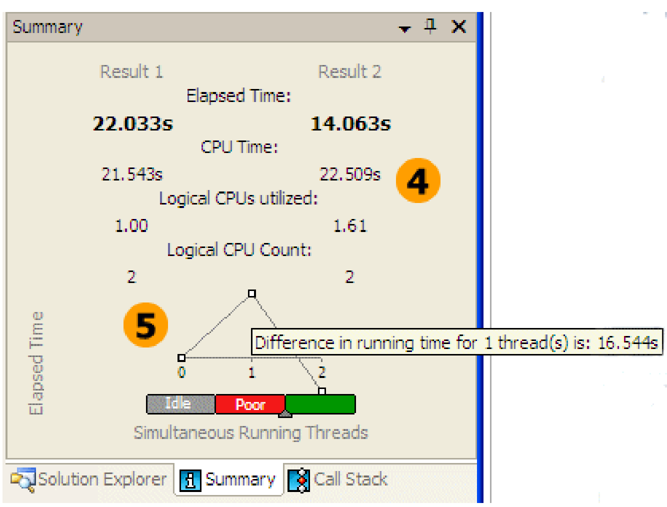
2. Click the **Compare Results**  button on the **Profile** toolbar.
   The **Compare Results** dialog box opens.

3. Specify the concurrency results you want to compare:




The **Concurrency: Bottom-up** window opens:



| | |
|---|---|
| 1 | CPU time for the single-threaded `matrix`.exe application with Poor processor utilization. |
| 2 | CPU time for the optimized multiple-threaded matrix.exe application with Ok processor utilization. |
| 3 | CPU time column providing difference between two results in the following format: <Difference CPU Time> = <Result 1 CPU Time> – <Result 2 CPU Time>. For example, for `algorithm_3`, CPU time optimization for Result 2 is 16 seconds. |
| 4 | Comparison summary provides data for two results: 1) elapsed time is the execution time of the application; 2) CPU time is the sum of CPU time for all threads; 3) logical CPUs utilized is the average utilization of all cores during application run; 4) logical CPU count for your machine. |
| 5 | Concurrency graph displaying the difference in running time between two results <Result 1 running time> – <Result 2 running time>. |

The comparison summary shows that with the multiple threaded version of the `matrix`.exe application (Result 2) you achieved the Ideal processor utilization (86-115% of the target concurrency) when running two threads and got 16-second optimization for the `algorithm_3` hotspot function.

# 6 Next Steps

This guide focuses on basic features of the Intel® Parallel Amplifier. To explore more features and get most of the Intel Parallel Amplifier, try the following resources:

| Resource | Notes |
|---|---|
| *Intel® Parallel Amplifier User's Guide* | Online help integrated into Microsoft* Visual Studio*. User's Guide provides full information on the product.<br><br>To access the user's guide, from the Visual Studio **Help** menu select **Intel Parallel Amplifier > Intel Parallel Amplifier Help**. To view the context-sensitive help for the active window, press **F1**. |
| *Sample Code Guide* | Guide to the sample code located in the `<`*`install-dir`*`>\documentation\<locale>` folder. This guide explores most typical usage scenarios of interpreting and handling the performance bottlenecks.<br><br>To access the Sample Code Guide, from the Visual Studio **Help** menu select **Intel Parallel Amplifier > Sample Code Guide**. |
| *Documentation Index* | Use this html page to locate other Intel® Parallel Amplifier resources. To open this html page, from the |

| | |
|---|---|
| | Windows* Start menu, choose **Intel Parallel Studio > Intel Parallel Amplifier > Intel Parallel Amplifier Documentation**. |
| *Intel® Parallel Studio* resources | Intel® Parallel Studio provides the most comprehensive set of tools for parallelism:<br>• Intel® Parallel Advisor helps developers understand where to add parallelism to existing source code, including identifying hotspots and common data conflicts<br>• Intel® Parallel Composer speeds software development incorporating parallelism with a C/C++ compiler and comprehensive threaded libraries<br>• Intel® Parallel Inspector helps developers detect and perform root-cause analysis on threading and memory errors in multithreaded applications<br>To open documentation that points to more resources for each installed Intel Parallel Studio product, from the Windows* Start menu, choose **Intel Parallel Studio > *product name* > *product name* Documentation**. |