Intel(R) Parallel Amplifier 2011 for Windows* OS

Getting Started Tutorials

Intel(R) Parallel Studio 2011

323355-001US

Legal Information

Legal Information

INFORMATION IN THIS DOCUMENT IS PROVIDED IN CONNECTION WITH INTEL PRODUCTS. NO LICENSE, EXPRESS OR IMPLIED, BY ESTOPPEL OR OTHERWISE, TO ANY INTELLECTUAL PROPERTY RIGHTS IS GRANTED BY THIS DOCUMENT. EXCEPT AS PROVIDED IN INTEL'S TERMS AND CONDITIONS OF SALE FOR SUCH PRODUCTS, INTEL ASSUMES NO LIABILITY WHATSOEVER, AND INTEL DISCLAIMS ANY EXPRESS OR IMPLIED WARRANTY, RELATING TO SALE AND/OR USE OF INTEL PRODUCTS INCLUDING LIABILITY OR WARRANTIES RELATING TO FITNESS FOR A PARTICULAR PURPOSE, MERCHANTABILITY, OR INFRINGEMENT OF ANY PATENT, COPYRIGHT OR OTHER INTELLECTUAL PROPERTY RIGHT.

UNLESS OTHERWISE AGREED IN WRITING BY INTEL, THE INTEL PRODUCTS ARE NOT DESIGNED NOR INTENDED FOR ANY APPLICATION IN WHICH THE FAILURE OF THE INTEL PRODUCT COULD CREATE A SITUATION WHERE PERSONAL INJURY OR DEATH MAY OCCUR.

Intel may make changes to specifications and product descriptions at any time, without notice. Designers must not rely on the absence or characteristics of any features or instructions marked "reserved" or "undefined." Intel reserves these for future definition and shall have no responsibility whatsoever for conflicts or incompatibilities arising from future changes to them. The information here is subject to change without notice. Do not finalize a design with this information.

The products described in this document may contain design defects or errors known as errata which may cause the product to deviate from published specifications. Current characterized errata are available on request.

Contact your local Intel sales office or your distributor to obtain the latest specifications and before placing your product order.

Copies of documents which have an order number and are referenced in this document, or other Intel literature, may be obtained by calling 1-800-548-4725, or go to: http://www.intel.com/design/literature.htm

Intel processor numbers are not a measure of performance. Processor numbers differentiate features within each processor family, not across different processor families. Go to http://www.intel.com/products/processor%5Fnumber/ for details.

BunnyPeople, Celeron, Celeron Inside, Centrino, Centrino Inside, Core Inside, i960, Intel, the Intel logo, Intel AppUp, Intel Atom, Intel Atom Inside, Intel Core, Intel Inside, Intel Inside logo, Intel NetBurst, Intel NetMerge, Intel NetStructure, Intel SingleDriver, Intel SpeedStep, Intel Sponsors of Tomorrow., the Intel Sponsors of Tomorrow. logo, Intel StrataFlash, Intel Viiv, Intel vPro, Intel XScale, InTru, the InTru logo, InTru soundmark, Itanium, Itanium Inside, MCS, MMX, Moblin, Pentium, Pentium Inside, skoool, the skoool logo, Sound Mark, The Journey Inside, vPro Inside, VTune, Xeon, and Xeon Inside are trademarks of Intel Corporation in the U.S. and other countries.

* Other names and brands may be claimed as the property of others.

Microsoft, Windows, Visual Studio, Visual C++, and the Windows logo are trademarks, or registered trademarks of Microsoft Corporation in the United States and/or other countries.

Java and all Java based trademarks and logos are trademarks or registered trademarks of Sun Microsystems, Inc. in the U.S. and other countries.

Microsoft product screen shot(s) reprinted with permission from Microsoft Corporation.

Portions Copyright (C) 2001, Hewlett-Packard Development Company, L.P.

Copyright (C) 1996-2010, Intel Corporation. All rights reserved

Contents

Legal Information......iii
Introducing the Intel® Parallel Amplifier 7
Prerequisites......9
Navigation Quick Start......11
Key Terms and Concepts......13

Chapter 1: Intel® Parallel Amplifier Tutorial: Finding Hotspots
Learning Objectives......17
Workflow Steps to Identify and Analyze Hotspots......17
Choose Target......19
Build Target......20
Run Hotspots Analysis......25
Interpret Result Data......26
Analyze Code......28
Tune Algorithms......30
Compare with Previous Result......34
Summary......36

Chapter 2: Intel® Parallel Amplifier Tutorial: Identifying Concurrency
Learning Objectives......39
Workflow Steps to Analyze CPU Utilization......39
Choose Target......41
Build Target......42

Run Concurrency Analysis....47
Interpret Result Data....48
Analyze Code....51
Add Parallelism....53
Compare with Previous Result....56
Summary....59

Chapter 3: Intel® Parallel Amplifier Tutorial: Analyzing Locks and Waits

Learning Objectives....61
Workflow Steps to Identify Locks and Waits....61
Choose Target....62
Build Target....64
Run Locks and Waits Analysis....69
Interpret Result Data....70
Analyze Code....73
Remove Lock....75
Compare with Previous Result....78
Summary....81

Chapter 4: More Resources

Getting Help and Support....83
Product Support....84

Introducing the Intel® Parallel Amplifier

The Intel® Parallel Amplifier, an Intel® Parallel Studio tool, provides information on the performance of your code.

Use the Amplifier to analyze the following types of performance issues in your applications:

- Identify the most time-consuming (hot) functions in your application
- Locate sections of code that do not effectively utilize available processor time
- Determine the best sections of code to optimize for sequential performance and for threaded performance
- Locate synchronization objects that affect the application performance
- Find whether, where, and why your application spends time on input/output operations
- Identify and compare the performance impact of different synchronization methods, different numbers of threads, or different algorithms

Intel Parallel Amplifier Tutorials

These tutorials tell you how to use the Amplifier to analyze the performance of a sample application on a multi-core system, determine if all cores are fully utilized when the application is running, and identify sections of code that can benefit from threading.

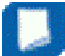
- Finding Hotspots
- Identifying Concurrency
- Analyzing Locks and Waits

Check http://software.intel.com/en-us/articles/intel-software-product-tutorials/ for the following:

- Printable version (PDF) of all Amplifier tutorials
- ShowMe videos of each Amplifier tutorial

Prerequisites

You need the following tools, skills, and knowledge to effectively use these tutorials.



NOTE. The instructions and screen shots in these tutorials refer to the Visual Studio* 2005 integrated development environment (IDE). However, you can use these instructions for later versions of the Visual Studio* IDE.

Required Tools

You need the following tools to use these tutorials:

- Intel® Parallel Amplifier
- Sample code shipped with the Amplifier
- Amplifier Help

To install the Amplifier, follow the instructions in the Release Notes.

To install and set up Amplifier sample code:

1. Copy the `tachyon_studio.zip` file from the `Samples\<locale>` folder in the Intel® Parallel Studio installation folder (the default installation folder is `C:\Program Files\Intel\Parallel Studio 2011`) to a writable directory or share on your system.
2. Extract the sample(s) from the `.zip` file.

NOTE.

- Samples are non-deterministic. Your screens may vary from the screen shots shown throughout these tutorials.
- Samples are designed only to illustrate Amplifier features and do not represent best practices for tuning the code. Results may vary depending on the nature of the analysis.

To access Amplifier Help:

See the Getting Help topic.

Required Skills and Knowledge

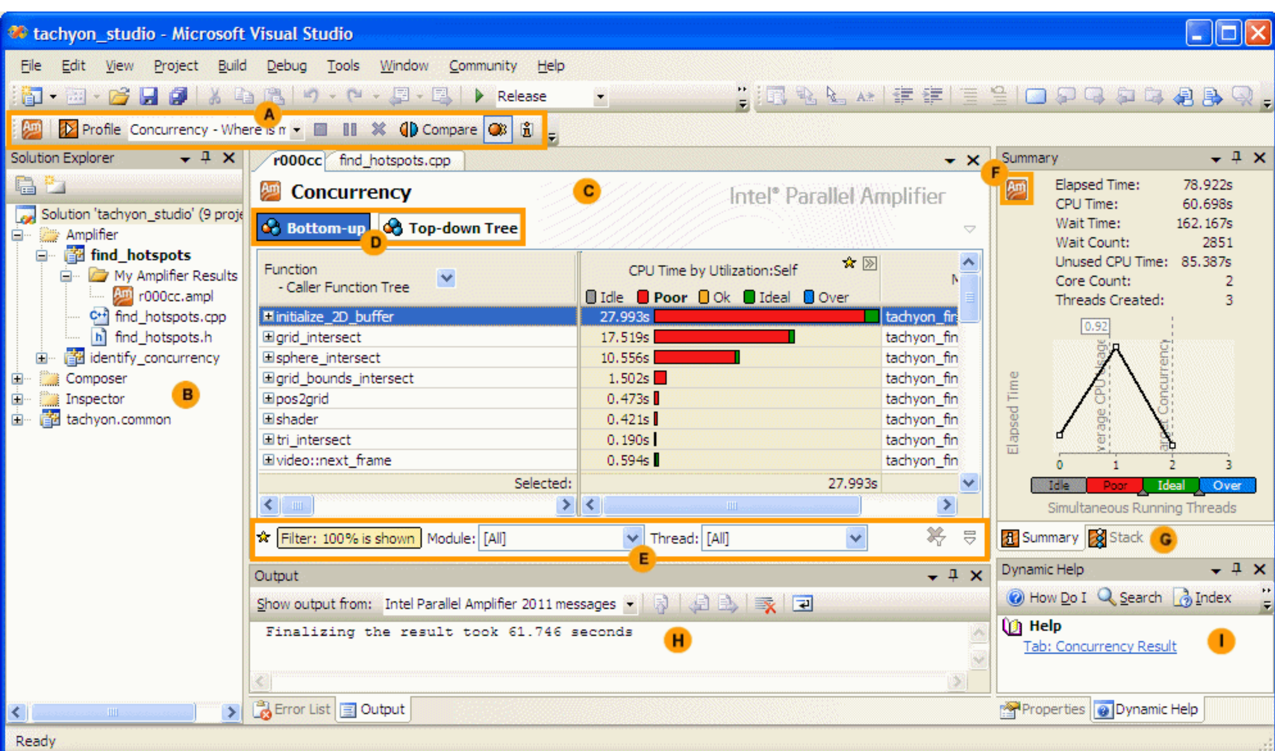
These tutorials are designed for developers with the following skills and knowledge:

- Basic understanding of the Microsoft Visual Studio* 2005 development environment (IDE), including how to:
    - Open a project/solution.
    - Display the **Solution Explorer** and **Output** windows.
    - Compile and link a project.
    - Ensure a project compiled successfully.
    - Access the **Document Explorer** window.

Navigation Quick Start

Intel® Parallel Amplifier/Microsoft Visual Studio* 2005 Integration

The Amplifier integrates into the Visual Studio* development environment (IDE) and can be accessed from the menus, toolbar, and **Solution Explorer** in the following manner:



	Use the **Amplifier** toolbar to configure and control result collection.

B	Amplifier results `*.ampl` show up in the **Solution Explorer** under the **My Amplifier Results** folder. To configure and control result collection, right-click the project in the **Solution Explorer** and select the **Intel Parallel Amplifier 2011** menu from the pop-up menu. To manage previously collected results, right-click the result (for example, `r000cc.ampl`) and select the required command from the pop-up menu.
C	Use the Amplifier result tabs to manage result data.
D	Click buttons on navigation toolbars to change window views and toggle window panes on and off.
E	Use the filter toolbar to filter out the result data according to the selected categories.
F	Click icons to display help snippets that briefly describe window purpose and provide links to more help information. (You can also press the F1 key to display help pages that more completely describe window purpose.)
G	Use the Summary and Call Stack tabs to view basic metrics on the overall application performance and call paths for a function selected in the result tab.
H	Use the **Output** window to view target execution and analysis output.
I	In Microsoft Visual Studio* 2005/2008, use the **Dynamic Help** window to access help topics related to the current Amplifier window.

Key Terms and Concepts

Key Terms

baseline: A performance metric used as a basis for comparison of the application versions before and after optimization. Baseline should be measurable and reproducible.

CPU time: The amount of time a thread spends executing on a logical processor. For multiple threads, the CPU time of the threads is summed. The application CPU time is the sum of the CPU time of all the threads that run the application.

Elapsed time:The total time your target ran, calculated as follows: **Wall clock time at end of application – Wall clock time at start of application**.

hotspot: A section of code that took a long time to execute. Some hotspots may indicate bottlenecks and can be removed, while other hotspots inevitably take a long time to execute due to their nature.

target: A *target* is an executable file you analyze using the Intel® Parallel Amplifier.

Wait time: The amount of time that a given thread waited for some event to occur, such as: synchronization waits and I/O waits.

Key Concept: Concurrency Analysis

The Concurrency analysis helps identify hotspot functions where processor utilization is poor. When cores are idle at a hotspot, you have an opportunity to improve performance by getting those cores working for you.

The Concurrency analysis provides information on how many threads were running at each moment during application execution. It includes threads which are currently running or ready to run and therefore are not waiting at a defined waiting or blocking API. The Amplifier also shows CPU time while the hotspot was executing. A red bar indicates time where the processors are poorly utilized giving you exactly the information you need to decide where you should tune.

Key Concept: CPU Utilization

For the Concurrency and the Locks and Waits analyses, the Intel® Parallel Amplifier identifies a processor utilization scale, calculates the target concurrency, and defines default utilization ranges depending on the number of processor cores. You can change the utilization ranges by dragging the slider in the Summary tab.

Utilization Type	Default color	Description
Idle		All threads in the program are waiting - no threads are running. There can be only one node in the **Summary** tab graph indicating idle utilization.
Poor		Poor utilization. By default, poor utilization is when the number of threads is up to 50% of the target concurrency.
OK		Acceptable (OK) utilization. By default, OK utilization is when the number of threads is between 51-85% of the target concurrency.
Ideal		Ideal utilization. By default, ideal utilization is when the number of threads is between 86-115% of the target concurrency.
Over		Over-utilization. By default, over-utilization is when the number of threads is more than 115% of the target concurrency.

Key Concept: Data of Interest

The Amplifier maintains a special column called Data of Interest. This column is highlighted with yellow background and a yellow star in the column header .

The data in the Data of Interest column is used by various windows as follows:

- The Call Stack tab calculates the contribution, shown in the contribution bar, using the Data of Interest column values.
- The Filter bar uses the data of interest values to calculate the percentage indicated in the filtered option.
- The Source/Assembly window uses this column for hotspot navigation.

If a view has more than one column with numeric data or bars, you can change the default **Data of Interest** column by right-clicking the required column and selecting the **Set Column as Data of Interest** command from the pop-up menu.

Key Concept: Finalization

Finalization is a process when the Amplifier converts the collected data to a database, resolves symbol information, and pre-computes data to make further analysis more efficient and responsive. The Amplifier finalizes data automatically when generating results.

You may want to re-finalize a result to:

- update symbol information after changes in the search directories settings
- resolve the number of [Unknown]-s in the results

Key Concept: Hotspots Analysis

The Hotspots analysis helps understand the application flow and identify sections of code that took a long time to execute (*hotspots*). A large number of samples collected at a specific process, thread, or module can imply high processor utilization and potential performance bottlenecks. Some hotspots can be removed, while other hotspots are fundamental to the application functionality and cannot be removed.

The Intel® Parallel Amplifier creates a list of functions in your application ordered by the amount of time spent in a function. It also detects the call stacks for each of these functions so you can see how the hot functions are called.

The Amplifier uses a low overhead (about 5%) statistical sampling algorithm that gets you the information you need without a significant slowing of application execution.

Key Concept: Locks and Waits Analysis

While the Concurrency analysis helps identify where your application is not parallel, the Locks and Waits analysis helps identify the cause of the ineffective processor utilization. One of the most common problems is threads waiting too long on synchronization objects (locks). Performance suffers when waits occur while cores are under-utilized.

During the Locks and Waits analysis you can estimate the impact each synchronization object introduces to the application and understand how long the application was required to wait on each synchronization object, or in blocking APIs, such as sleep and blocking I/O.

Intel® Parallel Amplifier Tutorial: Finding Hotspots

1

Learning Objectives

This tutorial shows how to use the Hotspots analysis of the Intel® Parallel Amplifier to understand where the sample application is spending time, identify *hotspots* - the most time-consuming program units, and detect how they were called. Some hotspots may indicate bottlenecks that can be removed, while other hotspots are inevitable and take a long time to execute due to their nature. Typically, the hotspot functions identified during the Hotspots analysis use the most time-consuming algorithms and are good candidates for parallelization.

The Hotspots analysis is useful to analyze the performance of both serial and parallel applications.

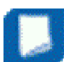
Estimated completion time: 15 minutes.

After you complete this tutorial, you should be able to:

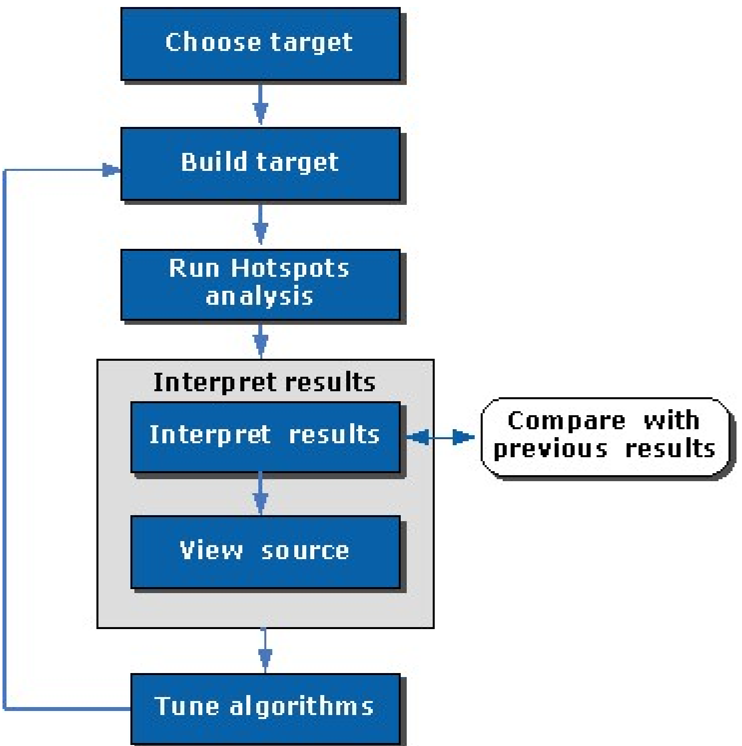
- Choose an analysis target.
- Choose the Hotspots analysis type.
- Run the Hotspots analysis to locate most time-consuming functions in an application.
- Analyze the function call flow and threads.
- Analyze the source code to locate the most time-critical code lines.
- Compare results before and after optimization.

Workflow Steps to Identify and Analyze Hotspots

You can use the Intel® Parallel Amplifier to identify and analyze hotspot functions in your serial or parallel application by performing a series of steps in a workflow. This tutorial guides you through these workflow steps while using a sample ray-tracer application named `tachyon`.



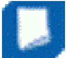
NOTE. Click Show Me for a video demonstration. Show Me videos require Adobe* Flash* Player.



1. Choose a target to analyze for hotspots.
2. Configure environment and project settings and build your target.
3. Choose and run the Hotspots analysis.
4. Interpret the result data.
5. View and analyze code of the performance-critical function.
6. Modify the code to tune the algorithms or rebuild the code with Intel® Compiler.
7. Re-build the target, re-run the Hotspots analysis, and compare the result data before and after optimization.

Choose Target

Before you start you need to choose the target in the Microsoft Visual Studio* environment. For this tutorial, your target is a ray-tracer application, `tachyon`. To learn how to install and set up the sample code, see Prerequisites.



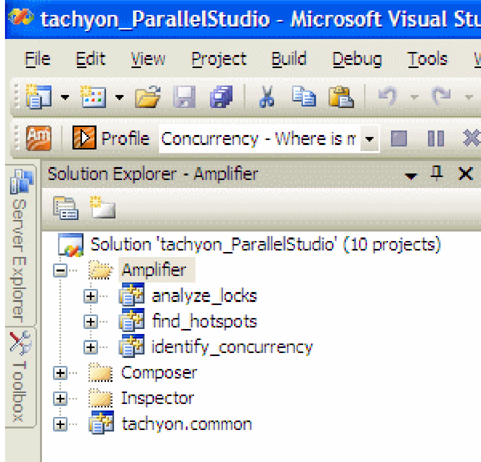
NOTE. The steps below are provided for Microsoft Visual Studio 2005. They may slightly differ for other versions of Visual Studio.

To choose a target:

1. From the Visual Studio menu, select **File > Open > Project/Solution...**.

   The **Open Project** dialog box opens.

2. In the **Open Project** dialog box, browse to the location you used to unzip the `tachyon_studio.zip` file and select the `tachyon_ParallelStudio.sln` file.

   The solution is added to Visual Studio IDE and shows up in the Solution Explorer.

3. In the Solution Explorer, right-click the **find_hotspots** project and select **Project > Set as StartUp Project**.

   find_hotspots appears in bold in the Solution Explorer.

Recap

You selected the **find_hotspots** project as the target for the Hotspots analysis.

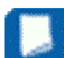
Key Terms and Concepts

- Term: target
- Concept: Hotspots Analysis

Build Target

After choosing the analysis target, do the following to ensure the Intel® Parallel Amplifier provides the most accurate information on the performance of your application:

- Build the target in the release mode with full optimizations, which is recommended for performance analysis.
- Run the application without debugging to create a performance baseline.
- Configure the Microsoft Visual Studio* environment to download the debug information for system libraries so that Amplifier can properly identify system functions and classify and attribute functions.
- Configure Visual Studio project properties to generate the debug information for your application so that Amplifier can open the source code.



NOTE. The steps below are provided for Microsoft Visual Studio 2005. They may slightly differ for other versions of Visual Studio.

Choose a Build Mode and Build a Target

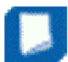
1. Go to the **Build > Configuration Manager...** dialog box and select the **Release** mode for your target project.
2. From the Visual Studio menu, select **Build > Build find_hotspots**.

   The `tachyon_find_hotspots.exe` application is built.

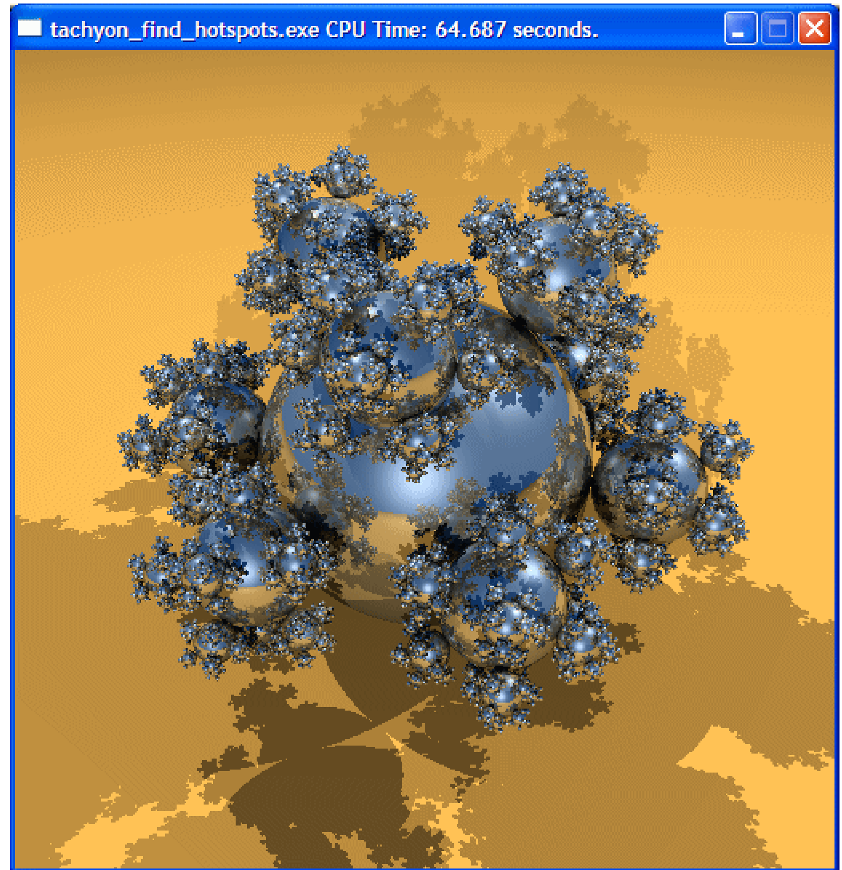
Create a Performance Baseline

1. From the Visual Studio menu, select **Debug > Start Without Debugging**.

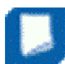
   The `tachyon_find_hotspots` application starts running.



NOTE. Before you start the application, minimize the amount of other software running on your computer to get more accurate results.



2. Note the execution time displayed in the window caption. For the `tachyon_find_hotspots` executable in the figure above, the execution time is 64.687 seconds. The total execution time is the baseline against which you will compare subsequent runs of the application.



NOTE. Run the application several times, note the execution time for each run, and use the average number. This helps to minimize skewed results due to transient system activity.

Enable Downloading the Debug Information for System Libraries

1. Go to **Tools > Options...**.

   The **Options** dialog box opens.
2. From the left pane, select **Debugging > Symbols**.
3. In the **Symbol file (.pdb) locations** field, click the button and specify the following address: http://msdl.microsoft.com/download/symbols.
4. Make sure the added address is checked.
5. In the **Cache symbols from symbol servers to this directory** field, specify a directory where the downloaded symbol files will be stored.
6. For Microsoft Visual Studio* 2005, check the **Load symbols using the updated settings when this dialog is closed** box.

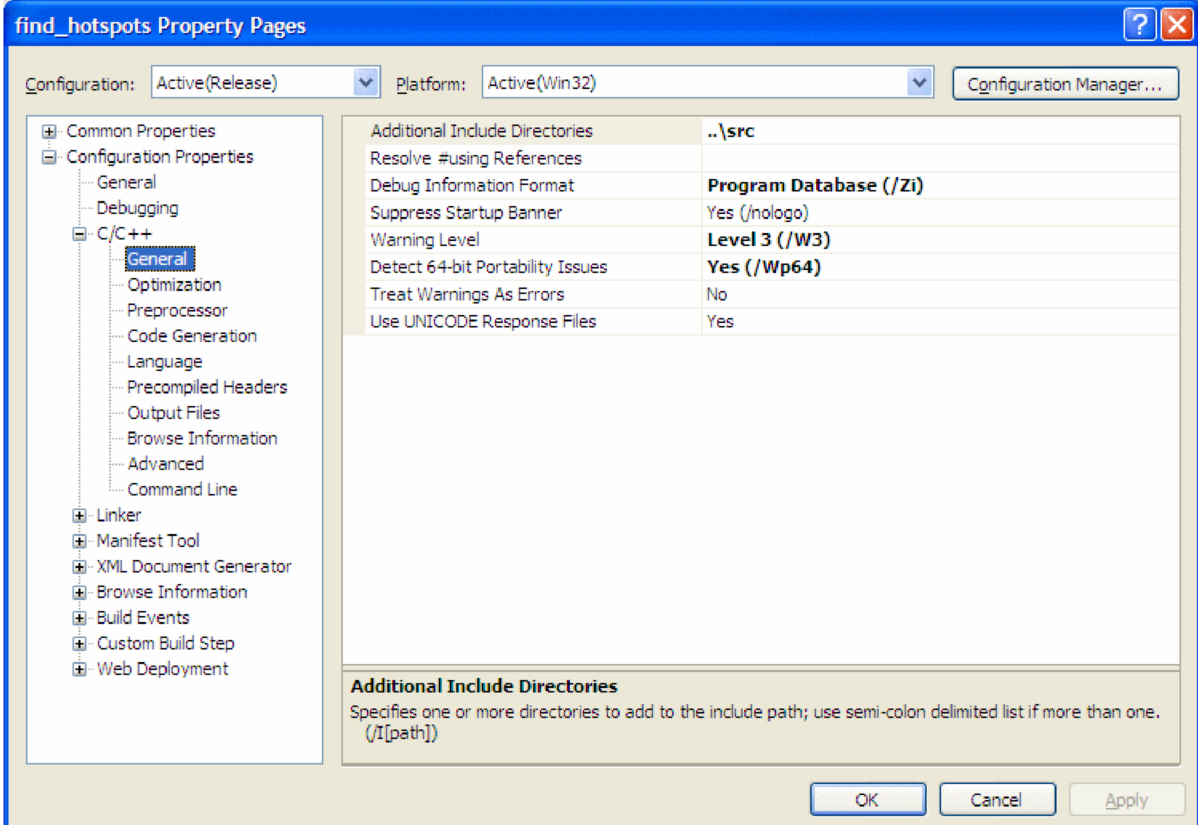
7. Click **OK**.

Enable Generating Debug Information for Your Binary Files

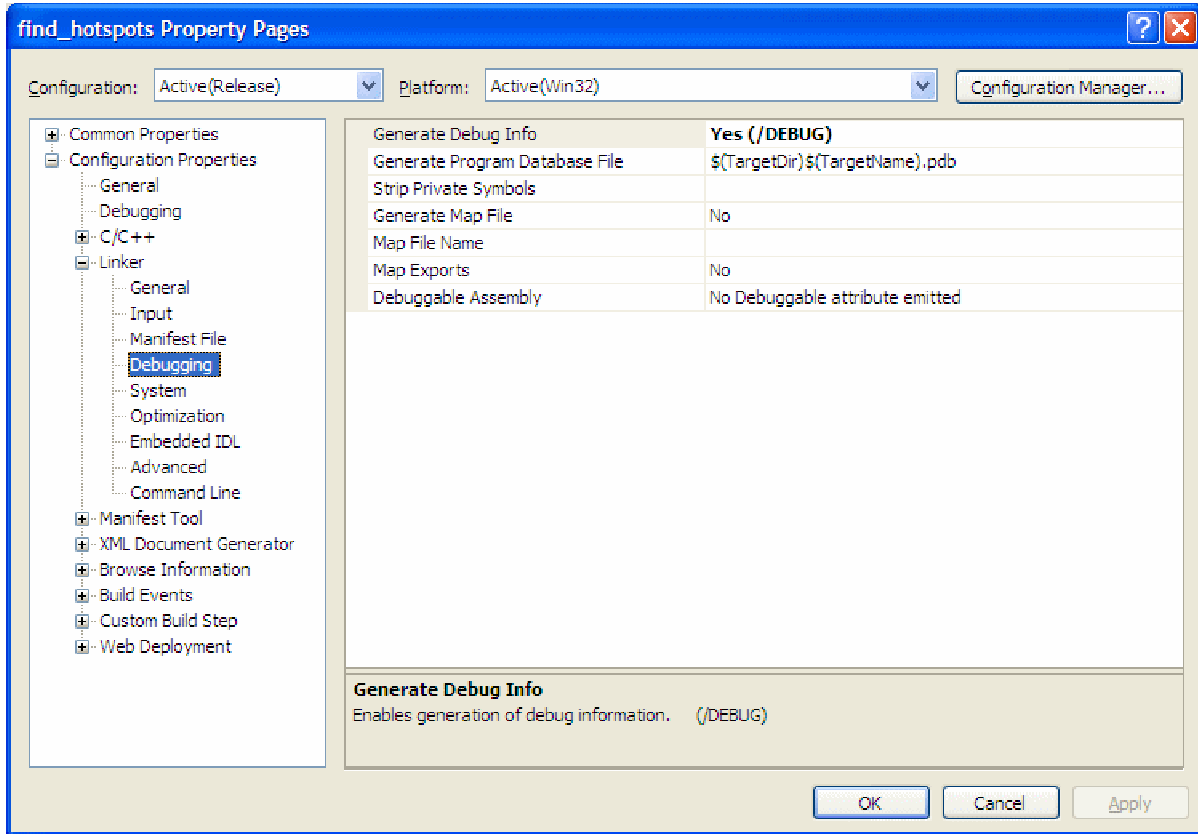
1. Select the **find_hotspots** project and go to **Project > Properties**.

2. From the **find_hotspots Property Pages** dialog box, select **Configuration Properties > General** and make sure the selected **Configuration** (top of the dialog) is **Active(Release)**.

3. From the **find_hotspots Property Pages** dialog box, select **C/C++ > General** pane and specify the **Debug Information Format** as **Program Database (/Zi)**.



4. From the **find_hotspots Property Pages** dialog box, select **Linker > Debugging** and set the **Generate Debug Info** option to **Yes (/DEBUG)**.



Recap

You chose the target for the Hotspots analysis, set up your environment to enable generating symbol information for system libraries and your binary files, built the target in the Release mode, and created the performance baseline. Your application is ready for analysis.

Key Terms and Concepts

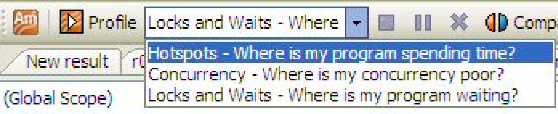
- Term: target, baseline

Run Hotspots Analysis

After building the application, you can run it to analyze its performance. In this tutorial, you run the Hotspots analysis to identify the hotspots that took much time to execute.

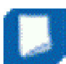
To run an analysis:

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



2. Click the **Profile** button.

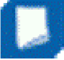
Amplifier launches the `tachyon_find_hotspots` application that renders `balls.dat` as an input file, calculates the execution time, and exits. Amplifier finalizes the collected results and opens the Hotspots result tab.



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

Recap

You launched the Hotspots data collection that analyzes function calls and CPU time spent in each program unit of your application.



NOTE. This tutorial explains how to run an analysis from the Amplifier graphical user interface (GUI). You can also use the Amplifier command-line interface (`ampl-cl` command) to run an analysis. For more details, check the *Command-line Interface Support* section of the Amplifier Help.

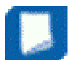
Key Terms and Concepts

- Term: hotspot, Elapsed time
- Concept: Hotspot Analysis, Finalization

Interpret Result Data

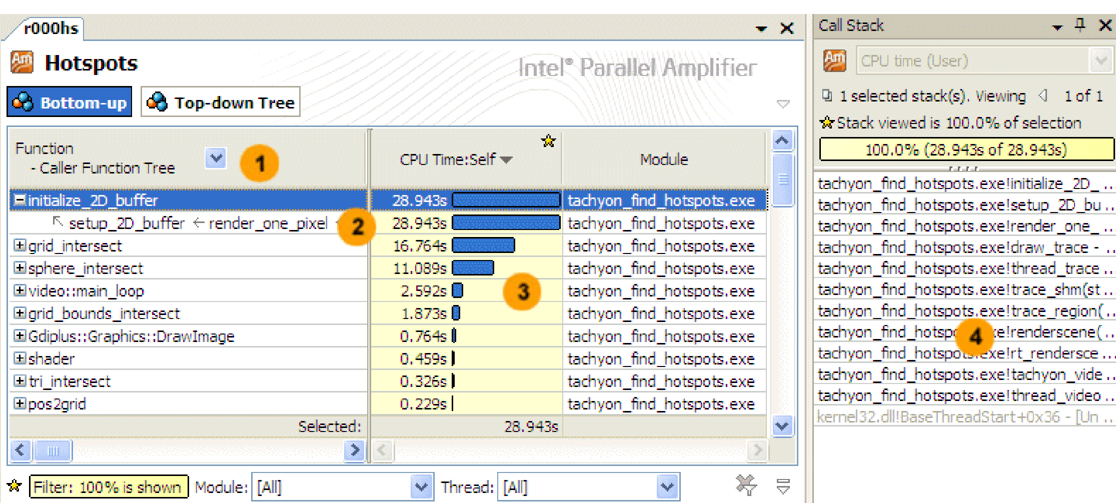
When the sample application exits, the Bottom-up window displays functions ordered bottom up: a child function is placed directly above its parent. To interpret the data on the sample code performance, do the following:

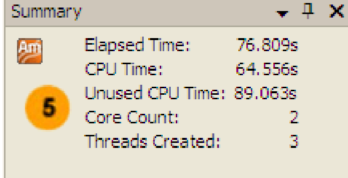
- Understand the basic performance metrics provided by the Hotspots analysis.
- Analyze the most time-consuming functions.



NOTE. The screen shots and execution time data provided in this tutorial are created on a system with two CPU cores. Your data may vary depending on the number and type of CPU cores on your system.

Understand the Basic Hotspots Metrics





1 **Function – Bottom-up Tree** 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. Double-clicking a stack opens the body of the first caller function in the Source window.

3 CPU time is the active time taken to execute a function. For multiple threads, CPU time is summed up. By default, this column is marked with a yellow star as the Data of Interest column for the Hotspots Bottom-up window. This means that the data of this column is used for different types of calculations, for example: call stack contribution, percentage value on the filter toolbar.

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. Double-clicking a stack function opens the body of this function in the Source window. The code line calling the next function in the stack is highlighted.

5 Summary data on the analysis run: 1) **Elapsed Time** is the execution time of the application from start to termination; 2) **CPU Time** is the sum of CPU time for all threads; 3) **Unused CPU Time** is the total time for each core when it was either waiting or not utilized by the application; 4) **Core Count** is the logical CPU count for your machine; 5) **Threads Created** by your system during the application run.

Analyze the Most Time-consuming Functions

Analyze the **CPU Time:Self** column values in the Bottom-up window. By default, all data in the grid is sorted by this column. Functions that took most CPU time to execute are listed on top. The first function `initialize_2D_buffer` took 28.943 seconds to execute. Click the plus sign ⊞ at the `initialize_2D_buffer` function to expand the stacks calling this function. You see that it was called only once by the `setup_2D_buffer` function. You can analyze the Call Stack tab to find more detailed information on the function sequence calling the `initialize_2D_buffer` function.

The `initialize_2D_buffer` function is the biggest hotspot. Focus on this function to see if you can find a way to improve its performance.

Recap

You identified a function that took the most CPU time and could be a good candidate for algorithm tuning.

Key Terms and Concepts

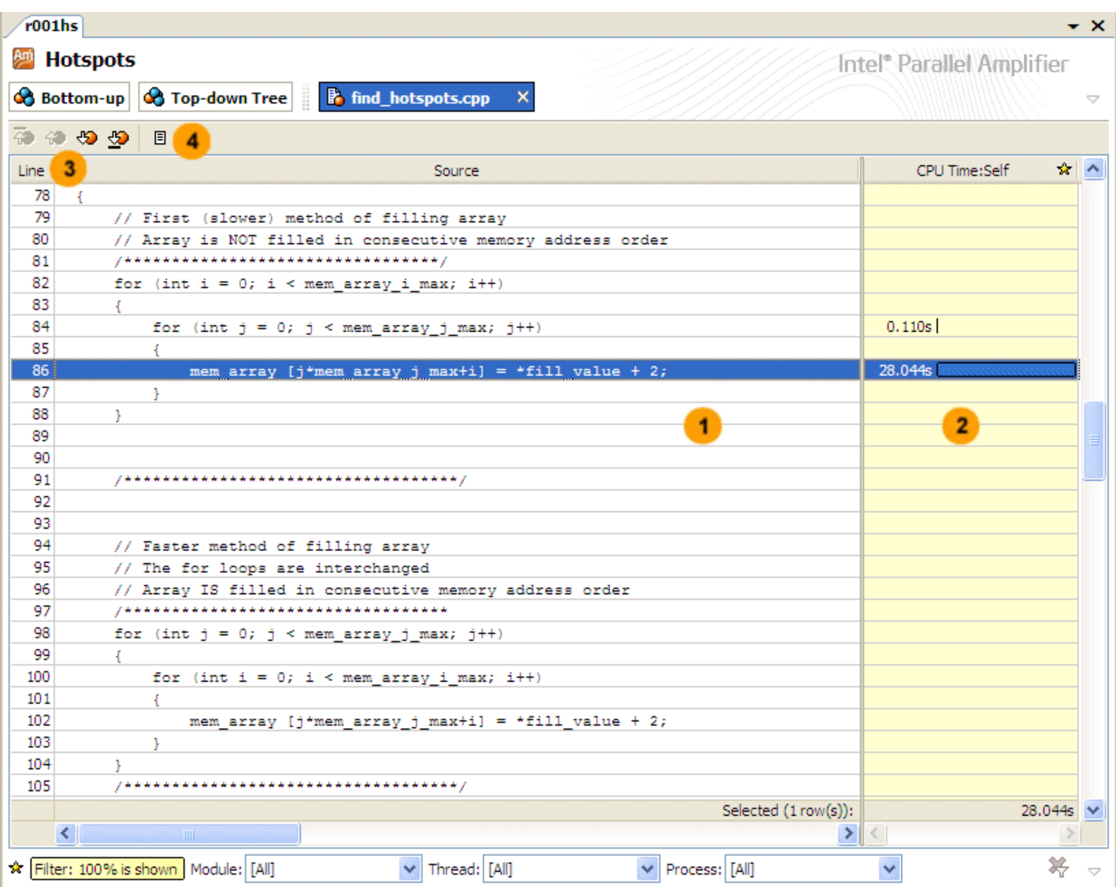
- Term: Elapsed time, CPU time
- Concept: Hotspots Analysis, Data of Interest

Analyze Code

In the Bottom-up window, you identified `initialize_2D_buffer` as the hottest function. In the Bottom-up window, double-click this function to open the Source window and analyze the source code:

- Understand basic options provided in the Source window.
- Identify the hottest code lines.

Understand Basic Source View Options



The table below explains some of the features available in the Source window when viewing the Hotspots analysis data.

Source pane displaying the source code of the application if the function symbol information is available. The code line that took the most CPU time to execute is highlighted. The source code in the Source pane is not editable.

If the function symbol information is not available, the Assembly window opens displaying assembler instructions for the selected hotspot function. To enable the Source view, make sure to build the target properly.

2 Processor time 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 most performance-critical code lines. Hotspot navigation is based on the metric column selected as a Data of Interest. For the Hotspots analysis, this is **CPU Time: Self**.

4 Source file editor button to open and edit your code.

Identify the Hottest Code Lines

When you identify a hotspot in the serial code, you can make some changes in the code to tune the algorithms and speed up that hotspot. Another option is to parallelize the sample code by adding threads to the application so that it performs well on multi-core processors. This tutorial focuses on algorithm tuning.

By default, when you double-click the hotspot in the Bottom-up window, Amplifier opens the source file related to this function highlighting the code line that took the most CPU time. For the `initialize_2D_buffer` function, the hottest code line is 86. This code is used to initialize a memory array using non-sequential memory locations. Click the Source Editor button on the Source window toolbar to open the default code editor and work on optimising the code.

Recap

You identified the code section that took the most CPU time to execute.

Key Terms and Concepts

- Term: hotspot, CPU time
- Concept: Hotspots Analysis, Data of Interest

Tune Algorithms

In the **Source** window, you identified that in the `initialize_2D_buffer` hotspot function the code line 86 took the most CPU time. Focus on this line and do the following:

- Open the code editor.
- Resolve the performance problem using any of these options:

- Optimize the algorithm used in this code section.
- Recompile the code with the Intel® Compiler.

Open the Code Editor

Click the Source Editor button to open the `find_hotspots.cpp` file in the default code editor at the hotspot line:

```
77  void initialize_2D_buffer (unsigned int mem_array [], unsigned int *fill_value)
78  {
79      // First (slower) method of filling array
80      // Array is NOT filled in consecutive memory address order
81      /*********************************/
82      for (int i = 0; i < mem_array_i_max; i++)
83      {
84          for (int j = 0; j < mem_array_j_max; j++)
85          {
86              mem_array [j*mem_array_j_max+i] = *fill_value + 2;
87          }
88      }
89
90
91      /***********************************/
92
93
94      // Faster method of filling array
95      // The for loops are interchanged
96      // Array IS filled in consecutive memory address order
97      /***********************************
98      for (int j = 0; j < mem_array_j_max; j++)
99      {
100         for (int i = 0; i < mem_array_i_max; i++)
101         {
102             mem_array [j*mem_array_j_max+i] = *fill_value + 2;
103         }
104     }
105     /***********************************/
106 }
```

The hotspot line is used to initialize a memory array using non-sequential memory locations. For demonstration purposes, the code lines are commented as a slower method of filling the array.

Resolve the Problem

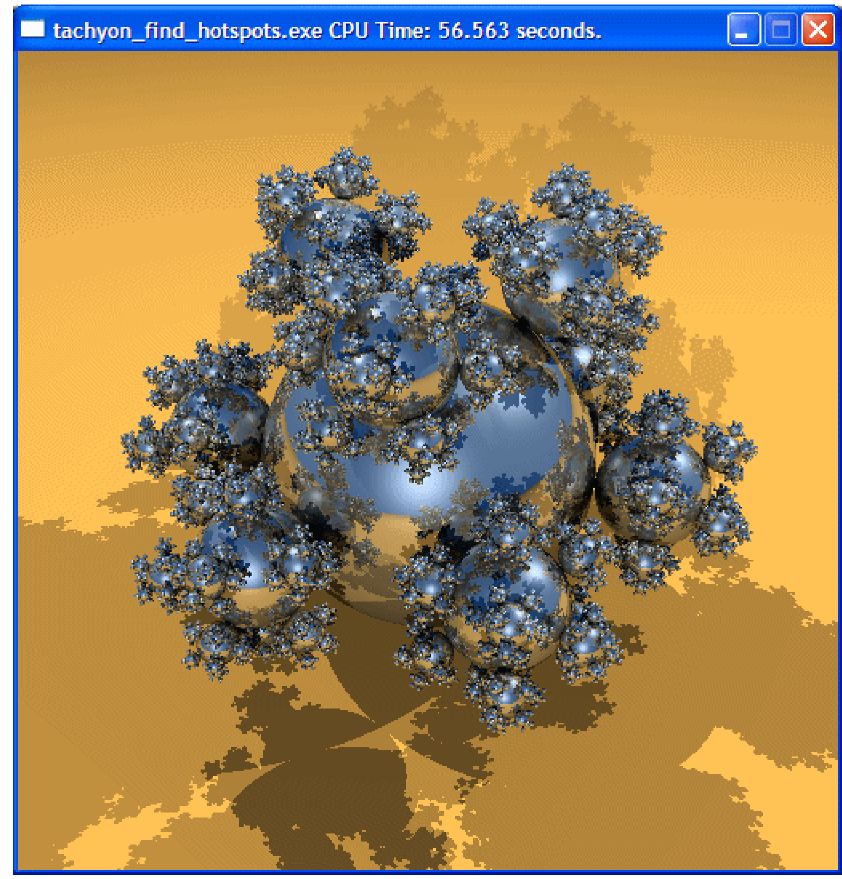
To resolve this issue, use one of the following methods:

Option 1: Optimize your algorithm

1. Edit line 81 to comment out code lines 82-88 marked as a "First (slower) method".
2. Edit line 97 to uncomment code lines 98-104 marked as a "Faster method".

   In this step, you interchange the `for` loops to initialize the code in sequential memory locations.
3. From the Visual Studio menu, select **Build > Rebuild find_hotspots**.

   The project is rebuilt.
4. From Visual Studio **Debug** menu, select **Start Without Debugging** to run the application.



Visual Studio runs the `tachyon_find_hotspots.exe`. Note that execution time has reduced from 64.687 seconds to 56.563 seconds.

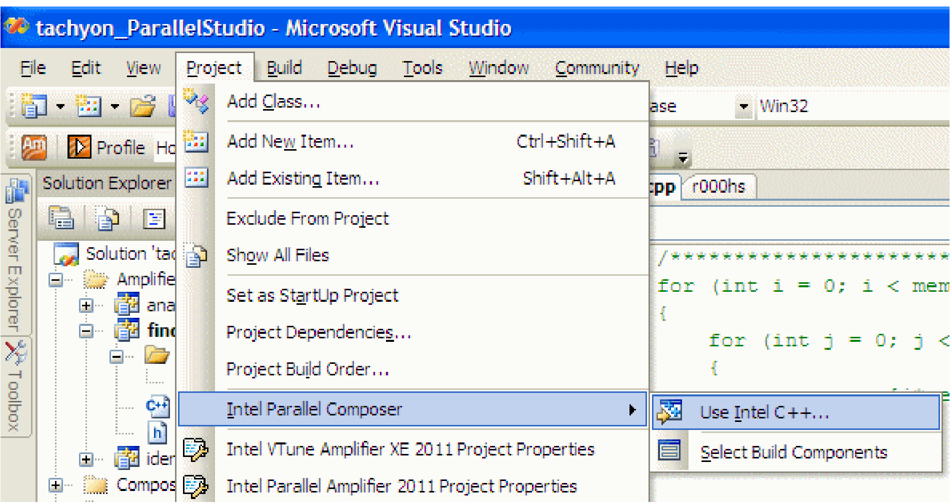
Option 2: Recompile the code with Intel® Compiler

This option assumes that you have Intel® Parallel Composer installed. Composer is part of Intel® Parallel Studio. By default, the Intel® Compiler, one of the Composer's components, uses powerful optimization switches, which typically provides some gain in performance. For more details on the Intel compiler, see the Intel Parallel Composer documentation.

As an alternative, you may consider running the default Microsoft Visual Studio compiler applying more aggressive optimization switches.

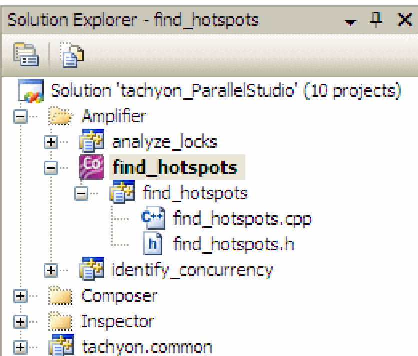
To recompile the code with the Intel compiler:

1. From Visual Studio **Project** menu, select **Intel Parallel Composer > Use Intel C++...**.



2. In the **Confirmation** window, click **OK** to confirm your choice.

   The project in Solution Explorer appears with the Composer icon:



3. From the Visual Studio menu, select **Build > Rebuild find_hotspots**.

   The project is rebuilt with the Intel Compiler.

4. From the Visual Studio menu, select **Debug > Start Without Debugging**.

   Visual Studio runs the `tachyon_find_hotspots.exe`. Note that the execution time reduced from 64.687 to 58.672 seconds.

Recap

You interchanged the loops in the hotspot function, rebuilt the application, and got performance gain of 6-8 seconds. You also considered an alternative optimization technique using the Intel C++ compiler.

Key Terms and Concepts

- Term: hotspot

Compare with Previous Result

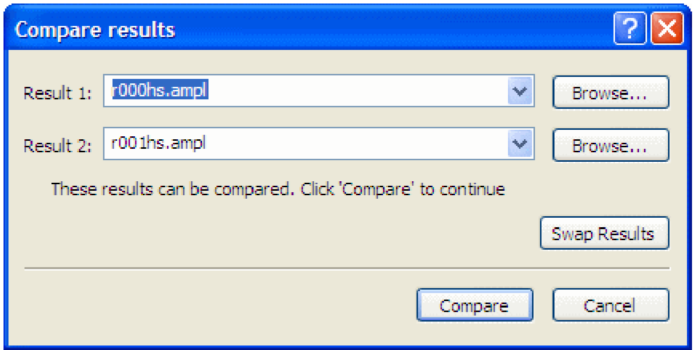
You made sure that the loop interchange mechanism you applied to optimize your code gave you 6-8 seconds of improvement in the application execution time. To understand whether you got rid of the hotspot and what kind of optimization you got per function, re-run the Hotspots analysis on the optimized code and compare results:

- Compare results before and after optimization.
- Identify the performance gain.

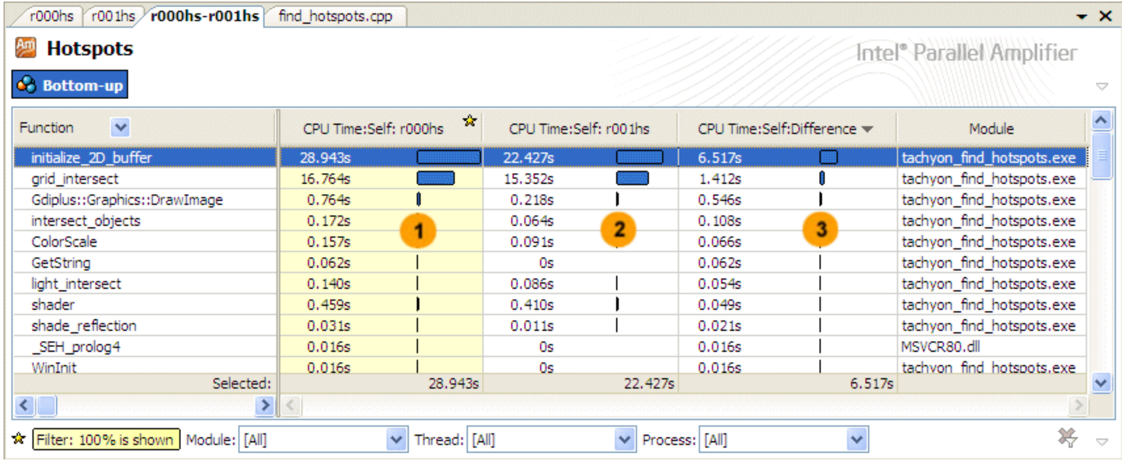
Compare Results Before and After Optimization

1. Run the Hotspots analysis on the modified code.
2. Click the Compare button on the Amplifier toolbar.

   The **Compare Results** dialog box opens.
3. Specify the Hotspots analysis results you want to compare:

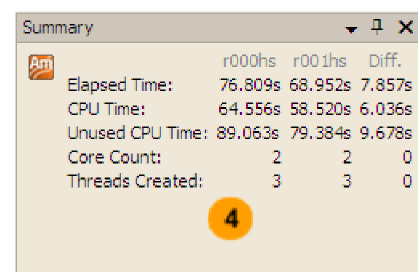


The Hotspots Bottom-up window opens, showing the CPU time usage across the two results and the differences side by side:



Function	CPU Time:Self: r000hs	CPU Time:Self: r001hs	CPU Time:Self:Difference	Module
initialize_2D_buffer	28.943s	22.427s	6.517s	tachyon_find_hotspots.exe
grid_intersect	16.764s	15.352s	1.412s	tachyon_find_hotspots.exe
Gdiplus::Graphics::DrawImage	0.764s	0.218s	0.546s	tachyon_find_hotspots.exe
intersect_objects	0.172s	0.064s	0.108s	tachyon_find_hotspots.exe
ColorScale	0.157s	0.091s	0.066s	tachyon_find_hotspots.exe
GetString	0.062s	0s	0.062s	tachyon_find_hotspots.exe
light_intersect	0.140s	0.086s	0.054s	tachyon_find_hotspots.exe
shader	0.459s	0.410s	0.049s	tachyon_find_hotspots.exe
shade_reflection	0.031s	0.011s	0.021s	tachyon_find_hotspots.exe
_SEH_prolog4	0.016s	0s	0.016s	MSVCR80.dll
WinInit	0.016s	0s	0.016s	tachyon_find_hotspots.exe
Selected:	28.943s	22.427s	6.517s	

1. CPU time for the initial version of the `tachyon_find_hotspots.exe` application.
2. CPU time for the optimized version of the `tachyon_find_hotspots.exe`.
3. Difference in CPU time between the two results in the following format: <Difference CPU Time> = <Result 1 CPU Time> – <Result 2 CPU Time>.
4. Comparison summary data for the two results and their difference calculated as r000hs – r001hs= Diff.



Identify the Performance Gain

Compare CPU time data for the first hotspot. 28.943s - 22.427s = 6.517s, which means that you got the optimization of 6.5 seconds for the `initialize_2D_buffer` function. The Elapsed time also shows about 8 seconds of optimization for the whole application execution.

Recap

You ran the Hotspots analysis on the optimized code and compared the results before and after optimization using the Compare mode of the Amplifier. Compare analysis results regularly to look for regressions and to track how incremental changes to the code affect its performance. You may also want to use the Amplifier command-line interface and run the `ampl-cl` command to test your code for regressions. For more details, see the *Command-line Interface Support* section in the Amplifier online help.

Key Terms and Concepts

- Term: hotspot, Elapsed time, CPU time
- Concept: Hotspots Analysis

Summary

You have completed the Finding Hotspots tutorial. Here are some important things to remember when using the Amplifier to analyze your code for hotspots:

Step 1. Choose and Build Your Target

- Configure the Microsoft* symbol server and your project properties to get the most accurate results for system and user binaries and to analyze the performance of your application at the code line level.
- Create a performance baseline to compare the application versions before and after optimization. Make sure to use the same workload for each application run.

Step 2. Run Analysis

- Use the Amplifier toolbar or **Tools** menu to choose and run the analysis. You can also use the `ampl-cl` command.
- If required, modify the default analysis settings from **Tools > Options... > Intel Parallel Amplifier 2011 > Collection**. For example, you may limit the data collection to a predefined amount of data or enable the Amplifier to collect more accurate CPU time data.

Step 3. Interpret Results and Resolve the Issue

- Start analyzing the performance of your application from the Summary tab to explore the performance metrics for the whole application. Then, move to the Bottom-up window to analyze the performance per function. Focus on the hotspots - functions that took the most CPU time - , by default, located at the top of the window.
- Double-click the hotspot function in the Bottom-up window or Call Stack tab to open its source code at the code line that took the most CPU time.
- Consider using Intel® Compiler, part of the Intel® Parallel Composer, to optimize your tuning algorithms. Explore the compiler documentation for more details.

Step 4. Compare Results Before and After Optimization

- Perform regular regression testing by comparing analysis results before and after optimization. From GUI, click the **Compare Results** button on the Amplifier toolbar. From command line, use the `ampl-cl` command.

Intel® Parallel Amplifier Tutorial: Identifying Concurrency

2

Learning Objectives

This tutorial shows how to use the Concurrency analysis of the Intel® Parallel Amplifier to understand whether your application effectively utilizes all available cores and identify the most serial code to parallelize. Ideal processor utilization occurs when the number of running threads equals the number of available cores. You have to focus on the hotspots with poor processor utilization and make all processor cores work. If you have a hotspot where cores are idle, consider adding parallelism, rebalancing or reducing contention.

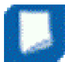
Estimated completion time: 15 minutes.

After you complete this tutorial, you should be able to:

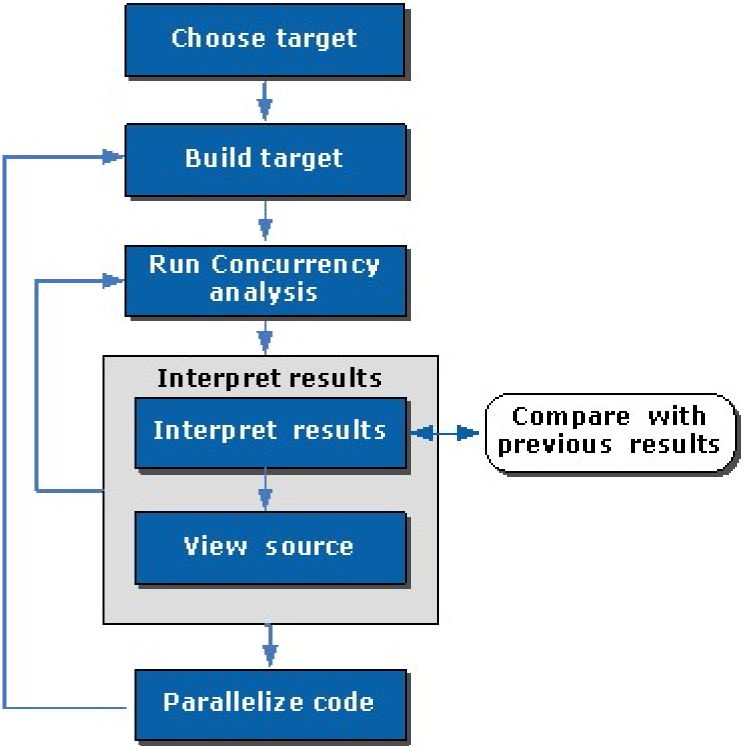
- Choose an analysis target.
- Run the Concurrency analysis.
- Estimate the concurrency level in your application.
- Identify the most time-consuming functions with poor CPU utilization.
- Analyze the source code to locate the most critical code lines.
- Compare results before and after optimization.

Workflow Steps to Analyze CPU Utilization

You can use the Intel® Parallel Amplifier to understand how your application utilized available cores by performing a series of steps in a workflow. This tutorial guides you through these workflow steps while using a sample ray-tracer application named `tachyon`.



NOTE. Click [Show Me] for a video demonstration. Show Me videos require Adobe* Flash* Player.



1. Choose a target to analyze for concurrency.
2. Configure environment and project settings and build the target.
3. Run the Concurrency analysis.
4. Interpret the result data.
5. View and analyze code of the performance-critical function.
6. Modify the code to add parallelism.
7. Re-build the target, re-run the Concurrency analysis, and compare the result data before and after optimization.

Choose Target

Before you start, you need to choose your analysis target in the Microsoft Visual Studio* environment. For this tutorial, your target is a ray-tracer application, `tachyon`. To learn how to install and set up the sample code, see Prerequisites.

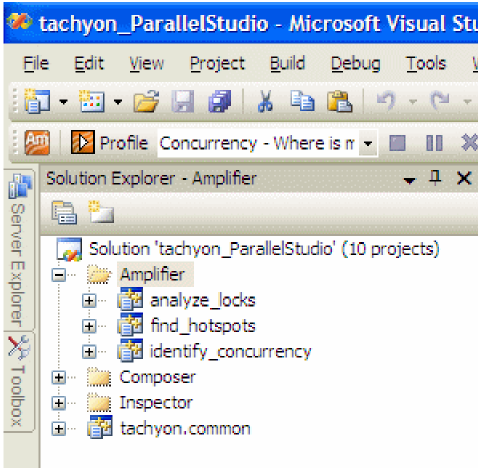
NOTE. The steps below are provided for Microsoft Visual Studio 2005. They may slightly differ for other versions of Visual Studio.

To choose a project:

1. From the Visual Studio menu, select **File > Open > Project/Solution...**.

   The **Open Project** dialog box opens.

2. In the **Open Project** dialog box, to the location you used to unzip the `tachyon_studio.zip` file and select the `tachyon_ParallelStudio.sln` file.

   The solution is added to Visual Studio and shows up in the Solution Explorer.

3. In Solution Explorer, right-click the **identify_concurrency** project and select **Project > Set as StartUp Project**.

   identify_concurrency appears in bold in Solution Explorer.

Recap

You selected the **identify_concurrency** project as the target for the Concurrency analysis.

Key Terms and Concepts

- Term: target

Build Target

After choosing the analysis target, do the followig to ensure the Intel® Parallel Amplifier provides the most accurate information on the performance of your application:

- Build the target in the release mode with full optimizations, which is recommended for performance analysis.
- Run the application without debugging to create a performance baseline.
- Configure the Microsoft Visual Studio* environment to download the debug information for system libraries so that Amplifier can properly identify system functions and classify and attribute functions.
- Configure Visual Studio project properties to generate the debug information for your application so that Amplifier can open the source code.

NOTE. The steps below are provided for Microsoft Visual Studio 2005. They may slightly differ for other versions of Visual Studio.

Choose a Build Mode and Build a Target

1. Go to the **Build > Configuration Manager...** dialog box and select the **Release** mode for your target project.
2. From the Visual Studio menu, select **Build > Build identify_concurrency**.

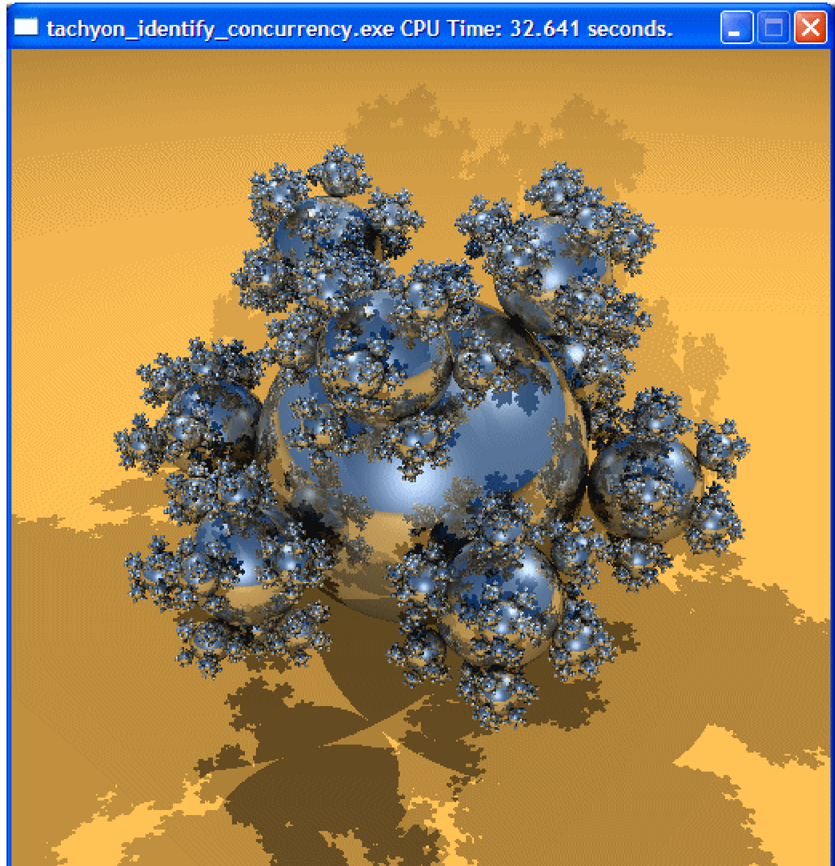
   The `tachyon_identify_concurrency.exe` application is built.

Create a Performance Baseline

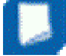
1. From the Visual Studio menu, select **Debug > Start Without Debugging**.

   The `tachyon_identify_concurrency.exe` application runs in multiple sections depending on the number of CPUs in your system.

NOTE. To get more accurate results, before you start the application, minimize the amount of other software running on your computer and disconnect from the network.



2. Note the execution time displayed in the window caption. For `tachyon_identify_concurrency.exe`, the execution time is 32.641 seconds. The total execution time is the baseline against which you will compare subsequent runs of the application.



NOTE. Run the application several times, note the execution time for each run, and use the average number. This helps to minimize skewed results due to transient system activity.

Enable Downloading the Debug Information for System Libraries

1. Go to **Tools > Options...**.

   The **Options** dialog box opens.

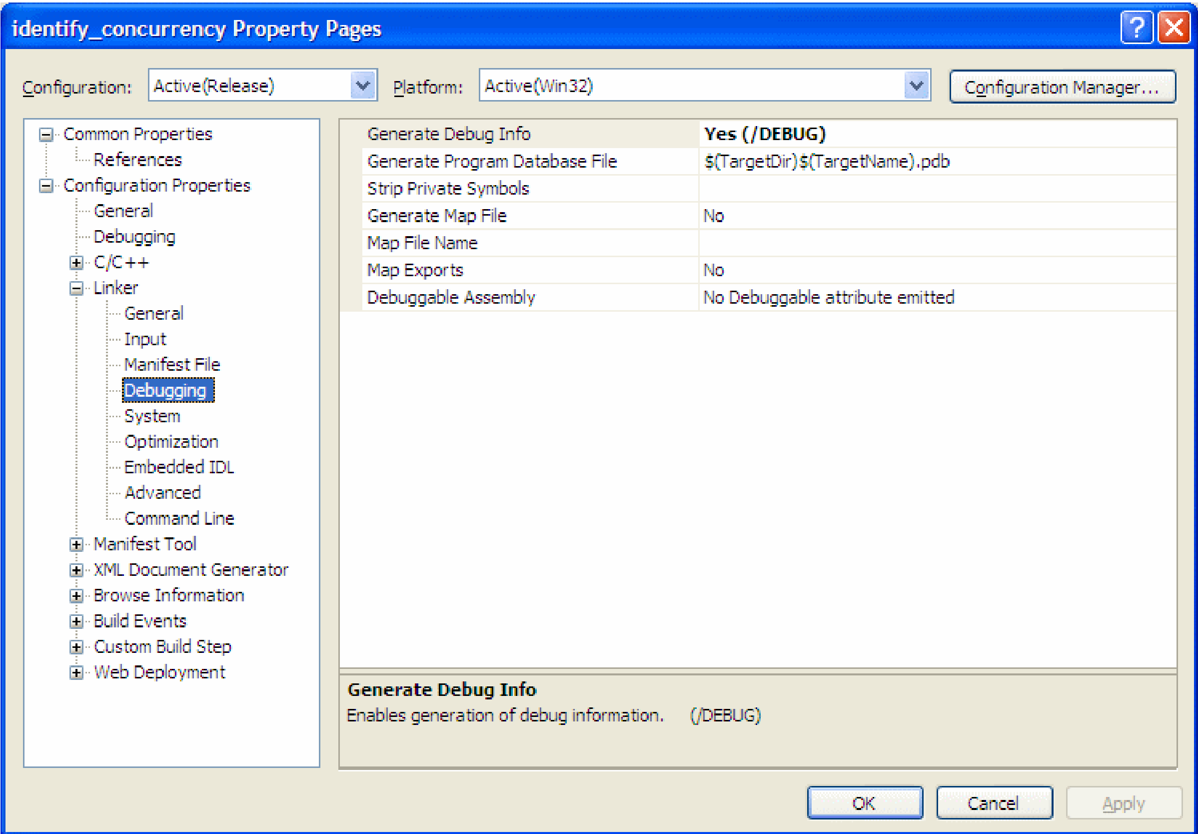
2. From the left pane, select **Debugging > Symbols**.
3. In the **Symbol file (.pdb) locations** field, click the button and specify the following address: http://msdl.microsoft.com/download/symbols.
4. Make sure the added address is checked.
5. In the **Cache symbols from symbol servers to this directory** field, specify a directory where the downloaded symbol files will be stored.
6. For Microsoft* Visual Studio* 2005, check the **Load symbols using the updated settings when this dialog is closed** box.

7. Click **Ok**.

Enable Generating Debug Information for Your Binary Files

1. Select the **identify_concurrency** project and go to **Project > Properties**.
2. From the **identify_concurrency Property Pages** dialog box, select **Configuration Properties > General** and make sure the selected **Configuration** (top of the dialog) is **Active(Release)**.
3. From the **identify_concurrency Property Pages** dialog box, select **C/C++ > General** pane and specify the **Debug Information Format** as **Program Database (/Zi)**.

4. From the **identify_concurrency Property Pages** dialog box, select **Linker > Debugging** and set the **Generate Debug Info** option to **Yes (/DEBUG)**.



Recap

You chose the target for the Concurrency analysis, set up your environment to enable generating symbol information for system libraries and your binary files, built the target in the Release mode, and created the performance baseline. Your application is ready for analysis.

Key Terms and Concepts

- Term: target, baseline

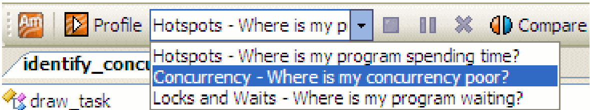
- Concept: Concurrency Analysis

Run Concurrency Analysis

After building the target, you can run it to analyze its performance. In this tutorial, you run the Concurrency analysis to identify the most serial code to parallelize.

To run an analysis:

1. From the Amplifier toolbar, select **Concurrency – Where is my concurrency poor?**



2. Click the **Profile** button.

   The Amplifier launches `tachyon_identify_concurrency.exe` that renders `balls.dat` as an input file, calculates the elapsed time, and exits. The Amplifier finalizes the collected data and opens the results in the Concurrency reslt tab.

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

Recap

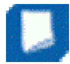
You ran the Concurrency data collection that analyzes CPU time spent in each program unit of your application and estimates whether your application utilizes available logical CPUs effectively.

NOTE. This tutorial explains how to run an analysis from the Amplifier graphical user interface (GUI). You can also use the Amplifier command-line interface (`ampl-cl` command) to run an analysis. For more details, check the *Command-line Interface Support* section of the Amplifier Help.

Key Terms and Concepts

- Term: hotspot
- Concept: Concurrency Analysis, Finalization

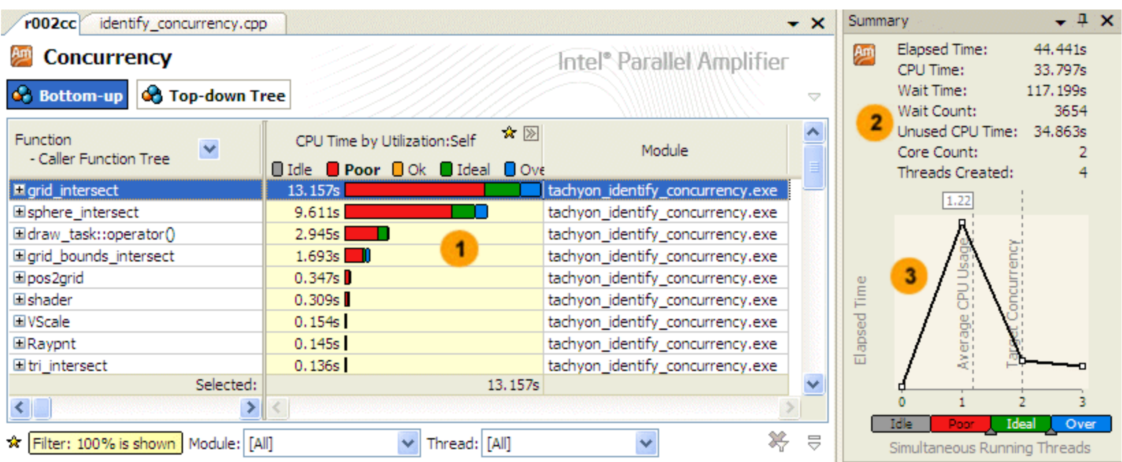
Interpret Result Data

 When the sample application exits, the Intel® Parallel Amplifier finalizes the results and opens the Concurrency window showing the call stack with functions ordered bottom-up. Follow these steps to interpret the concurrency data:

- Understand the basic performance metrics provided by the Concurrency analysis.
- Identify functions with poor CPU utilization.

NOTE. The screenshots and execution time data provided in this tutorial are created on a system with two CPU cores. Your data may vary depending on the number and type of CPU cores on your system.

Understand the Basic Concurrency Metrics

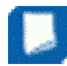


1 The utilization of the processor time per program unit (function, by default). By default, the functions are sorted by **Poor** processor utilization type. Bars showing OK or Ideal utilization (orange and green) are utilizing the processors well. You should focus your optimization efforts on functions with the longest poor CPU utilization (red bars if the bar format is selected). Next, search for the longest over-utilized time (blue bars).

This is the Data of Interest column for the concurrency analysis results. This means that the data of this column is used for different types of calculations, for example: call stack contribution, percentage value on the filter toolbar.

2 Summary data on the application performance: 1) **Elapsed Time** is the execution time of the application from start to termination; 2) **CPU Time** is the sum of CPU time for all threads; 3) **Wait Time** is the amount of time the application threads waited for some event to occur, such as synchronization waits and I/O waits; 4) **Wait Count** is the overall number of times the system wait API was called for the analyzed application; 5) **Unused CPU Time** is the total time for each core when it was either waiting or not utilized by the application; 6) **Core Count** is the logical CPU count for your machine; 5) **Threads Created** by your system during the application run.

3 Graph representing the Elapsed time and utilization level for the specified number of running threads. Ideally, your graph nodes should be within the Ok or Ideal utilization range.



NOTE. OK utilization level is not available on systems with a small number of cores.

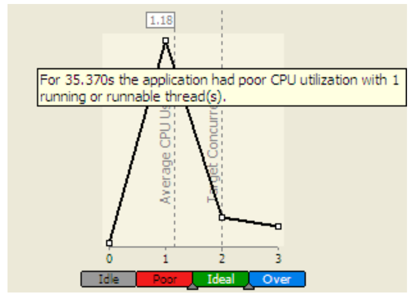
The **Target Concurrency** value, by default, is equal to the number of physical cores. Consider this number as your optimization goal.

Average CPU Usage is calculated as CPU time / Elapsed time. Use this number as a baseline for your performance measurements. The closer this number to the number of cores, the better.

Identify Functions with Poor CPU Utilization

Start with analyzing the data provided in the Summary tab. The Summary tab shows that `tachyon_identify_concurrency.exe` is a multithreaded application running three threads on a machine with two cores. But it is not using available cores effectively. The Average CPU Usage on the graph is about 1 while your target should be making it as closer to 2 as possible (for the system with 2 cores).

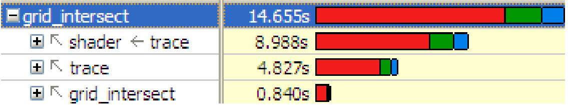
Hover over the second graph node to understand how long the application ran serially:



The tooltip shows that the application ran one thread for almost 35 seconds, which is classified as Poor CPU utilization.

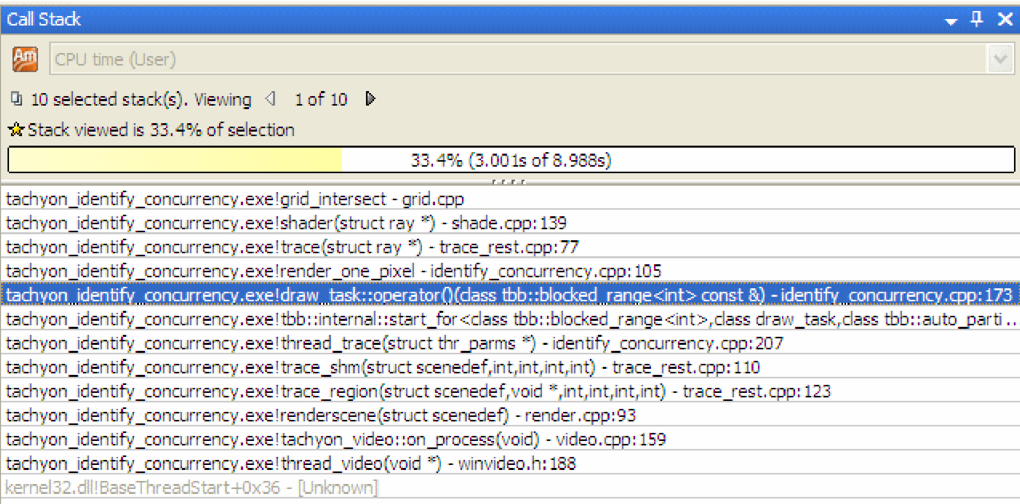
The Concurrency Bottom-up window identifies the `grid_intersect` function as the biggest hotspot. The red bars in the **CPU Time by Utilization** column indicate that most of the time processor cores were underutilized.

Click the plus sign ⊞ at the `grid_intersect` function in the Bottom-up window to expand the list of call stacks. You see three major call stacks.



The most time-critical call stack ends with the `shader` function.

Select this call stack in the Bottom-up window and explore the Call Stack tab that displays the call sequence to the `grid_intersect` function. Knowing the code, you may guess that the `draw_task` function could be the source of the bad concurrency. Double-click the function in the Call Stack tab to go to the source of this function and look for potential issues there.



Recap

You identified a hotspot function that has the worst CPU utilization and could be a good candidate for adding parallelism. You explored the call sequences to the hotspot function and found the function that could introduce bad concurrency. Your next step is to analyze the code of this function.

Key Terms and Concepts

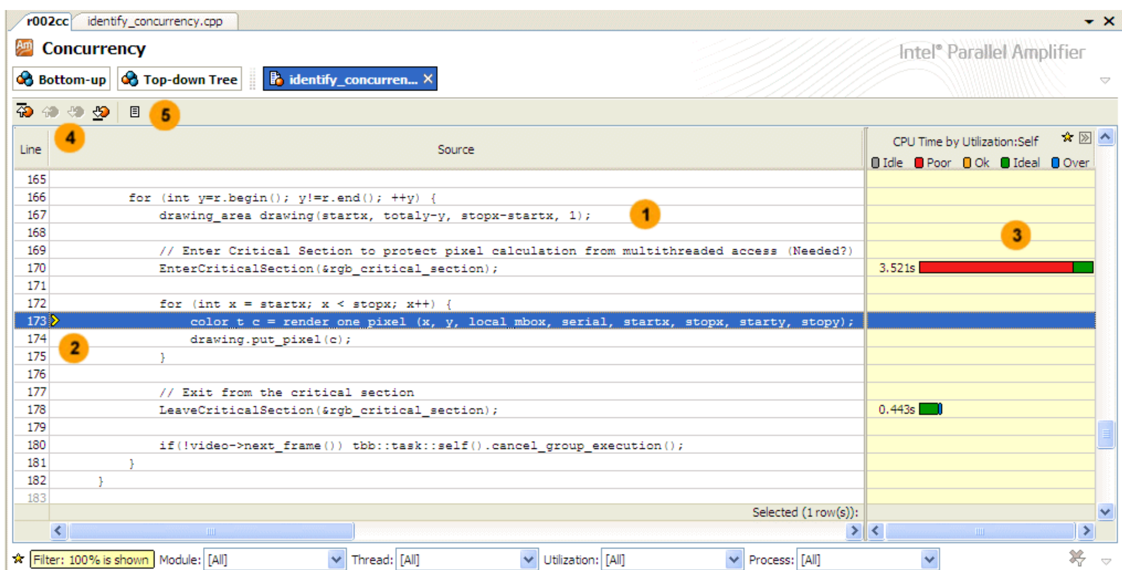
- Term: Elapsed time, CPU time
- Concept: Concurrency Analysis, CPU Utilization, Data of Interest

Analyze Code

You identified that the `initialize_2D_buffer` function took the most CPU time with poor processor utilization. Analyzing the stack calling this function, you discovered the `draw_task` function that potentially could cause performance problems. Double-click the `draw_task` function in the Call Stack tab to open the Source window and analyze the code of this function:

- Understand basic options provided in the Source window.
- Identify the hottest code lines.

Understand Basic Source View Options



The table below explains some of the features available in the Source window when viewing the Concurrency analysis data.

1	Source code of the application displayed if the function symbol information is available. If you go to the source by double-clicking the hotspot function in the Bottom-up window, the code line that took the most CPU time to execute this function is highlighted. The source code in the Source window is not editable. If the function symbol information is not available, the Assembly window opens displaying assembler instructions for the selected wait function. To enable the Source view, make sure to build the target properly.
2	Function call site. If you go to the source from the stack function, the Source window highlights the code line that called the next function from the stack and marks this call site with an yellow arrow.
3	Processor time and CPU utilization bar attributed to a particular code line. The colored bar represents the utilization of the available CPU time according to the Amplifier-defined utilization levels (Idle, Poor, Ok, Ideal, and Over). The longer the bar, the higher the value. Ok utilization level is not available for systems with a small number of cores.

4 Hotspot navigation buttons to switch between most performance-critical code lines. Hotspot navigation is based on the metric column selected as a Data of Interest. For the Concurrency analysis, this is **CPU Time by Utilization : Self**.

5 Source file editor button to open and edit your code in the Visual Studio editor.

Identify the Hottest Code Lines

Click the Hotspot navigation button on the Source window toolbar to go to largest hotspot in the `draw_task` function. The Amplifier highlights line 170 entering the critical section `rgb_critical section`. Executing this line took about 3.5 seconds and most of the time the processor was underutilized. Click the Hotspot navigation button to go to the next hotspot in the function. The Amplifier highlights line 178 where the critical section exits.

The `rgb_critical section` is the place where the application is serializing. Each thread has to wait for the critical section to be available before it can proceed. Only one thread can be in the critical section at a time. You need to optimize the code to make it more concurrent. Click the Source Editor button on the Source window toolbar to open the Visual Studio editor and work on optimizing the code.

Recap

You identified the code section that took the most CPU time to execute and significantly underutilized processor cores.

Key Terms and Concepts

- Term: CPU time
- Concept: CPU Utilization, Concurrency Analysis, Data of Interest

Add Parallelism

In the Source window, you identified that in the `draw_task` function the code line 170 underutilized the processor cores significantly and took the most CPU time to execute. Focus on this line and do the following:

- Open the code editor.

- Modify the code to add parallelism.

Open the Code Editor

Click the Source Editor button to open the `identify_concurrency.cpp` file in the Visual Studio editor at the hotspot line:

```
draw_task                    operator (const tbb::blocked_range<int> &r) const
163         unsigned int * local_mbox = (unsigned int *) alloca(mboxsize);
164         memset(local_mbox,0,mboxsize);
165
166         for (int y=r.begin(); y!=r.end(); ++y) {
167             drawing_area drawing(startx, totaly-y, stopx-startx, 1);
168
169             // Enter Critical Section to protect pixel calculation from multithreaded access (Needed?)
170             EnterCriticalSection(&rgb_critical_section);
171
172             for (int x = startx; x < stopx; x++) {
173                 color_t c = render_one_pixel (x, y, local_mbox, serial, startx, stopx, starty, stopy);
174                 drawing.put_pixel(c);
175             }
176
177             // Exit from the critical section
178             LeaveCriticalSection(&rgb_critical_section);
179
180             if(!video->next_frame()) tbb::task::self().cancel_group_execution();
181         }
182     }
183
184     draw_task () {}
185
186 };
187
188 static void parallel_thread (void) {
```

Modify the Code to Add Parallelism

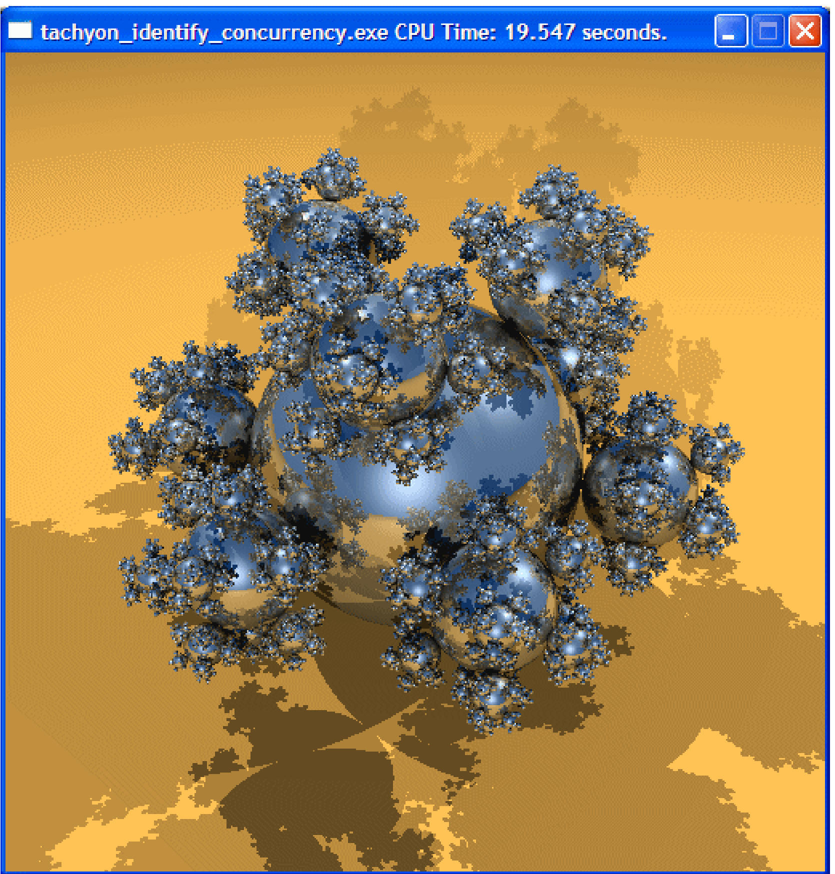
The `rgb_critical_section` was introduced to protect calculation from multithreaded access. The code is actually thread safe and the critical section is not really needed.

To resolve this issue:

1. Comment out code lines 170 and 178 to disable the critical section.
2. From Solution Explorer, select the **identify_concurrency** project.
3. From Visual Studio menu, select **Build > Rebuild identify_concurrency**.

   The project is rebuilt.
4. From Visual Studio menu, select **Debug > Start Without Debugging** to run the application.

Visual Studio runs the `tachyon_identify_concurrency.exe`. Note that execution time reduced from 32 seconds to 19.5 seconds.



Recap

You optimized the application execution time by removing the unnecessary critical section that consumed a lot of CPU time.

Key Terms and Concepts

- Term: hotspot

Compare with Previous Result

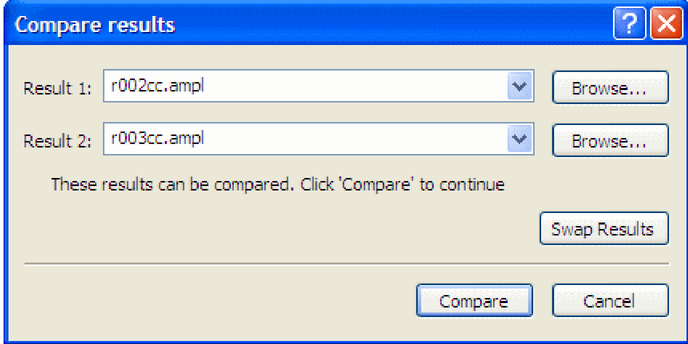
You made sure that removing the critical section gave you 12.5 seconds of optimization in the application execution time. To understand the impact of your changes per function and how the CPU utilization has changed, re-run the Concurrency analysis on the optimized code and compare results:

- Compare results before and after optimization.
- Identify the performance gain.

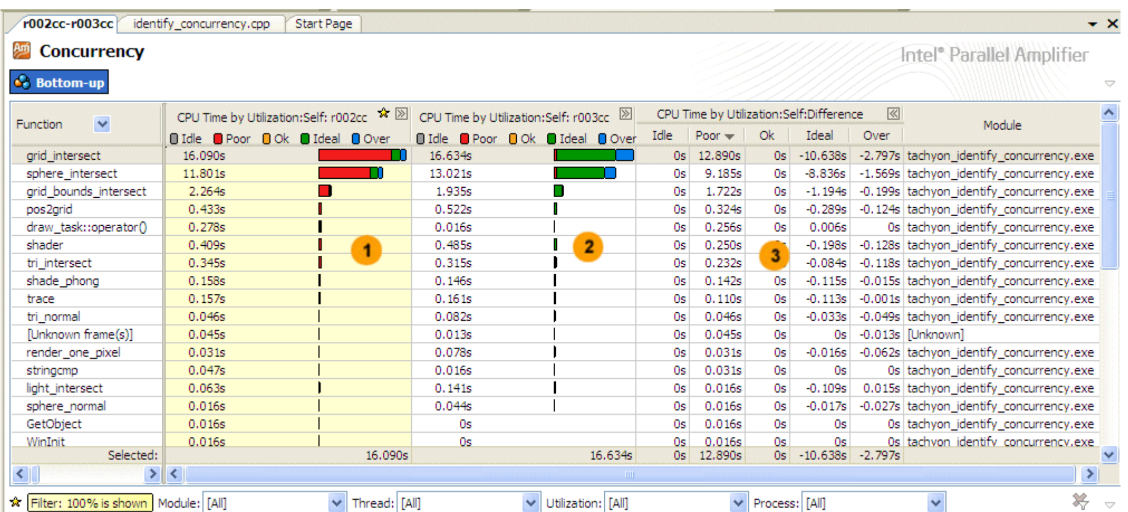
Compare Results Before and After Optimization

1. Run the Concurrency analysis on the modified code.
2. Click the Compare button on the Amplifier toolbar.

   The **Compare Results** dialog box opens.
3. Specify the Concurrency analysis results you want to compare and click the **Compare** button:



The Concurrency Bottom-up window opens, showing the CPU time utilization across the two results and the differences side by side:



r002cc-r003cc | identify_concurrency.cpp | Start Page

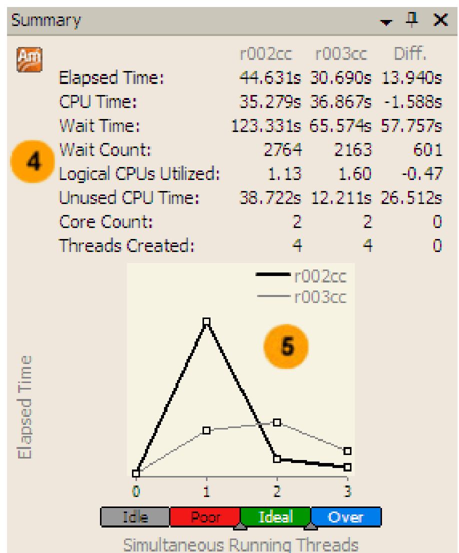
Concurrency

Bottom-up

Intel® Parallel Amplifier

Function	CPU Time by Utilization:Self: r002cc	CPU Time by Utilization:Self: r003cc	Difference: Idle	Difference: Poor	Difference: Ok	Difference: Ideal	Difference: Over	Module
grid_intersect	16.090s	16.634s	0s	12.890s	0s	-10.638s	-2.797s	tachyon_identify_concurrency.exe
sphere_intersect	11.801s	13.021s	0s	9.185s	0s	-8.836s	-1.569s	tachyon_identify_concurrency.exe
grid_bounds_intersect	2.264s	1.935s	0s	1.722s	0s	-1.194s	-0.199s	tachyon_identify_concurrency.exe
pos2grid	0.433s	0.522s	0s	0.324s	0s	-0.289s	-0.124s	tachyon_identify_concurrency.exe
draw_task::operator()	0.278s	0.016s	0s	0.256s	0s	0.006s	0s	tachyon_identify_concurrency.exe
shader	0.409s	0.485s	0s	0.250s	[illegible]	-0.198s	-0.128s	tachyon_identify_concurrency.exe
tri_intersect	0.345s	0.315s	0s	0.232s	[illegible]	-0.084s	-0.118s	tachyon_identify_concurrency.exe
shade_phong	0.158s	0.146s	0s	0.142s	0s	-0.115s	-0.015s	tachyon_identify_concurrency.exe
trace	0.157s	0.161s	0s	0.110s	0s	-0.113s	-0.001s	tachyon_identify_concurrency.exe
tri_normal	0.046s	0.082s	0s	0.046s	0s	-0.033s	-0.049s	tachyon_identify_concurrency.exe
[Unknown frame(s)]	0.045s	0.013s	0s	0.045s	0s	0s	-0.013s	[Unknown]
render_one_pixel	0.031s	0.078s	0s	0.031s	0s	-0.016s	-0.062s	tachyon_identify_concurrency.exe
stringcmp	0.047s	0.016s	0s	0.031s	0s	0s	0s	tachyon_identify_concurrency.exe
light_intersect	0.063s	0.141s	0s	0.016s	0s	-0.109s	0.015s	tachyon_identify_concurrency.exe
sphere_normal	0.016s	0.044s	0s	0.016s	0s	-0.017s	-0.027s	tachyon_identify_concurrency.exe
GetObject	0.016s	0s	0s	0.016s	0s	0s	0s	tachyon_identify_concurrency.exe
WinInit	0.016s	0s	0s	0.016s	0s	0s	0s	tachyon_identify_concurrency.exe
Selected:	16.090s	16.634s	0s	12.890s	0s	-10.638s	-2.797s	

Filter: 100% is shown Module: [All] Thread: [All] Utilization: [All] Process: [All]



Summary

	r002cc	r003cc	Diff.
Elapsed Time:	44.631s	30.690s	13.940s
CPU Time:	35.279s	36.867s	-1.588s
Wait Time:	123.331s	65.574s	57.757s
Wait Count:	2764	2163	601
Logical CPUs Utilized:	1.13	1.60	-0.47
Unused CPU Time:	38.722s	12.211s	26.512s
Core Count:	2	2	0
Threads Created:	4	4	0

1. CPU time for the initial version of the code.

2. CPU time for the optimized version of the code.

3. Difference in CPU time per utilization level between the two results in the following format: <Difference CPU Time> = <Result 1 CPU Time> – <Result 2 CPU Time>. By default, the Difference column is expanded to display comparison data per utilization level. You may collapse the column to see the total difference data per CPU time.

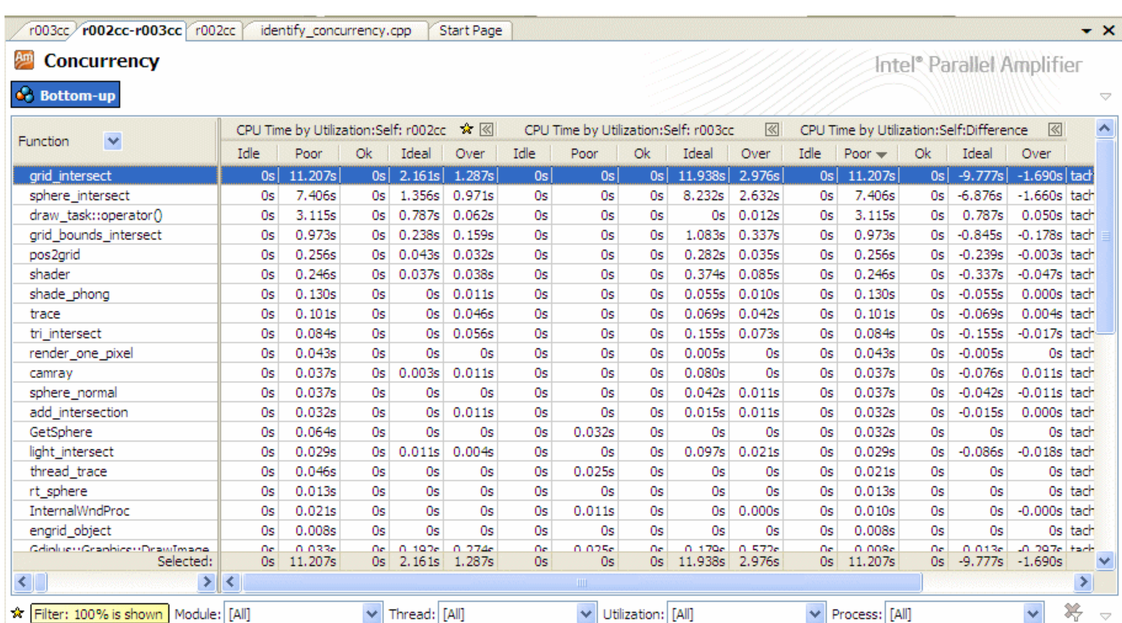
4. Comparison summary data for the two results and their difference calculated as r002cc – r003cc= Diff.

   Logical CPUs Utilized is the average utilization of all cores during application run.

5. Concurrency graphs for both compared results.

Identify the Performance Gain

Click the ⊠ button for the first three columns to expand them per utilization level.



Function	CPU Time by Utilization:Self: r002cc					CPU Time by Utilization:Self: r003cc					CPU Time by Utilization:Self:Difference				
	Idle	Poor	Ok	Ideal	Over	Idle	Poor	Ok	Ideal	Over	Idle	Poor	Ok	Ideal	Over
grid_intersect	0s	11.207s	0s	2.161s	1.287s	0s	0s	0s	11.938s	2.976s	0s	11.207s	0s	-9.777s	-1.690s
sphere_intersect	0s	7.406s	0s	1.356s	0.971s	0s	0s	0s	8.232s	2.632s	0s	7.406s	0s	-6.876s	-1.660s
draw_task::operator()	0s	3.115s	0s	0.787s	0.062s	0s	0s	0s	0s	0.012s	0s	3.115s	0s	0.787s	0.050s
grid_bounds_intersect	0s	0.973s	0s	0.238s	0.159s	0s	0s	0s	1.083s	0.337s	0s	0.973s	0s	-0.845s	-0.178s
pos2grid	0s	0.256s	0s	0.043s	0.032s	0s	0s	0s	0.282s	0.035s	0s	0.256s	0s	-0.239s	-0.003s
shader	0s	0.246s	0s	0.037s	0.038s	0s	0s	0s	0.374s	0.085s	0s	0.246s	0s	-0.337s	-0.047s
shade_phong	0s	0.130s	0s	0s	0.011s	0s	0s	0s	0.055s	0.010s	0s	0.130s	0s	-0.055s	0.000s
trace	0s	0.101s	0s	0s	0.046s	0s	0s	0s	0.069s	0.042s	0s	0.101s	0s	-0.069s	0.004s
tri_intersect	0s	0.084s	0s	0s	0.056s	0s	0s	0s	0.155s	0.073s	0s	0.084s	0s	-0.155s	-0.017s
render_one_pixel	0s	0.043s	0s	0s	0s	0s	0s	0s	0.005s	0s	0s	0.043s	0s	-0.005s	0s
camray	0s	0.037s	0s	0.003s	0.011s	0s	0s	0s	0.080s	0s	0s	0.037s	0s	-0.076s	0.011s
sphere_normal	0s	0.037s	0s	0s	0s	0s	0s	0s	0.042s	0.011s	0s	0.037s	0s	-0.042s	-0.011s
add_intersection	0s	0.032s	0s	0s	0.011s	0s	0s	0s	0.015s	0.011s	0s	0.032s	0s	-0.015s	0.000s
GetSphere	0s	0.064s	0s	0s	0s	0s	0.032s	0s	0s	0s	0s	0.032s	0s	0s	0s
light_intersect	0s	0.029s	0s	0.011s	0.004s	0s	0s	0s	0.097s	0.021s	0s	0.029s	0s	-0.086s	-0.018s
thread_trace	0s	0.046s	0s	0s	0s	0s	0.025s	0s	0s	0s	0s	0.021s	0s	0s	0s
rt_sphere	0s	0.013s	0s	0s	0s	0s	0s	0s	0s	0s	0s	0.013s	0s	0s	0s
InternalWndProc	0s	0.021s	0s	0s	0s	0s	0.011s	0s	0s	0.000s	0s	0.010s	0s	0s	-0.000s
engrid_object	0s	0.008s	0s	0s	0s	0s	0s	0s	0s	0s	0s	0.008s	0s	0s	0s
Selected:	0s	11.207s	0s	2.161s	1.287s	0s	0s	0s	11.938s	2.976s	0s	11.207s	0s	-9.777s	-1.690s

For the `grid_intersect` hotspot function, the difference is 11.207s - 0s = 11.207s of Poor CPU utilization time, which means that this function in Result 2 (optimized) did not have Poor utilization at all and effectively used processor cores. Respectively, the Ideal CPU utilization time has increased significantly: 2.161s - 11.938s = -9.777s, which means that after removing the critical section all CPU cores were effectively utilized during almost 10 seconds.

The Elapsed time data in the Summary tab shows the optimization of 13.940 seconds for the whole application execution.

Recap

You ran the Concurrency analysis on the optimized code and compared the results before and after optimization using the Compare mode of the Intel® Parallel Amplifier. The comparison shows that, with the optimized version of the `tachyon_identify_concurrency.exe` application (r003cc result), you managed to better utilize all available cores and got 13-second speedup for the application Elapsed time. Compare analysis results regularly to look for regressions and to track how incremental changes to the code affect its performance. You may also want to use the Amplifier command-line interface and run the `ampl-cl` command to test your code for regressions. For more details, see the *Command-line Interface Support* section in the Amplifier online help.

Key Terms and Concepts

- Term: hotspot
- Concept: Concurrency Analysis

Summary

You have completed the Identifying Concurrency tutorial. Here are some important things to remember when using the Amplifier to analyze your code for concurrency:

Step 1. Choose and Build Your Target

- Configure the Microsoft* symbol server and your project properties to get the most accurate results for system and user binaries and to analyze the performance of your application at the code line level.
- Create a performance baseline to compare the application versions before and after optimization. Make sure to use the same workload for each application run.

Step 2. Run Analysis

- Use the Amplifier toolbar or **Tools** menu to choose and run the analysis. You can also use the `ampl-cl` command.
- If required, modify the default analysis settings from **Tools > Options... > Intel Parallel Amplifier 2011 > Collection**. For example, you may limit the data collection to a predefined amount of data or enable the Amplifier to collect more accurate CPU time data.

Step 3. Interpret Results and Resolve the Issue

- Start analyzing the performance of your application from the Summary tab to explore the performance metrics for the whole application. Then, move to the Bottom-up window to analyze the performance per function. Focus on the hotspot functions that under- or over-utilized the available logical CPUs during their execution and took the most CPU time. By default, the functions that under-utilized available cores (Poor utilization level) most of all show up at the top of the window.
- Double-click the hotspot function in the Bottom-up window or Call Stack tab to open its source code at the code line with the worst CPU utilization.

Step 4. Compare Results Before and After Optimization

- Perform regular regression testing by comparing analysis results before and after optimization. From GUI, click the **Compare Results** button on the Amplifier toolbar. From command line, use the `ampl-cl` command.
- Expand each data column by clicking the button to identify the performance gain per CPU utilization level.

Intel® Parallel Amplifier Tutorial: Analyzing Locks and Waits

3

Learning Objectives

This tutorial shows how to use the Locks and Waits analysis of the Intel® Parallel Amplifier to identify one of the most common reasons for an inefficient parallel application - threads waiting too long on synchronization objects (locks) while processor cores are underutilized. Focus your tuning efforts on objects with long waits where the system is underutilized.

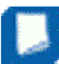
Estimated completion time: 15 minutes.

After you complete this tutorial, you should be able to:

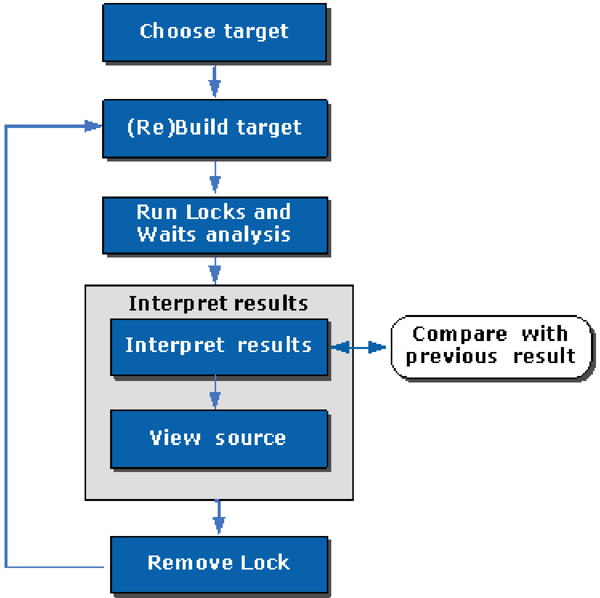
- Choose an analysis target.
- Choose the Locks and Waits analysis type.
- Run the Locks and Waits analysis.
- Identify the synchronization objects with long waits and poor CPU utilization.
- Analyze the source code to locate the most critical code lines.
- Compare results before and after optimization.

Workflow Steps to Identify Locks and Waits

You can use the Intel® Parallel Amplifier to understand the cause of the ineffective processor utilization by performing a series of steps in a workflow. This tutorial guides you through these workflow steps while using a sample ray-tracer application named `tachyon`.



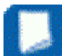
NOTE. Click Show Me for a video demonstration. Show Me videos require Adobe* Flash* Player.



1. Choose a target to analyze for locks and waits.
2. Configure environment and project settings and build the target.
3. Run the Locks and Waits analysis.
4. Interpret the result data.
5. View and analyze code of the performance-critical function.
6. Modify the code to remove the lock.
7. Re-build the target, re-run the Locks and Waits analysis, and compare the result data before and after optimization.

Choose Target

Before you start you need to choose the target in the Microsoft Visual Studio* environment. For this tutorial, your target is a ray-tracer application, `tachyon`. To learn how to install and set up the sample code, see Prerequisites.



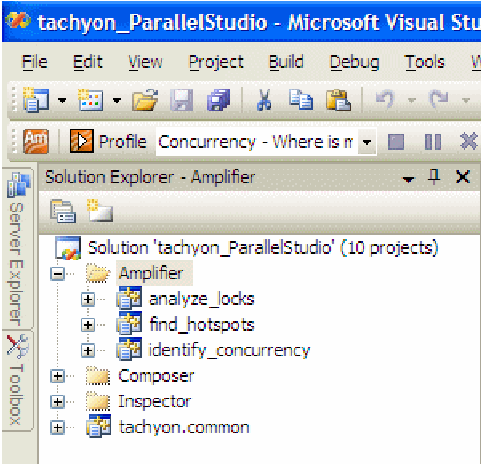
NOTE. The steps below are provided for Microsoft Visual Studio* 2005. Steps for other versions of Visual Studio IDE may slightly differ. See online help for details.

To choose a target:

1. From the Visual Studio menu, select **File > Open > Project/Solution...**.

   The **Open Project** dialog box opens.

2. In the **Open Project** dialog box, browse to the location you used to unzip the `tachyon_studio.zip` file and select the `tachyon_ParallelStudio.sln` file.

   The solution is added to Visual Studio and shows up in the Solution Explorer.

3. In Solution Explorer, right-click the **analyze_locks** project and select **Project > Set as StartUp Project**.

   analyze_locks appears in bold in Solution Explorer.

Recap

You selected the **analyze_locks** project as the target for the Locks and Waits analysis.

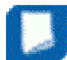
Key Terms and Concepts

- Term: target

Build Target

After choosing the analysis target, do the following to ensure the Intel® Parallel Amplifier provides the most accurate information on the performance of your application:

- Build the target in the release mode with full optimizations, which is recommended for performance analysis.
- Run the application without debugging to create a performance baseline.
- Configure the Microsoft Visual Studio* environment to download the debug information for system libraries so that Amplifier can properly identify system functions and classify and attribute functions.
- Configure Visual Studio project properties to generate the debug information for your application so that Amplifier can open the source code.



NOTE. The steps below are provided for Microsoft Visual Studio* 2005. Steps for other versions of Visual Studio IDE may slightly differ. See online help for more details.

Choose a Build Mode and Build a Target

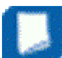
1. Go to the **Build > Configuration Manager...** dialog box and select the **Release** mode for your target project.
2. From the Visual Studio menu, select **Build > Build analyze_locks**.

   The `analyze_locks` application is built.

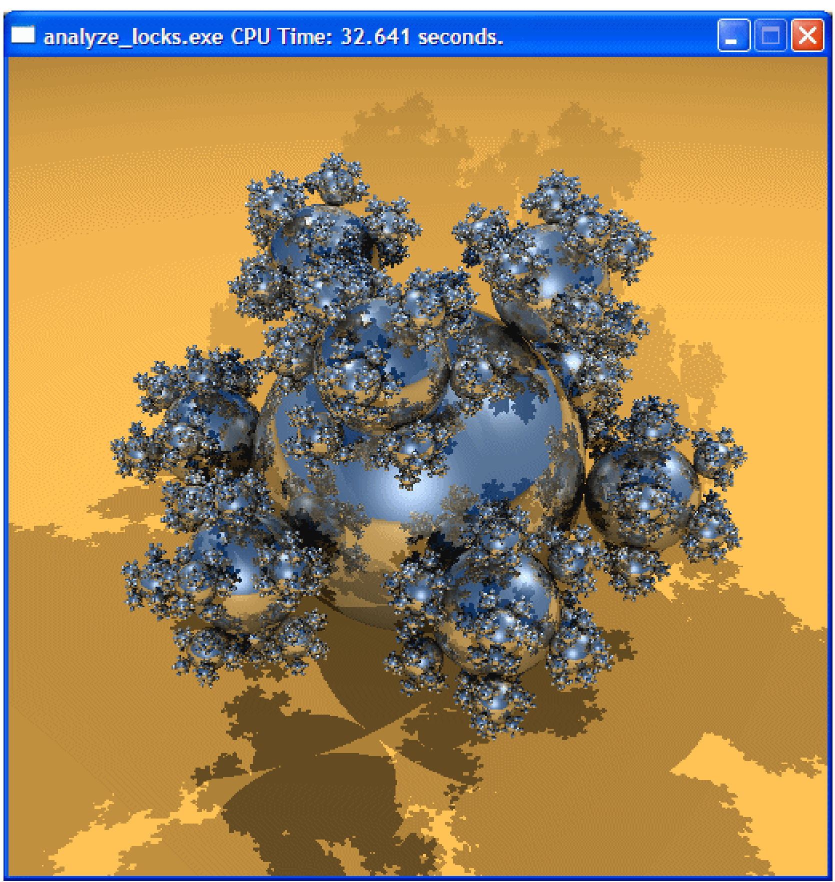
Create a Performance Baseline

1. From the Visual Studio menu, select **Debug > Start Without Debugging**.

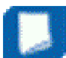
   The `analyze_locks` application runs in multiple sections (depending on the number of CPUs in your system).



NOTE. Before you start the application, minimize the amount of other software running on your computer to get more accurate results.



2. Note the execution time displayed in the window caption. For the `analyze_locks` executable in the figure above, the execution time is 32.641 seconds. The total execution time is the baseline against which you will compare subsequent runs of the application.



NOTE. Run the application several times, note the execution time for each run, and use the average number. This helps to minimize skewed results due to transient system activity.

Enable Downloading the Debug Information for System Libraries

1. Go to **Tools > Options...**.

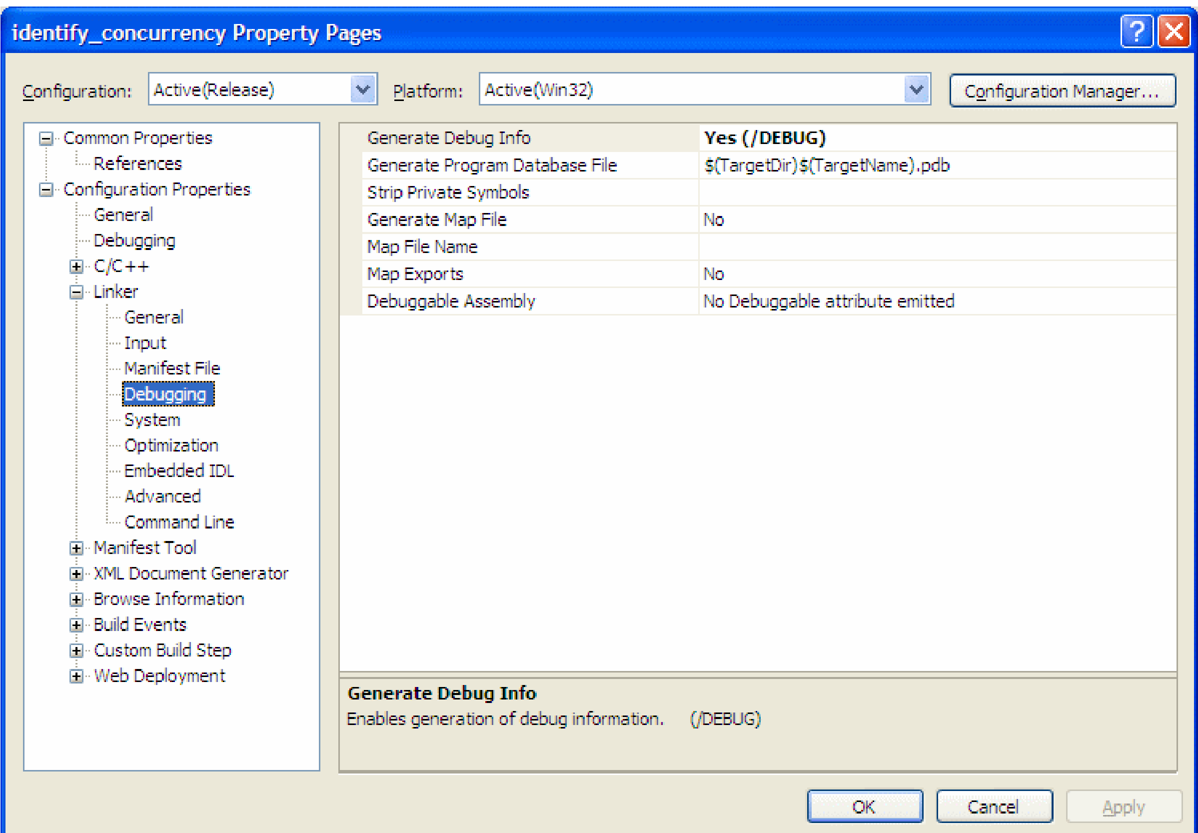
   The **Options** dialog box opens.
2. From the left pane, select **Debugging > Symbols**.
3. In the **Symbol file (.pdb) locations** field, click the button and specify the following address: http://msdl.microsoft.com/download/symbols.
4. Make sure the added address is checked.
5. In the **Cache symbols from symbol servers to this directory** field, specify a directory where the downloaded symbol files will be stored.
6. For Microsoft* Visual Studio* 2005, check the **Load symbols using the updated settings when this dialog is closed** box.

7. Click **Ok**.

Enable Generating Debug Information for Your Binary Files

1. Select the **analyze_locks** project and go to **Project > Properties**.
2. From the **analyze_locks Property Pages** dialog box, select **Configuration Properties > General** and make sure the selected **Configuration** (top of the dialog) is **Active(Release)**.
3. From the **analyze_locks Property Pages** dialog box, select **C/C++ > General** pane and specify the **Debug Information Format** as **Program Database (/Zi)**.

4. From the **analyze_locks Property Pages** dialog box, select **Linker > Debugging** and set the **Generate Debug Info** option to **Yes (/DEBUG)**.



Recap

You set up your environment to enable generating symbol information for system libraries and your binary files, built the target in the Release mode, and created the performance baseline. Your application is ready for analysis.

Key Terms and Concepts

- Term: target, baseline

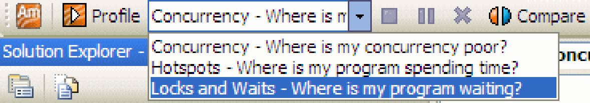
- Concept: Locks and Waits Analysis

Run Locks and Waits Analysis

After building the application, you can run it to analyze its performance. In this tutorial, you run the Locks and Waits analysis to identify synchronization objects that caused contention and fix the problem in the source.

To run an analysis:

1. From the Amplifier toolbar, select **Locks and Waits – Where is my program waiting?**



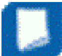
2. Click the **Profile** button.

The Amplifier launches the `analyze_locks` executable that renders `balls.dat` as an input file, calculates the execution time, and exits. The Amplifier finalizes the collected data and opens the results in the Locks and Waits result tab.

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

Recap

You ran the Locks and Waits data collection that analyzes how long the application had to wait on each synchronization object, or in blocking APIs, such as sleep and blocking I/O, and estimates processor utilization during the wait.



NOTE. This tutorial explains how to run an analysis from the Amplifier graphical user interface (GUI). You can also use the Amplifier command-line interface (`ampl-cl` command) to run an analysis. For more details, check the *Command-line Interface Support* section of the Amplifier Help.

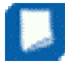
Key Terms and Concepts

- Concept: Locks and Waits Analysis, Finalization

Interpret Result Data

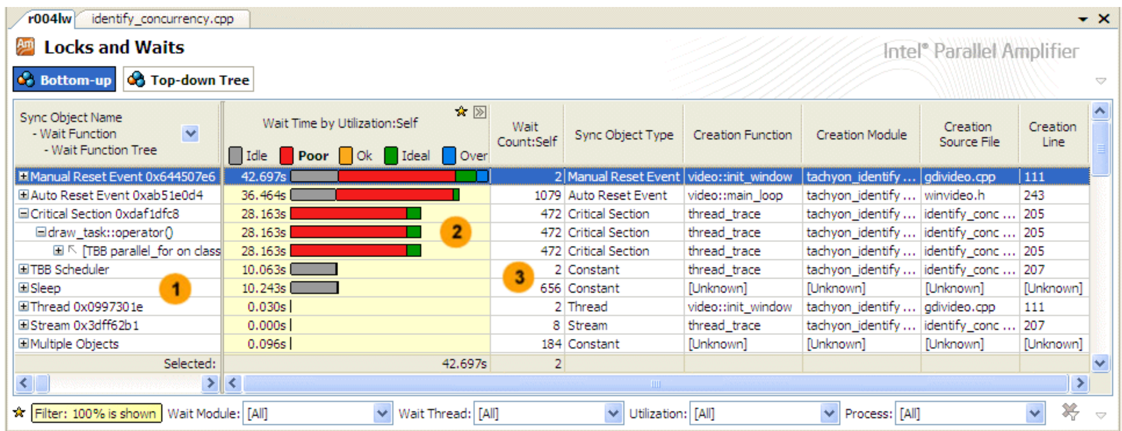
When the sample application exits, the Intel® Parallel Amplifier finalizes the results and opens the Locks and Waits window displaying CPU time spent on synchronization objects used in your application. Follow these steps to interpret the locks and waits data:

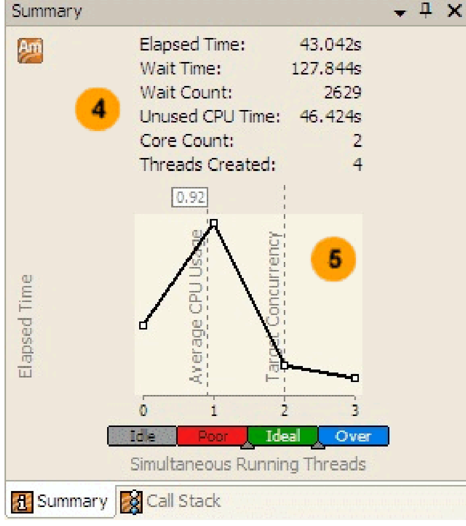
- Understand the basic performance metrics provided by the Locks and Waits analysis.
- Identify synchronization objects with high Wait time values and poor CPU utilization.



NOTE. The screen shots and execution time data provided in this tutorial are created on a system with two CPU cores. Your data may vary depending on the number and type of CPU cores on your system.

Understand the Basic Locks and Waits Metrics





1. Synchronization objects that control threads in the application. The hash (unique number) appended to some names of the objects identify the stack creating this synchronization object.

   For Intel® Threading Building Blocks (Intel® TBB), Amplifier is able to recognize all types of Intel TBB objects. To display an overhead introduced by Intel TBB library internals, the Amplifier creates a pseudo synchronization object **TBB scheduler** that includes all waits from the Intel TBB runtime libraries.

2. The utilization of the processor time when a given thread waited for some event to occur. By default, the synchronization objects are sorted by **Poor** processor utilization type. Bars showing OK or Ideal utilization (orange and green) are utilizing the processors well. You should focus your optimization efforts on functions with the longest poor CPU utilization (red bars if the bar format is selected). Next, search for the longest over-utilized time (blue bars).

   This is the Data of Interest column for the Locks and Waits analysis results. This means that the data of this column is used for different types of calculations, for example: call stack contribution, percentage value on the filter toolbar.

3. Number of times the corresponding system wait API was called. For a lock, it is the number of times the lock was contended and caused a wait. Usually you are recommended to focus your tuning efforts on the waits with both high Wait Time and Wait Count values, especially if they have poor utilization.

4. Summary data on the application performance: 1) **Elapsed Time** is the execution time of the application from start to termination; 2) **Wait Time** is the amount of time the application threads waited for some event to occur, such as synchronization waits and I/O waits; 4) **Wait Count** is the overall number of times the system wait API was called for the analyzed

application; 5) **Unused CPU Time** is the total time for each core when it was either waiting or not utilized by the application; 6) **Core Count** is the logical CPU count for your machine; 5) **Threads Created** by your system during the application run.

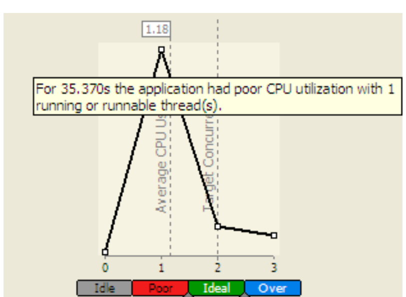
Graph representing the Elapsed time and utilization level for the specified number of running threads. Ideally, your graph nodes should be within the Ok or Ideal utilization range.

Note the **Target Concurrency** value. By default, this number is equal to the number of physical cores. Consider this number as your optimization goal.

Average CPU Usage is calculated as CPU time / Elapsed time. Use this number as a baseline for your performance measurements. The closer this number to the number of cores, the better.

Identify Locks

Start with analyzing the data provided in the Summary tab. The Summary tab shows that `tachyon_identify_concurrency.exe` is a multithreaded application running three threads on a machine with two cores. But it is not using available cores effectively. The Average CPU Usage on the graph is about 1 while your target should be making it as closer to 2 as possible (for the system with 2 cores).



Hover over the second graph node to understand how long the application ran serially. The tooltip shows that the application ran one thread for almost 35 seconds, which is classified as Poor CPU utilization.

Explore the Bottom-up window. You see that the top three synchronization objects caused the longest wait time. The red bars in the **Wait Time by Utilization** column indicate that most of the time for these objects processor cores were underutilized.

From the code knowledge, you may understand that the first two objects are most likely related to the join where the main program is waiting for the worker threads to finish. This should not be a problem. Consider the third item in the Bottom-up window that is more interesting. It is a Critical Section that shows much serial time and is causing a wait. Click the plus sign ⊞ at the object name to

expand the node and see the `draw_task` wait function that contains this critical section and Wait Function Trees. Double-click the Critical Section to see the source code for the wait function.

Recap

You identified a synchronization object with the high Wait Time and Wait Count values and poor CPU utilization that could be a lock affecting application concurrency. Your next step is to analyze the code of this function.

Key Terms and Concepts

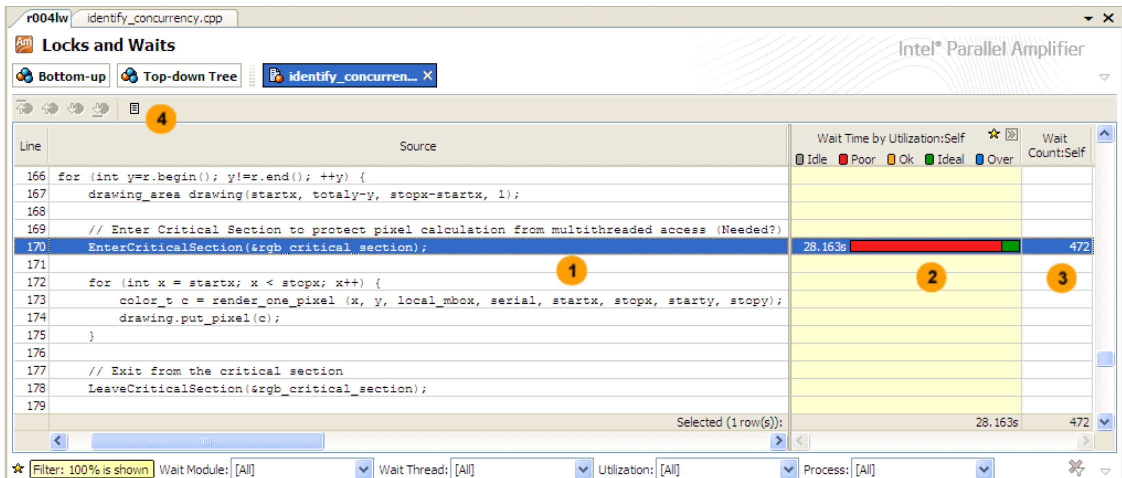
- Term: Elapsed time, Wait time
- Concept: Locks and Waits Analysis, CPU Utilization, Data of Interest

Analyze Code

You identified the critical section that caused significant Wait time and poor processor utilization. Double-click this critical section in the Bottom-up window to open the Source window and analyze the source code:

- Understand basic options provided in the Source window.
- Identify the hottest code lines.

Understand Basic Source View Options



The table below explains some of the features available in the Source window when viewing the Locks and Waits analysis data.

1	Source code of the application displayed if the function symbol information is available. When you go to the source by double-clicking the synchronization object in the Bottom-up window, the Amplifier opens the wait function containing this object and highlights the code line that took the most Wait time. The source code in the Source window is not editable. If the function symbol information is not available, the Assembly window opens displaying assembler instructions for the selected wait function. To enable the Source view, make sure to build the target properly.
2	Processor time and utilization bar attributed to a particular code line. The colored bar represents the distribution of the Wait time according to the Amplifier-defined utilization levels (Idle, Poor, Ok, Ideal, and Over). The longer the bar, the higher the value. Ok utilization level is not available for systems with a small number of cores. This is the Data of Interest column for the Locks and Waits analysis.
3	Number of times the corresponding system wait API was called while this code line was executing. For a lock, it is the number of times the lock was contended and caused a wait.

Source file editor button to open and edit your code in the Visual Studio editor.

Identify the Hottest Code Lines

The Amplifier highlights line 170 entering the critical section `rgb_critical section` in the `draw_task` function. The `draw_task` function was waiting for almost 28 seconds while this code line was executing and most of the time the processor was underutilized. During this time, the critical section was contended 472 times.

The `rgb_critical section` is the place where the application is serializing. Each thread has to wait for the critical section to be available before it can proceed. Only one thread can be in the critical section at a time. You need to optimize the code to make it more concurrent. Click the Source Editor button on the Source window toolbar to open the code editor and work on optimizing the code.

Recap

You identified the code section that caused a significant wait and during which the processor was poorly utilized.

Key Terms and Concepts

- Term: CPU time
- Concept: CPU Utilization, Locks and Waits Analysis, Data of Interest

Remove Lock

In the Source window, you located the critical section that caused a significant wait while the processor cores were underutilized and generated multiple wait cout. Focus on this line and do the following:

- Open the code editor.
- Modify the code to remove the lock.

Open the Code Editor

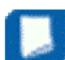
Click the Source Editor button to open the `analyze_locks.cpp` file in the Visual Studio editor at the hotspot code line:

```
draw_task                                   operator (const tbb::blocked_range<int> &r) const
163         unsigned int * local_mbox = (unsigned int *) alloca(mboxsize);
164         memset(local_mbox,0,mboxsize);
165
166         for (int y=r.begin(); y!=r.end(); ++y) {
167             drawing_area drawing(startx, totaly-y, stopx-startx, 1);
168
169             // Enter Critical Section to protect pixel calculation from multithreaded access (Needed?)
170             EnterCriticalSection(&rgb_critical_section);
171
172             for (int x = startx; x < stopx; x++) {
173                 color_t c = render_one_pixel (x, y, local_mbox, serial, startx, stopx, starty, stopy);
174                 drawing.put_pixel(c);
175             }
176
177             // Exit from the critical section
178             LeaveCriticalSection(&rgb_critical_section);
179
180             if(!video->next_frame()) tbb::task::self().cancel_group_execution();
181         }
182     }
183
184     draw_task () {}
185
186 };
187
188 static void parallel_thread (void) {
```

Remove the Lock

The `rgb_critical_section` was introduced to protect calculation from multithreaded access. The brief analysis shows that the code is thread safe and the critical section is not really needed.

To resolve this issue:

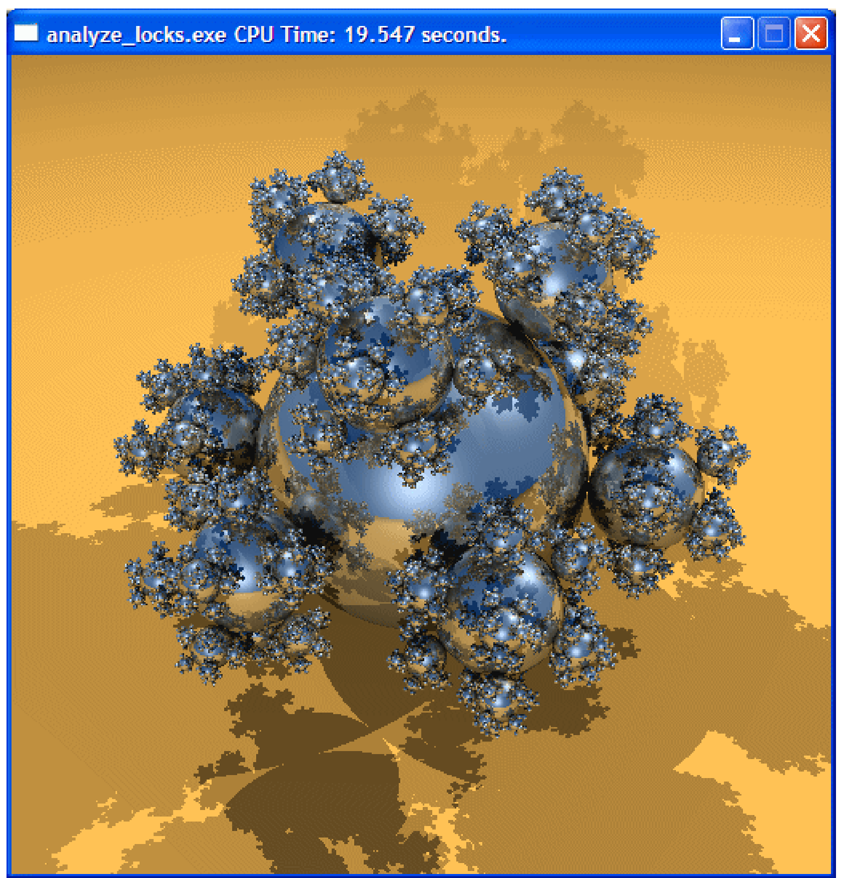


NOTE. The steps below are provided for Microsoft Visual Studio* 2005. Steps for other versions of Visual Studio IDE may slightly differ. See online help for details.

1. Comment out code lines 170 and 178 to disable the critical section.
2. From Solution Explorer, select the **analyze_locks** project.
3. From Visual Studio menu, select **Build > Rebuild analyze_locks**.

   The project is rebuilt.
4. From Visual Studio menu, select **Debug > Start Without Debugging** to run the application.

   Visual Studio runs the `analyze_locks.exe`. Note that execution time reduced from 32 seconds to 19.5 seconds.



Recap

You optimized the application execution time by removing the unnecessary critical section that caused a lot of Wait time.

Key Terms and Concepts

- Term: hotspot
- Concept: Locks and Waits Analysis

Compare with Previous Result

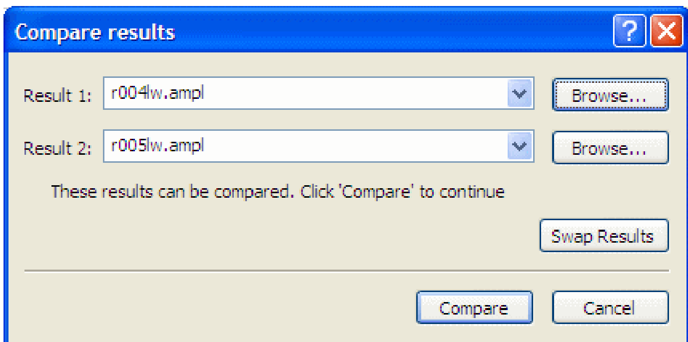
You made sure that removing the critical section gave you 12.5 seconds of optimization in the application execution time. To understand the impact of your changes and how the CPU utilization has changed, re-run the Locks and Waits analysis on the optimized code and compare results:

- Compare results before and after optimization.
- Identify the performance gain.

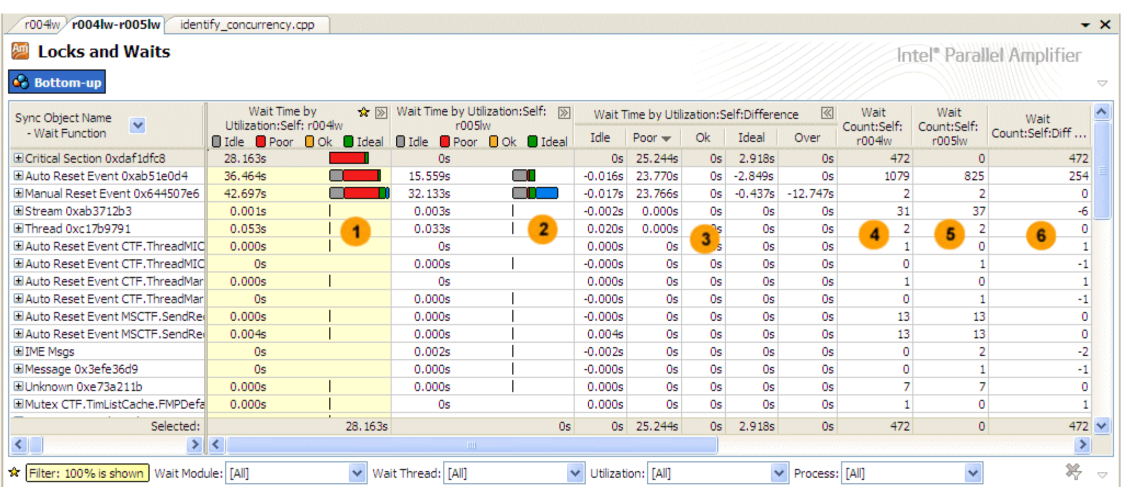
Compare Results Before and After Optimization

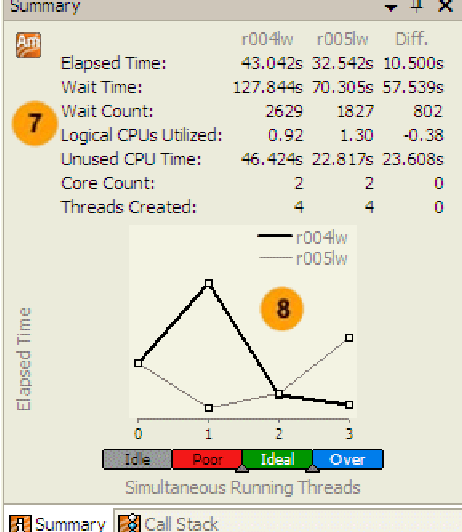
1. Run the Locks and Waits analysis on the modified code.
2. Click the Compare button on the Amplifier toolbar.

   The **Compare Results** dialog box opens.
3. Specify the Locks and Waits analysis results you want to compare:



   The Locks and Waits Bottom-up window opens, showing the list of synchronization objects used in the code, Wait time utilization across the two results, and the differences side by side:





1. Wait time and CPU utilization for the initial version of the code.
2. Wait time and CPU utilization for the optimized version of the code.
3. Difference in Wait time per utilization level between the two results in the following format: <Difference Wait Time> = <Result 1 Wait Time> – <Result 2 Wait Time>. By default, the Difference column is expanded to display comparison data per utilization level. You may collapse the column to see the total difference data per Wait time.
4. Wait count for the initial version of the code.
5. Wait count for the optimized version of the code.

6 Difference in Wait count between the two results in the following format: <Difference Wait Count> = <Results 1 Wait Count> - <Result 2 Wait Count>.

7 Comparison summary data for the two results and their difference calculated as r004lw – r005lw= Diff.

Logical CPUs Utilized is the average utilization of all cores during application run.

8 Concurrency graphs for both compared results.

Identify the Performance Gain

In the Bottom-up window, locate the Critical Section you identified as a bottleneck in your code. Since you removed it during optimization, the optimized result r005lw does not show any performance data for this synchronization object. If you collapse the **Wait Time:Difference** column by clicking the button, you see that with the optimized result you got almost 28 seconds of optimization in Wait time.

The Elapsed time data in the Summary tab shows the optimization of 10.5 seconds for the whole application execution and Wait time decreased by 57 seconds. According to the concurrency graph in the Summary tab, before optimization the application ran serially for the most of the time poorly utlizing available two processor cores but after optimization it ran multiple threads (two to three) most of the time, sometimes overutilizing the cores.

Recap

You ran the Locks and Waits analysis on the optimized code and compared the results before and after optimization using the Compare mode of the Amplifier . The comparison shows that, with the optimized version of the `tachyon_analyze_locks` application (r005lw result), you managed to effectively utilize all available cores and got 10.5-second speedup for the application Elapsed time. Compare analysis results regularly to look for regressions and to track how incremental changes to the code affect its performance. You may also want to use the Amplifier command-line interface and run the `ampl-cl` command to test your code for regressions. For more details, see the *Command-line Interface Support* section in the Amplifier online help.

Key Terms and Concepts

- Term: hotspot, Wait time, Elapsed time

- Concept: Locks and Waits Analysis, CPU Utilization

Summary

You have completed the Analyzing Locks and Waits tutorial. Here are some important things to remember when using the Amplifier to analyze your code for locks and waits:

Step 1. Choose and Build Your Target

- Configure the Microsoft* symbol server and your project properties to get the most accurate results for system and user binaries and to analyze the performance of your application at the code line level.
- Create a performance baseline to compare the application versions before and after optimization. Make sure to use the same workload for each application run.

Step 2. Run Analysis

- Use the Amplifier toolbar or **Tools** menu to choose and run the analysis. You can also use the `ampl-cl` command.
- If required, modify the default analysis settings from **Tools > Options... > Intel Parallel Amplifier 2011 > Collection**. For example, you may limit the data collection to a predefined amount of data or enable the Amplifier to collect more accurate CPU time data.

Step 3. Interpret Results and Resolve the Issue

- Start analyzing the performance of your application from the Summary tab to explore the performance metrics for the whole application. Then, move to the Bottom-up window to analyze the synchronization objects. Focus on the synchronization objects that under- or over-utilized the available logical CPUs and have the highest Wait time and Wait Count values. By default, the objects with the highest Wait time values show up at the top of the window.
- Expand the most time-critical synchronization object in the Bottom-up window and double-click the wait function it belongs to. This opens the source code for this wait function at the code line with the highest Wait time value.

Step 4. Compare Results Before and After Optimization

- Perform regular regression testing by comparing analysis results before and after optimization. From GUI, click the **Compare Results** button on the Amplifier toolbar. From command line, use the `ampl-cl` command.
- Expand each data column by clicking the button to identify the performance gain per CPU utilization level.

More Resources

4

Getting Help and Support

Intel® Parallel Amplifier provides a number of Getting Started tutorials. These tutorials use a sample application to demo you the basic product features and workflows. You can access these documents from Microsoft Visual Studio* environment either through the **Help** menu or by clicking the Amplifier icon .

From the Visual Studio user interface, select **Help > Intel Parallel Studio 2011 > Getting Started > Amplifier Tutorials** and explore available tutorials.

Browsing Help

In the Visual Studio IDE, you can browse and search for topics in different ways:

- Use **Help** > **Contents** to open the Contents window and browse the Table of Contents.
- To view help for an installed Parallel Studio tool, select **Help > Intel Parallel Studio 2011 > Parallel Studio Help > Amplifier Help**.
- Use **Help** > **Index** to open the Index window and access an index to Amplifier topics. Either type in the keyword you are looking for, or scroll through the list of keywords.
- Use **Help** > **Search** to open the Search page and search the full text of topics in the help.

Locating Intel Topics in the Document Explorer

To filter the documentation so that only the Intel documentation appears, select **Help** > **Contents** from the Visual Studio user interface. In the **Filtered by:** drop-down list, select **Intel**.

To determine where the currently displayed topic appears in the table of contents (TOC), click the **Sync with Table of Contents** button on the Visual Studio toolbar to highlight the topic in the Contents pane.

Navigating in the Product Usage Workflow

Where applicable, the Amplifier help topics provide a **Where am I in the workflow?** button. Click the button to view the workflow with a highlight on the stage that this topic discusses.

Activating Intel Search Filters in the Document Explorer

With Microsoft Visual Studio 2005 and 2008, you can include Intel documentation in all search results by checking the **Intel** search filter box for the **Language**, **Technology**, and **Content Type** categories. You must check the **Intel** search box for all three categories to include Intel documentation in your searches. Unchecking all three **Intel** search boxes excludes Intel documentation from search results. The Intel search filters work in combination with other search options for each category.

Using Context-Sensitive Help

Context-sensitive help enables easy access to help topics on active GUI elements. The following context-sensitive help features are available on a product-specific basis:

- **? Help**: In Visual Studio, click the ? button, in the upper-right corner of the dialog box or pane to get help for the dialog box or pane.
- **F1 Help**: Press F1 to get help for an active dialog box, property page, pane, or window.
- **Dynamic Help**: In Visual Studio 2005/2008, select **Help > Dynamic Help** to open the Dynamic Help window, which displays links to relevant help topics for the current window.

System Requirements

For detailed information on system requirements, see the product Release Notes.

Product Support

Product Website and Support

The following links provide information and support on Intel software products, including Intel® Parallel Studio:

- http://www.intel.com/software/products/

  At this site, you will find comprehensive product information, including:

  - Links to each product, where you will find technical information such as white papers and articles
  - Links to user forums
  - Links to news and events

- http://software.intel.com/en-us/articles/intel-parallel-studio/

  Intel® Software Network, Parallel Studio Support page, with links to support forums, startup help, knowledge base, and getting started video.

- http://software.intel.com/en-us/articles/tools/

  Intel® Software Development Products Knowledge Base.

- http://www.intel.com/software/products/support/

  Technical support information, to register your product, or to contact Intel.

For additional support information, see the Technical Support section of your Release Notes.

System Requirements

For detailed information on system requirements, see the Release Notes.