



Microsoft®
Parallel Computing Platform



Hands-On Lab

Introduction to the Visual Studio 2010 Parallel Performance Analyzer

Lab version: 1.0.0

Last updated: 3/13/2010



developer & platform **evangelism**

Contents

Introduction.....3

Parallel Performance Analysis Introduction4

 CPU Utilization View4

 Threads View (Parallel Performance)5

 Cores View7

 General tips:.....8

Introduction

Representing system responsiveness and productivity, performance has always been a fundamental metric; very related to customer satisfaction. Developers constantly require more powerful tools to analyze their applications and achieve this performance goal.

As a developer gains insight on his/her application, the more he/she understands how it is being executed on the target hardware, and how the operating system politics affect the application, the most he/she can optimize the application for performance. Regardless of how well we think we may know our applications, it is often the case that the real behavior of the application we are creating is drastically different than the application's mental model we may have. This difference is more evident when developing applications that use threads and execute in parallel.

Visual Studio 2010 includes new tools to make it easier for you to develop, understand, visualize and debug parallel applications. This micro lab will bring you up-to-date in becoming acquainted with the different views available in the **Parallel Performance Analysis (PPA)** Concurrency Visualization Tools. The PPA Tools contain different visualization models that can assist us in gaining a better understanding of each step our applications takes when running in parallel.

On this micro lab, we will work with a demo that utilizes 4 threads to take advantage of the 4 cores, present on the demo machine. We are going to ensure that the code is doing what we want and that parallelism is going according to the plan. We will use the different PPA Concurrency Visualization views to visualize this threaded demo behavior.

Parallel Performance Analysis Introduction

Parallel Performance Analysis (PPA) Concurrency Visualization views make it possible for us to see how a multi-threaded application interacts with the available hardware, the operating system, and other processors that are also running on the host machine. These views show the temporal relationships between your program threads and the system, presenting the regarded information graphically, and textually.

The Concurrency Visualizer relies on **Event Tracing for Windows (ETW)** functionality that is found in Windows Vista and later versions. ETW provides a set of event tracing features used to log and trace events that are raised by user-mode applications and kernel-mode drivers. For more information on ETW, please visit the following link: <http://msdn.microsoft.com/en-us/library/aa468736.aspx>

The Concurrency Visualizer can be used to locate performance bottlenecks, CPU underutilization, thread contention, thread, thread migration, synchronization delays, and areas of overlapped I/O. Graphical output in the Concurrency Visualizer views is linked to call stacks and source code. The high integration with Visual Studio 2010 enables application efficient workflows of test, build, analyze, fix, and retest.

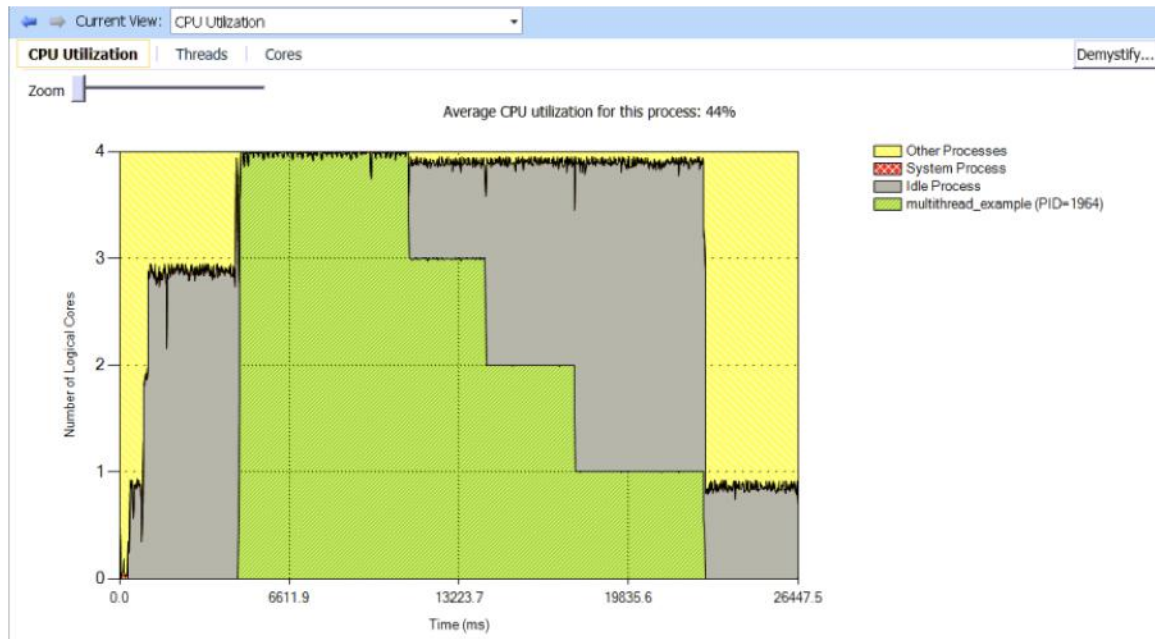
After the application is executed, traced and logged, the Concurrency Visualizer analyzes the regarded information and presents it visually in three views:

- **CPU Utilization View:** Describes system activity across all processors.
- **Threads View (Parallel Performance):** Describes the interactions between threads in your program.
- **Cores View:** Describes thread migration across cores.

For more information on the **Parallel Performance Analysis (PPA)** Concurrency Visualizer tool please visit the corresponding **MSDN** site: [http://msdn.microsoft.com/en-us/library/dd537632\(VS.100\).aspx](http://msdn.microsoft.com/en-us/library/dd537632(VS.100).aspx)

CPU Utilization View

Using an area graph, this view illustrates the identified variations, during the profiling run, for average core utilization by: the process being analyzed, the idle process, the System process, and other processes that are running on the system.

**Figure 1**

CPU Utilization View area graph

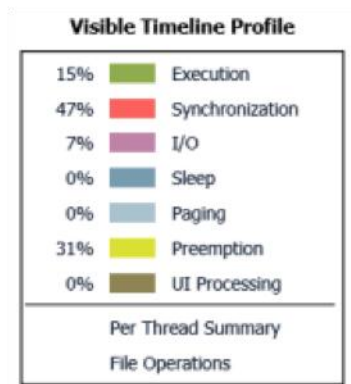
The CPU Utilization View is generated by dividing the profiling time into short time segments. For each segment, the average number of cores being utilized by the subject process(es) are plotted by computing the average number of process threads that are executing during each interval.

For more information on this view please visit the **msdn** site: [http://msdn.microsoft.com/en-us/library/dd627195\(VS.100\).aspx](http://msdn.microsoft.com/en-us/library/dd627195(VS.100).aspx)

Threads View (Parallel Performance)

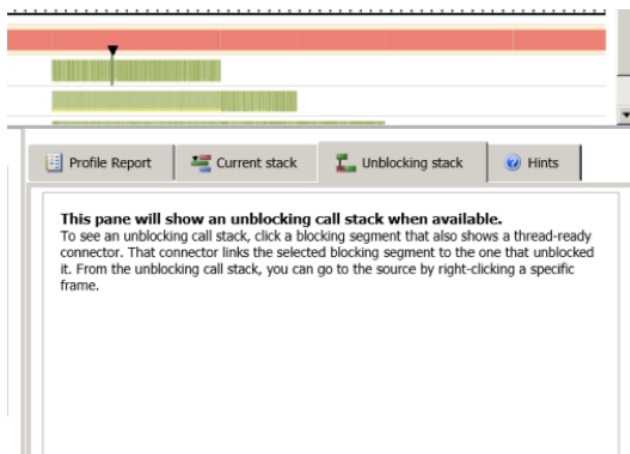
The Threads View is the most detailed and feature-rich view in the Concurrency Visualizer tool. You can identify where your application's threads are spending their time: whether executing application code, or blocking due to synchronization, I/O, or other reasons.

During profile analysis, all operating system context switch events per application thread are examined. Context switches can result from blocking on synchronization primitives, due to quantum expiration, or making blocking I/O requests. A category to every context switch is assigned, in order to give a good idea about why a thread has stopped executing. The categories are shown in the legend in the bottom left and are explained by their corresponding help topics.

**Figure 2***Threads View Legend*

The most accurate categorization of context switch events is achieved via analysis of the thread's call stack looking for well-known blocking APIs. In some cases, an "approximate" category is chosen due to practical limitations, such as the impossibility of cataloging all the blocking APIs that can exist in a Windows application. It is possible that the approximate category may be occasionally incorrect; however, via the call stack features of the tool, the user is able to identify the root cause of any blocking event by examining the call stacks that correspond to context switch events.

Another feature of this view is the ability to expose inter-thread dependencies. For example, if you identify a thread that is blocked on a synchronization object, the Threads View shows you the thread that unblocked it as well as what that thread was doing at that time, by showing its call stack when it unblocked the thread of interest.

**Figure 3***Threads View - Unblocking Stack*

When threads are executing, a sample profile is collected, so that users can also analyze which code is executed by one or more thread during an execution time window. In addition, to allowing interactive sampling of thread execution, this view provides call stack tree execution profiling and blocking reports.

For example, thread Blocking Details can be analyzed to understand the underlying reasons for thread blocking regions. The Threads View allows you to hover or select (by left mouse clicking) one of the regions you are interested on. A tooltip will be displayed with general information about the blocking event such as category, a blocking API if available, and blocking duration.

For the preemption category, this view shows the process id and thread id within the process that was scheduled on the CPU when the thread was stopped by the kernel. You can also select a blocking region in a channel of interest, which results in the display of the current stack in the bottom window.

In addition to the tooltip information, the “Current stack” tab shows the call stack that resulted in blocking your thread. By examining the call stack, you can determine the underlying reason for the thread blocking event.

By default, the complete call stacks is shown in this view, including user and kernel stacks. When the tool is able to identify a specific API as the responsible, the call stack is trimmed beyond that frame. If the tool is unable to determine the root function call that resulted in blocking, the entire call stack is exposed so the user can examine it and make that determination.

Since sometimes a path of execution can result in multiple blocking events. The Threads View provides a call-tree based profile report for each blocking category. It is valuable to understand the cumulative blocking delays organized by call stacks. You can view the profile by selecting one of the blocking category legend entries on the left.

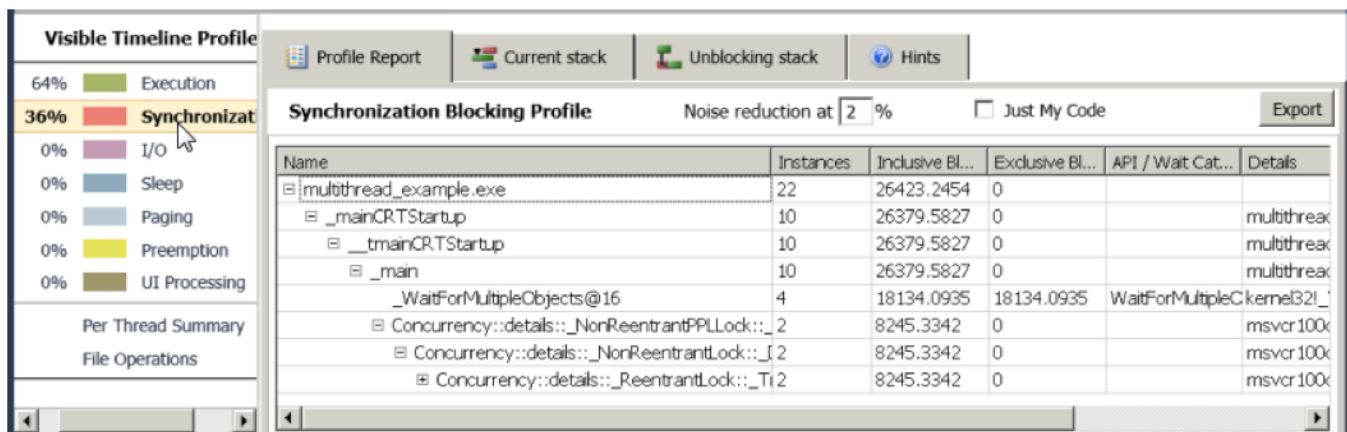


Figure 4

Threads View – Profile Report

These reports give you a quick way of prioritizing where you should invest your time performance tuning your application. For more information on usage and recommendations for this view please visit the **MSDN** site: [http://msdn.microsoft.com/en-us/library/dd627193\(VS.100\).aspx](http://msdn.microsoft.com/en-us/library/dd627193(VS.100).aspx)

Cores View

The Cores View contains a graph and a legend to show threads migration between cores. It is important to remember that core migration negatively affects performance. This view helps you to visualize where using threads is not helping the performance but generating a cross-core migration problem.

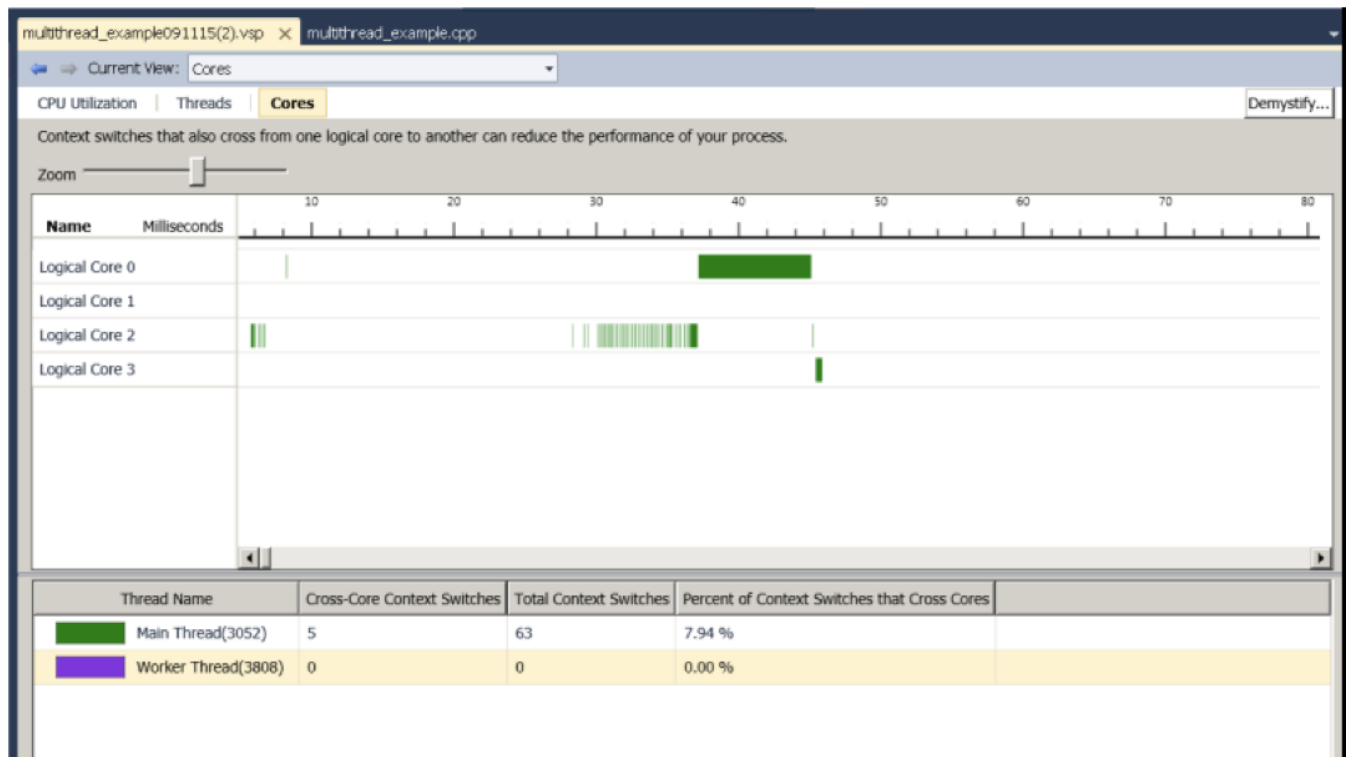


Figure 5
Cores View

Figure 5 above presents the logical cores on the y-axis and the time on the x-axis. You are able to zoom in and out the area you are interested in by highlighting the area and pressing CTRL plus the mouse wheel. Every thread has a unique color to identify each thread across cores over the time.

An entry for each color, on the bottom legend, shows the thread color and name, the number of cross-core context switches, the total context switches, and the percentage of context switches crossing across the cores.

For more information on usage and recommendations for this view please visit the **MSDN** site:

[http://msdn.microsoft.com/en-us/library/dd627197\(VS.100\).aspx](http://msdn.microsoft.com/en-us/library/dd627197(VS.100).aspx)

General tips

These three views can be accessed either through the profiler toolbar's Current View pull-down menu, through bitmap buttons at the top of the summary page, or through links in the views. The **active legend** (on the left pane of Figure 6) has multiple features. First, for every thread state category, you can click on the legend entry to get a callstack based report in the **Profile Report** tab summarizing where blocking events occurred in your application.

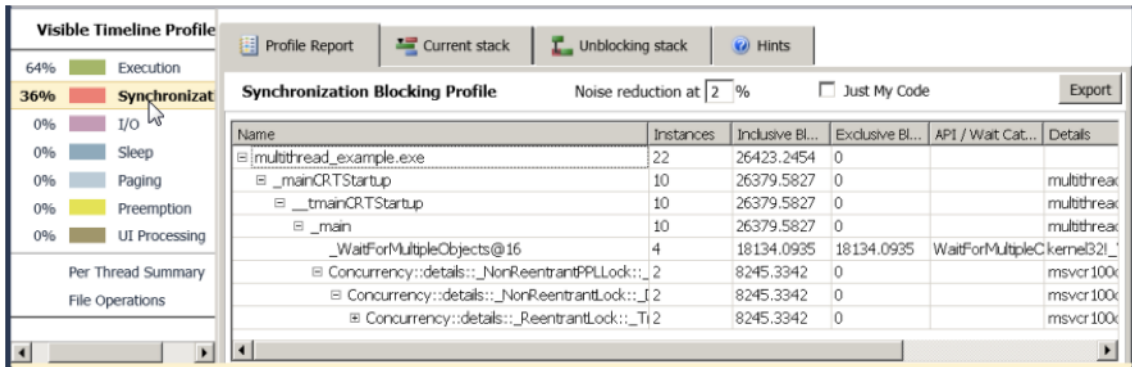


Figure 6
Profile Report - Callstack Based Report

For the execution category, you get a sample profile that tells you what work your application performed.

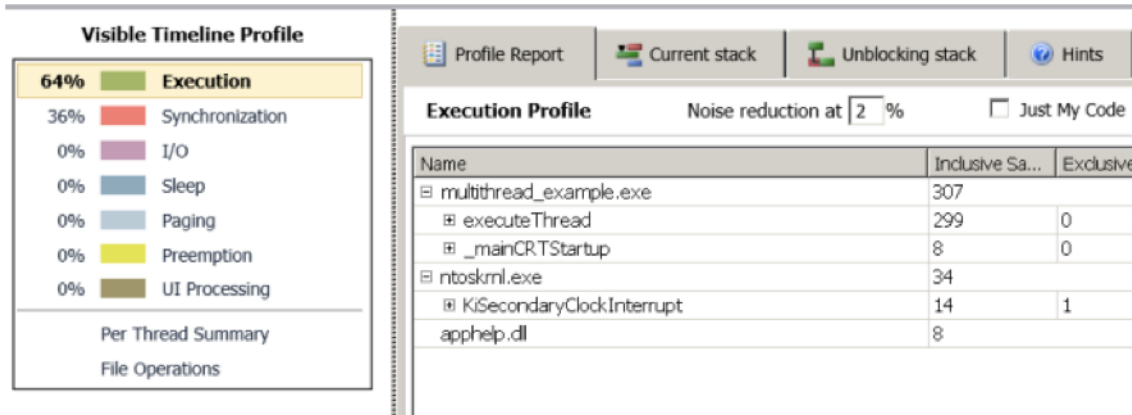


Figure 7
Profile Report –Execution Profile

As usual, all of the reports are filtered by the time range that you're viewing and the threads that are enabled in the view. You can change this by zooming in or out and by disabling threads in the view to focus your attention on certain areas. The legend also provides a summary of where time was spent as percentages shown next to the categories.

When you select an area in a thread's state, the "Current Stack" tab shows where your thread's execution stopped for blocking categories, or the nearest execution sample callstack within +/- 1ms of where you clicked for green segments.

When you select a **blocking category**, whenever it is possible, a link is drawn (dark line) to the thread that resulted in unblocking your thread. In addition, the Unblocking Stack tab shows you what the unblocking thread was doing by displaying its callstack when it unblocked your thread. This is a great mechanism to understand thread-to-thread dependencies.

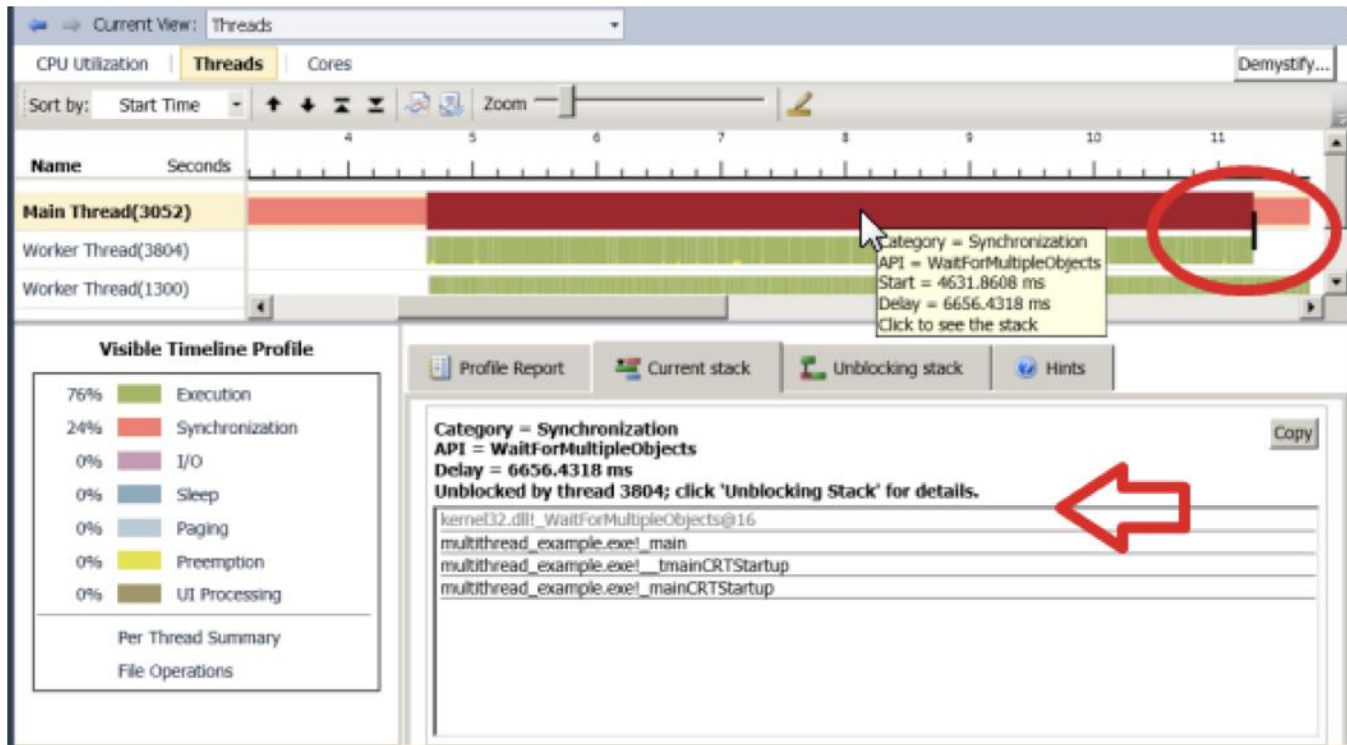


Figure 8

Blocked thread "Current Stack" tab and link(dark line) to unblocking thread

The Per Thread Summary report can be accessed from the active legend. This report is a guide that helps you understand improvements/regressions from one run to another and serves as a guide to help focus your attention on the threads and types of delay that are most important in your run. This is valuable for filtering threads/time and prioritizing your tuning effort.

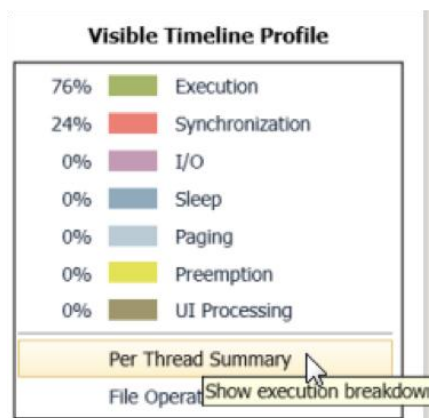
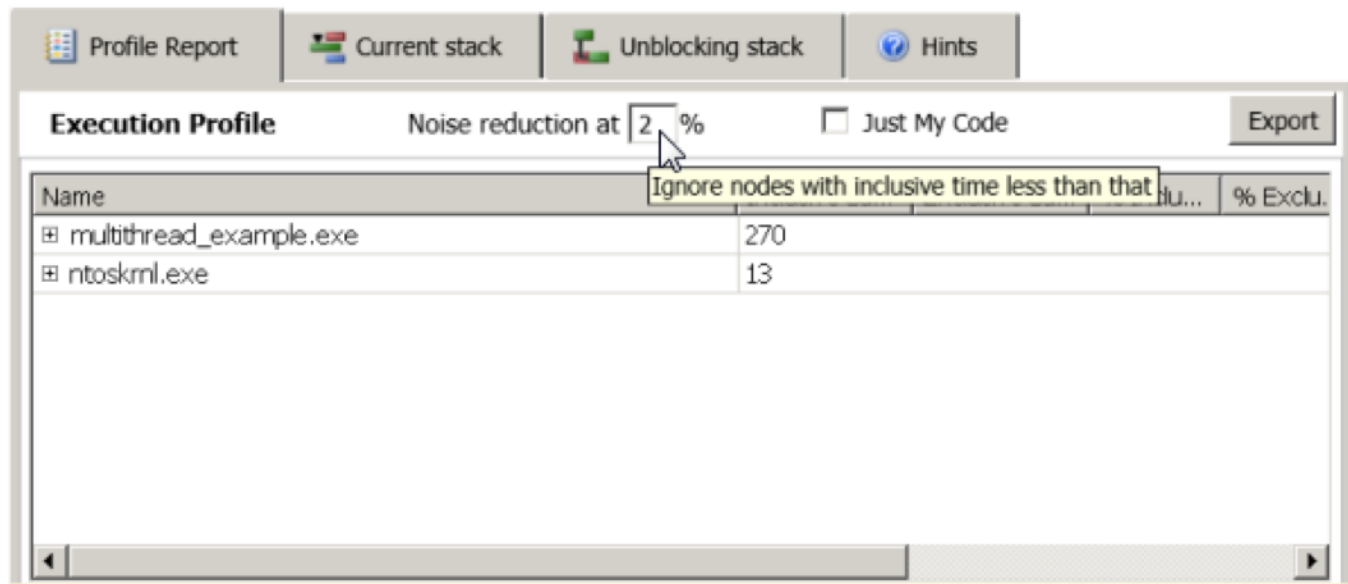


Figure 9

Active Legend – Per Thread Summary

In order to minimize noise, the profile reports filter out the callstacks that contribute to a reduced blocking time. You can change the noise reduction percentage if you desired a different level than the default. You can also remove stack frames that are outside your application from the profile reports.

**Figure 10**

Profile Report – Noise Reduction

Exercise 1: Working with the Concurrency Visualizer

This exercise will show you how to use the **Visual Studio 2010 Beta 2** release to analyze a multithreaded application. We will work with the different views available in the **Parallel Performance Analysis (PPA) Concurrency Visualizer**. In order to obtain the analysis reports, we will work with a demo that utilizes 4 threads. When the four threads finish executing, the main thread will also finish. We will use the different PPA Concurrency Visualization views to analyze the results.

Part 1: Creating the reports

1. Log onto the **lab machine with the credentials that were provided.**

2. Navigate and expand Visual Studio 2010 folder under:

Start -> All Programs -> Microsoft Visual Studio 2010

3. **Right click** on the **Microsoft Visual Studio 2010 – ENU** executable and select **Run as Administrator.**

4. Once Visual Studio is opened, select:

File -> Open -> Project/Solution...

5. Select the demo application on the following path:

C:\Server 2008 R2 Labs\Introduction to the PPA Analyzer\PPAmultithread_example

6. Click Open.

7. From the Solution Explorer, in Visual Studio 2010, click on the **multithread_example** project to expand the node.

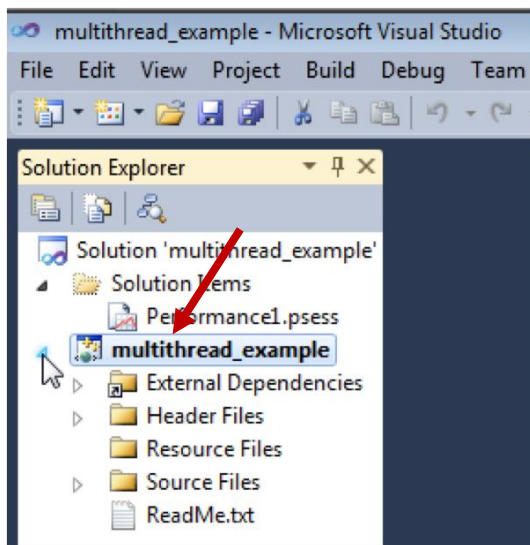
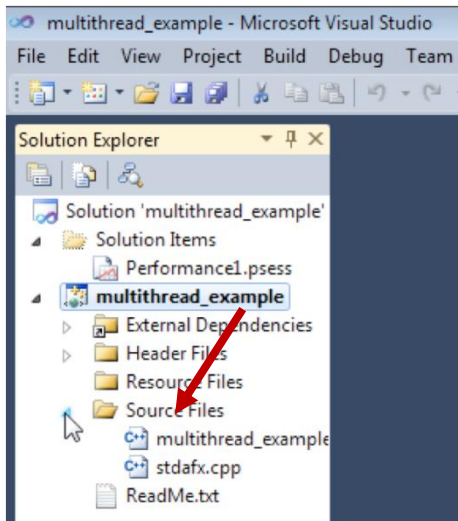


Figure 11

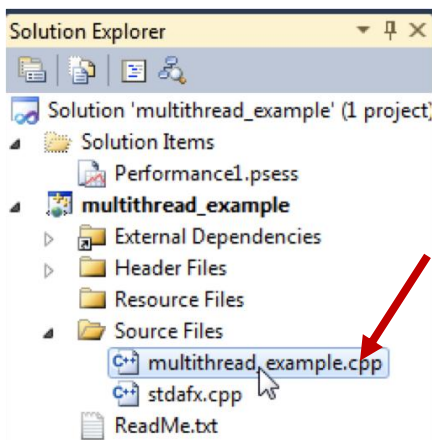
Solution Explorer - multithread_example project

8. Click on the **Source Files** node to expand it.

**Figure 12**

Solution Explorer - Source Files node

9. Double click on the **multithread_example.cpp** source file to open the project that we are going to analyze:

**Figure 13**

Solution Explorer - multithread_example.cpp source file

Note: The displayed code contains a main thread that creates four threads, after waiting for the user to press a key. Each thread executes a mathematical operation a determined million of times depending on their creation order. The first thread will calculate the operation twenty-five million times. The second thread will execute the calculation thirty-six million times. The third thread will execute forty-nine million times. The fourth thread will execute the math calculation sixty-four million times. All threads start at the same time. This way, the first thread will finish before the second thread; the second thread will finish before the third, and so on.

This application logic makes the application to require most of the host machine CPUs resources power at the beginning stages of the execution, freeing them sequentially.

10. Compile the project by selecting **Build Solution** from the **Build** main menu.
11. From the **Analyze** Menu, select **Launch Performance Wizard...**

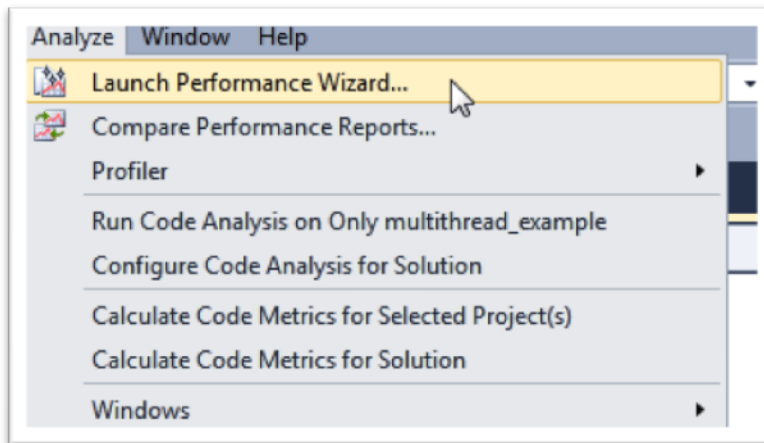


Figure 14

Analyze Menu - Launch Performance Wizard...

Note: The dialog below will be displayed:

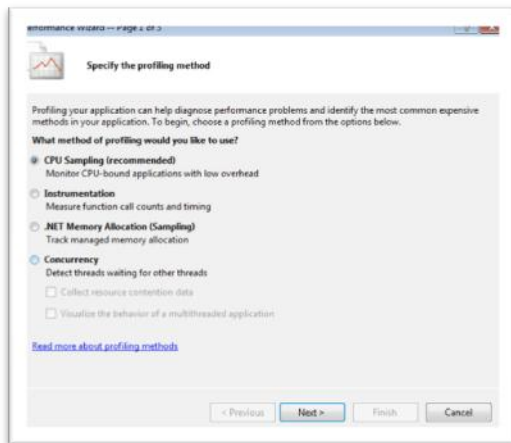
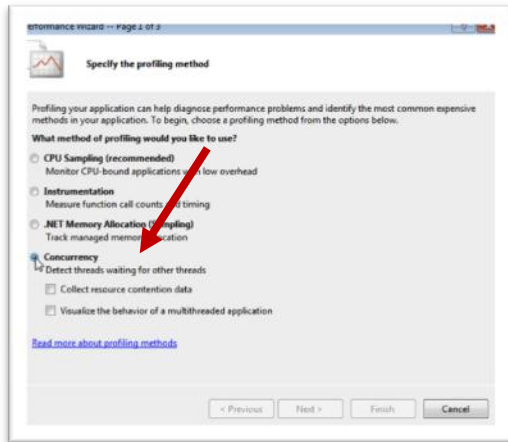


Figure 15

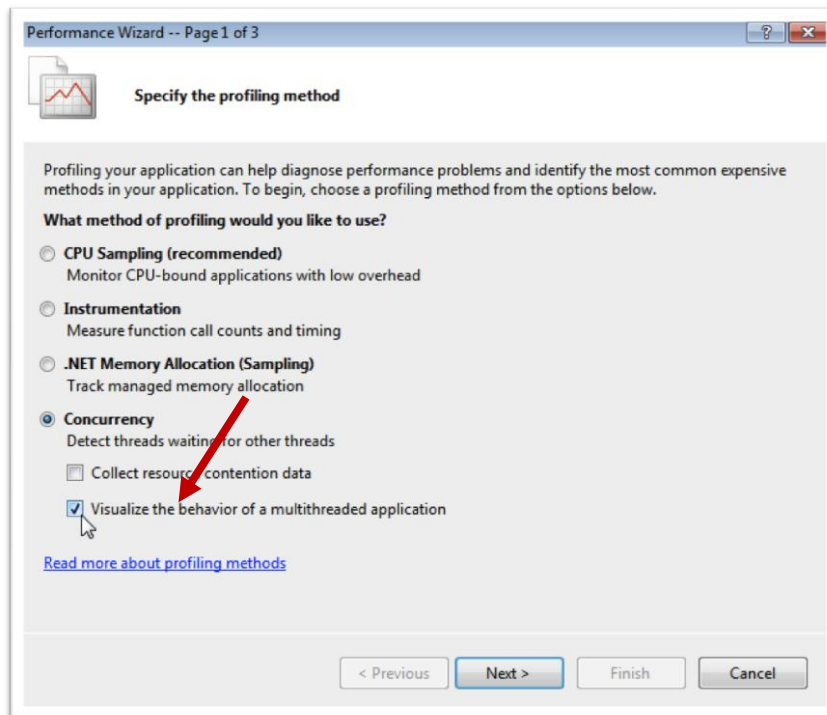
Performance Wizard

12. Select the **Concurrency** option; the two checkboxes will be enabled.

**Figure 16**

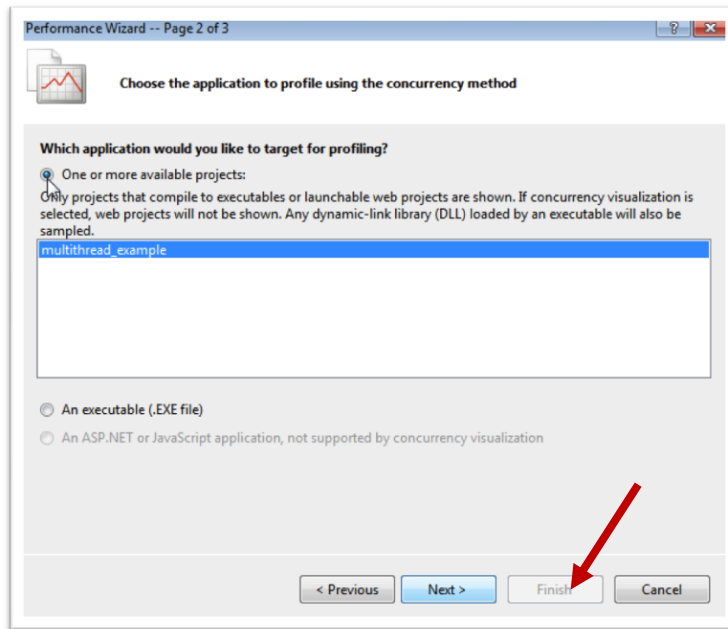
Performance Wizard - Concurrency option

13. Select **Visualize the behavior of a multithreaded application** and click **Next**:

**Figure 17**

Performance Wizard - Visualize the behavior of a multithreaded application option

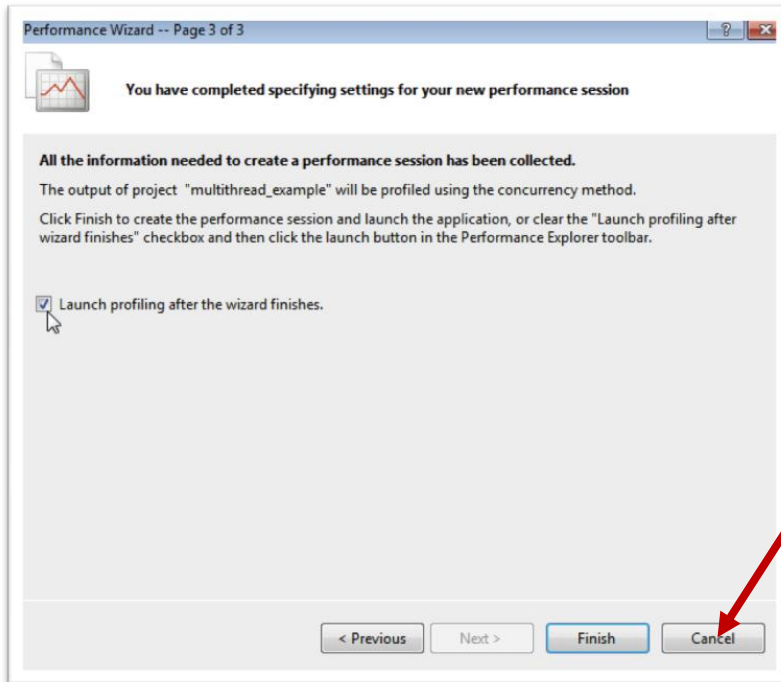
14. Leave the default option shown below and click **Next**:

**Figure 18**

Performance Wizard – One or more available projects option

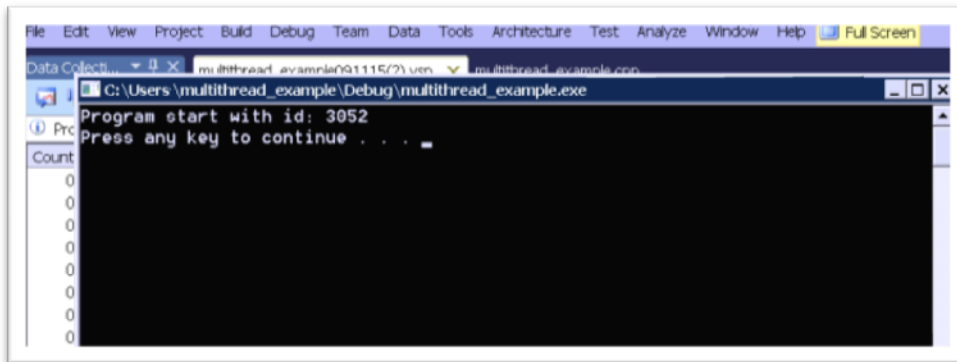
Note: This window allows you to: profiling executable files or profiling the code you have currently opened on the Visual Studio.

15. On the final screen, make sure the **Launch profiling after the wizard finishes** checkbox is selected and click **Finish**.

**Figure 19**

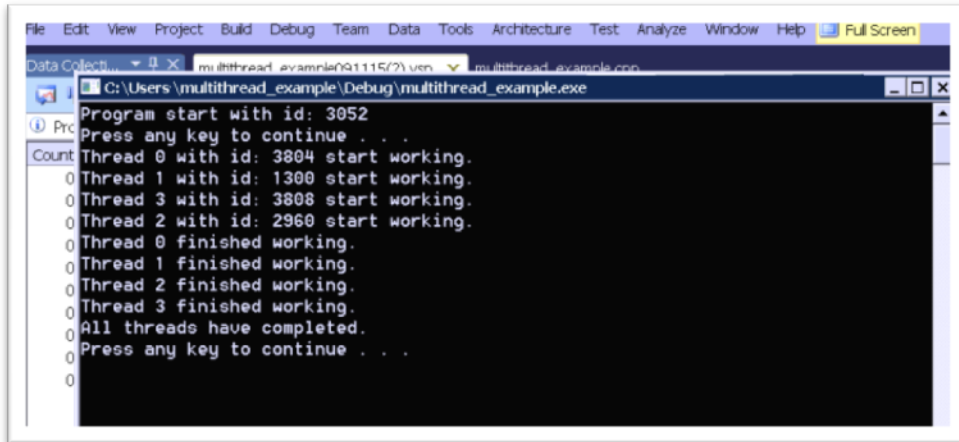
Performance Wizard – Launch profiling after the wizard finishes option

Note: The program will start executing.

**Figure 20**

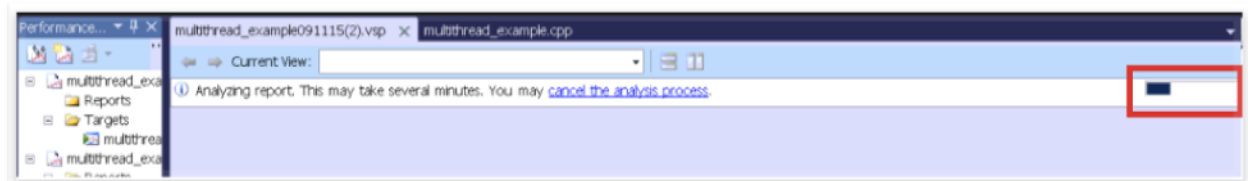
Application Starts

16. Press **Enter** to execute the application:

**Figure 21**

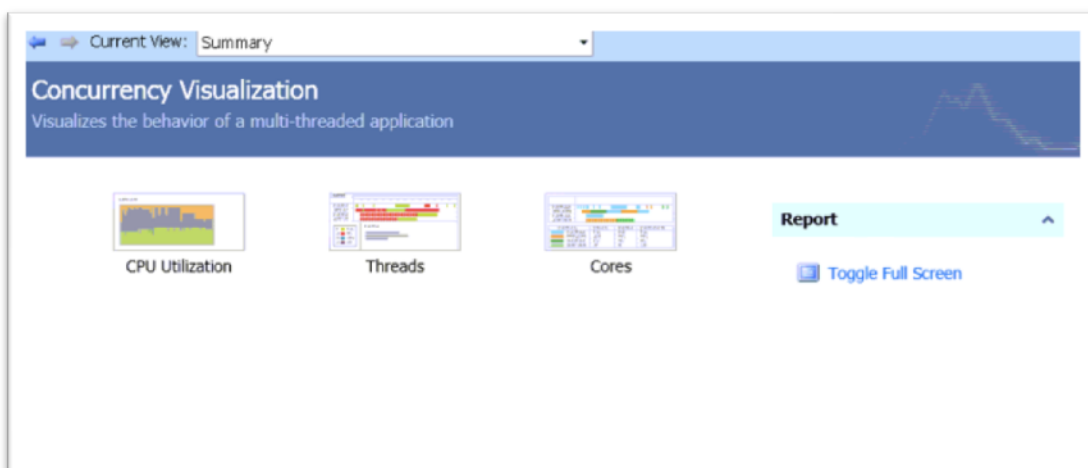
Application – Waiting for the user to press a key

Note: During the execution of your application the PPA collects the necessary information for the reports. After the application finish running, PPA analyzes the information to create a visual representation of the application characteristics:

**Figure 22**

PPA analyzing the collected information

Note: Once the profiling process finishes, the following visualization views will be presented:

**Figure 23**

Concurrency Visualization Views

17. Click on **Toggle Full Screen** in order to full-screen the **Concurrency Visualization** view.

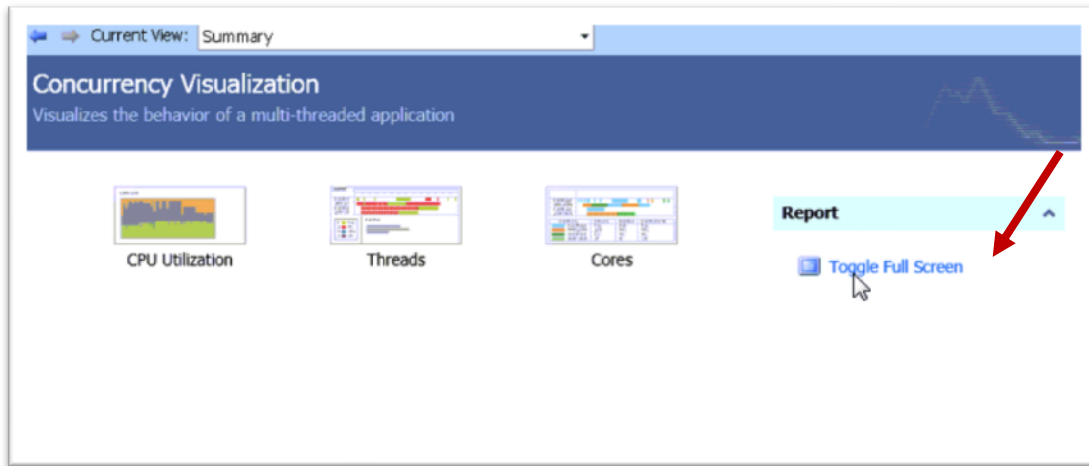


Figure 24

Concurrency Visualization – Toggle Full Screen

Part 2: CPU Utilization / Concurrency Analysis

Note: The **CPU Utilization** view is intended to tell you how well you use the CPU resources over time, and how much of the CPU available resources are being used by your application.

1. Click on **CPU Utilization** icon:

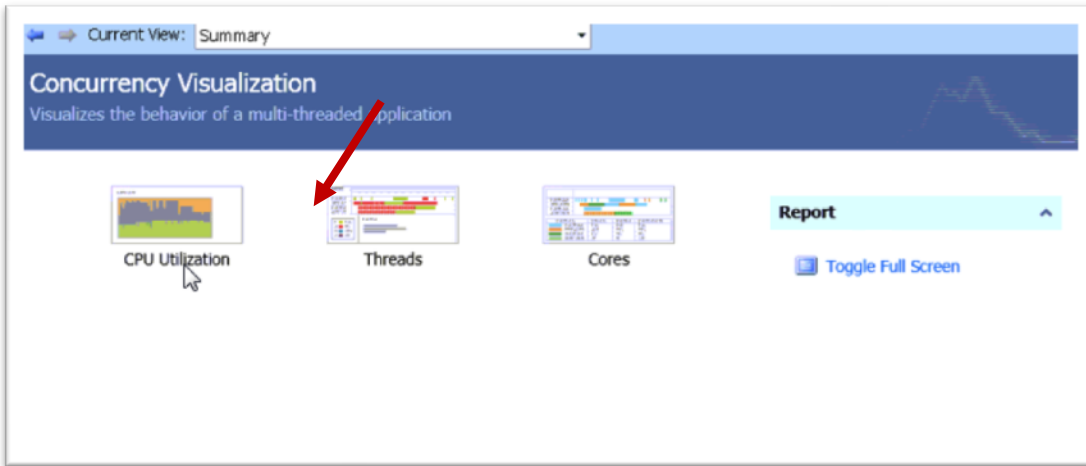


Figure 25

Concurrency Visualization – CPU Utilization

Note: A chart of the CPU Utilization versus the time over the life of your application execution will be shown.

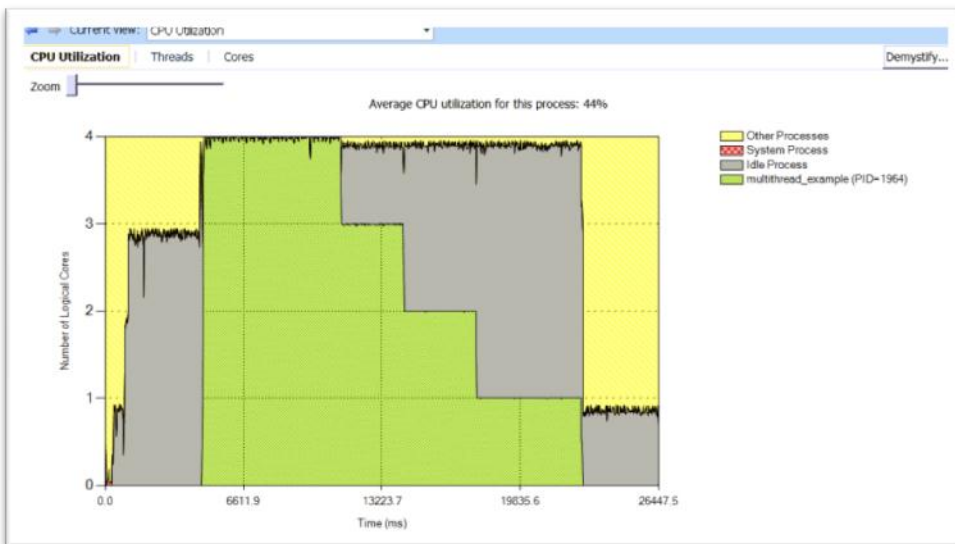


Figure 26

CPU Utilization View - CPU Utilization versus time chart

Note: From this chart you can appreciate that the main colors are green, grey and yellow. According to the legend on the left of the chart, the yellow color represents the execution of other processes on the CPU; green color represents your application execution; and gray color represents idle time.

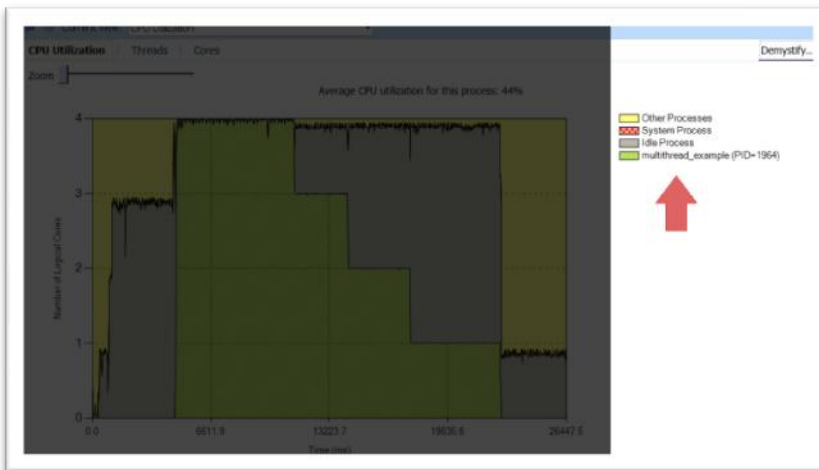


Figure 27

CPU Utilization View - Legend

Note: Based on this chart it appears that this demo application is doing a reasonable job of parallelism that uses most of the cores in this system. You can also appreciate how after some time, less cores are used by the application. These results match completely with our application logic,, since our demo application threads are finishing sequentially. You can appreciate this sequence in the graphic below:

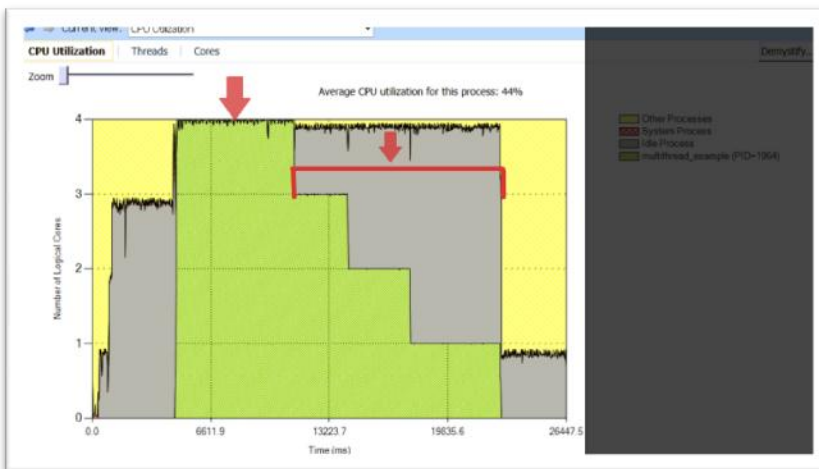


Figure 28

CPU Utilization View –Threads finished sequentially

Note: It is important to take in account that when we talk about logical cores they don't really matches with the physical cores, so from this example we cannot say that core 1 was in use the whole time. This Number of Logical Cores is more a metric of how much of the overall compute power you are using.

With that in mind, you can see that on the first part of the execution there is idle time, while the application waits for the Enter key. Then, the application creates four threads which start using the CPU's overall process power. This is a very good sign that indicates that most of the available CPU resources are being exploited. As long as the threads start finishing their work, the CPU utilization goes down to 3 logical cores, 2 logical cores and finish with idle process when the four threads finished their work and the application starts waiting for the final Enter key press.

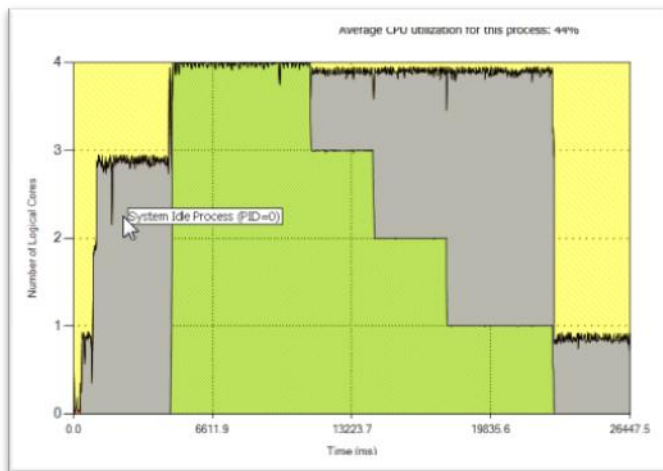


Figure 29

CPU Utilization View - Logical Cores Utilization

Note: You may have noticed that there is yellow color along the whole application execution that indicates that the process is sharing CPU with other processes on the system and some preemption takes place.

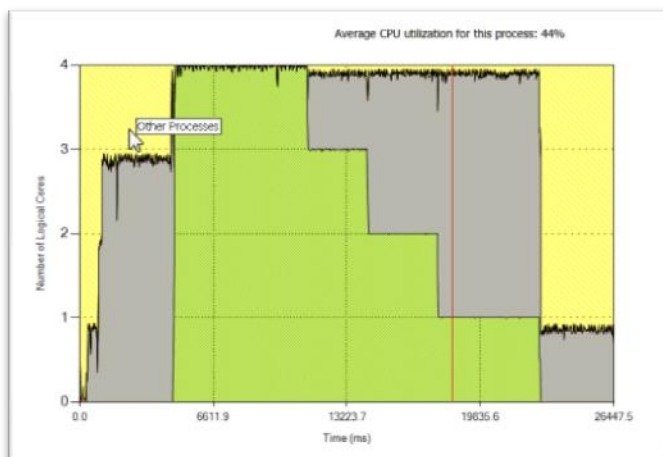


Figure 30

CPU Utilization View – Process Preemption

Note: Looking at this view you get a visual confirmation of the fact that we are exploiting more than just one core on the CPU and the fact that on some important time we are exploiting most of the cores. This is a very good sign when you are trying to parallelize your code, in order to make sure that this visualization matches with your mental concept of the applications behavior.

Part 3: Thread Blocking Analysis

1. From the current view, click on **Threads** view.

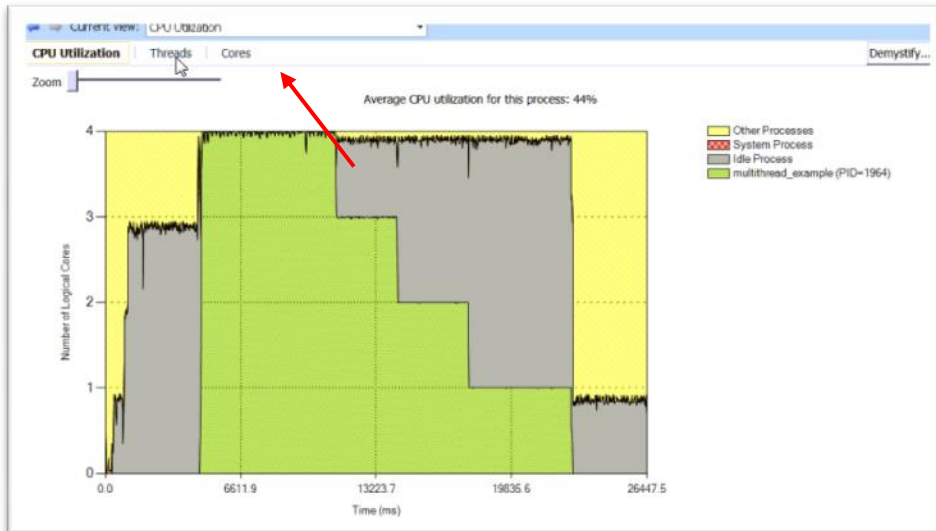


Figure 31

Selecting Threads View

Note: This will open the **Threads** view, as shown in the figure below.

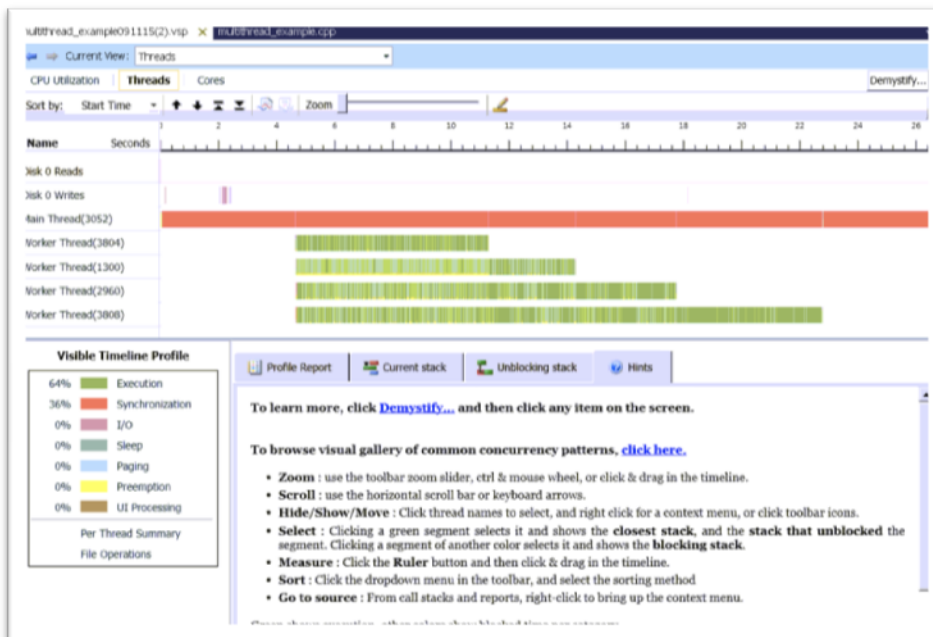


Figure 32

Threads View

Note: This view shows time going along the X-axis and we have each individual thread listed along the Y-axis. As we see in previous steps our program consists of a main thread which during execution creates four other threads.

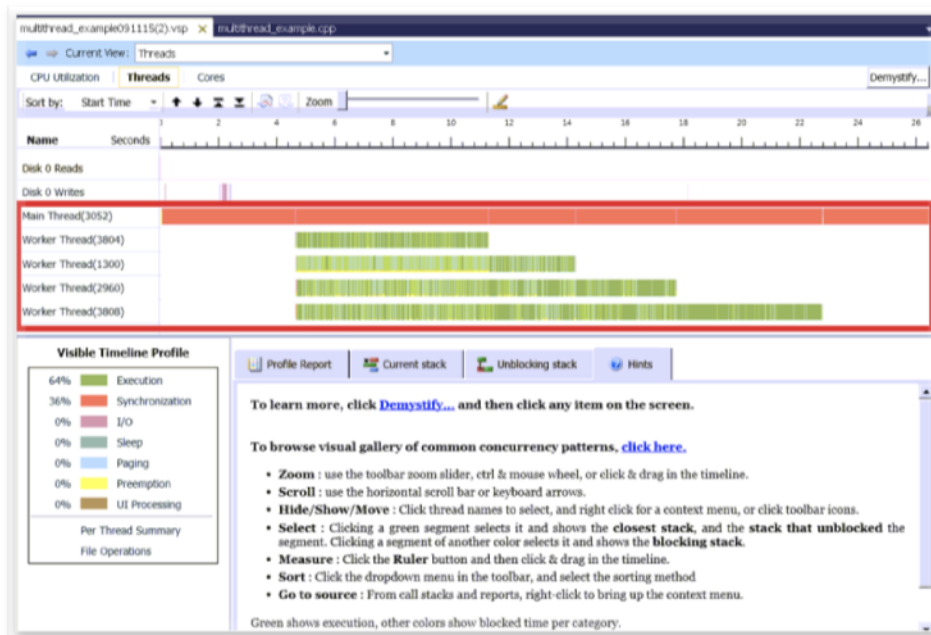


Figure 33

Threads View – Application threads

Note: There are other threads listed on the report, for example: system processes. But we are interested only in the five threads of our application so you can hide the other threads.

2. Right-click over the **Disk 0 Writes** thread.

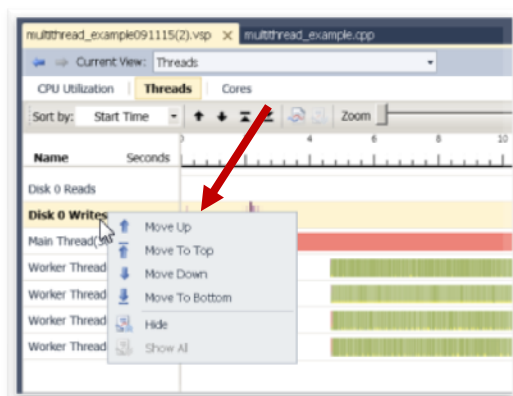
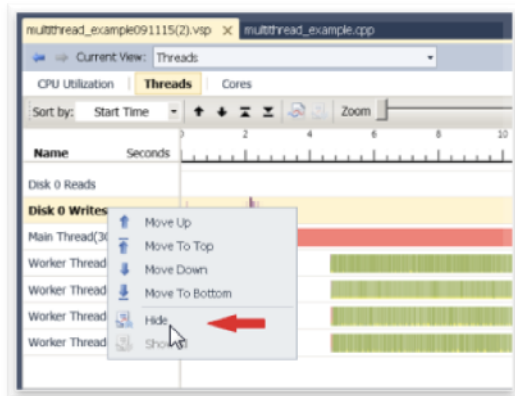


Figure 34

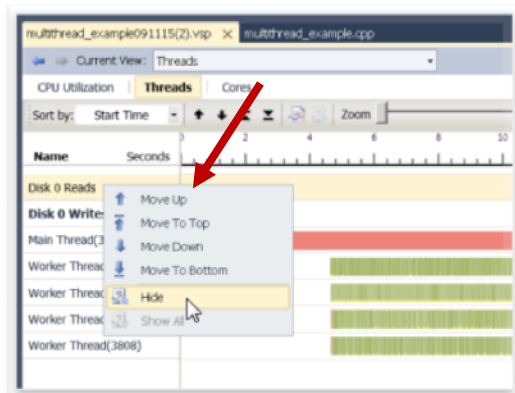
Threads View – System processes

3. Select the **Hide** option:

**Figure 35**

Threads View – Hiding Disk 0 Writes thread

4. Right-click over the **Disk 0 Reads** thread and select **Hide**:

**Figure 36**

Threads View – Hiding Disk 0 Reads thread

Note: If you have extra threads that not belong to our application you can also hide them.

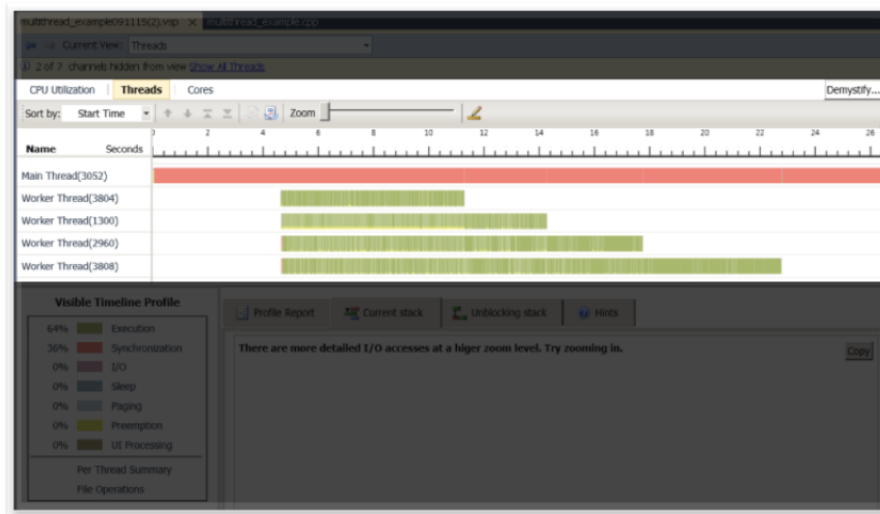


Figure 37

Threads View – Demo Application threads

Note: The legend on the left has information on what do each of the colors represent.

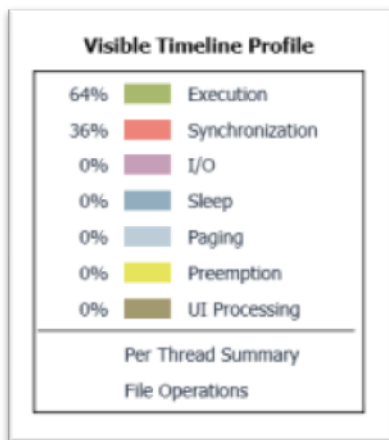


Figure 38

Threads View – Active Legend

Note: You can see in the chart that the four worker threads are mostly in green which is useful execution for our program. The yellow color stands for preemption. Going back to the CPU utilization chart there could be other processes not related to our program that could result in this preemption. On this chart we can appreciate that the main thread is on synchronization most of the time.

At any time you can click on the **Zoom** bar to analyze in more detailed any thread information.

- Click on the **Zoom** bar to analyze the beginning of the main thread in detail, and obtain more information (Zoom in until you can distinguish the colors shown in the following Figure):

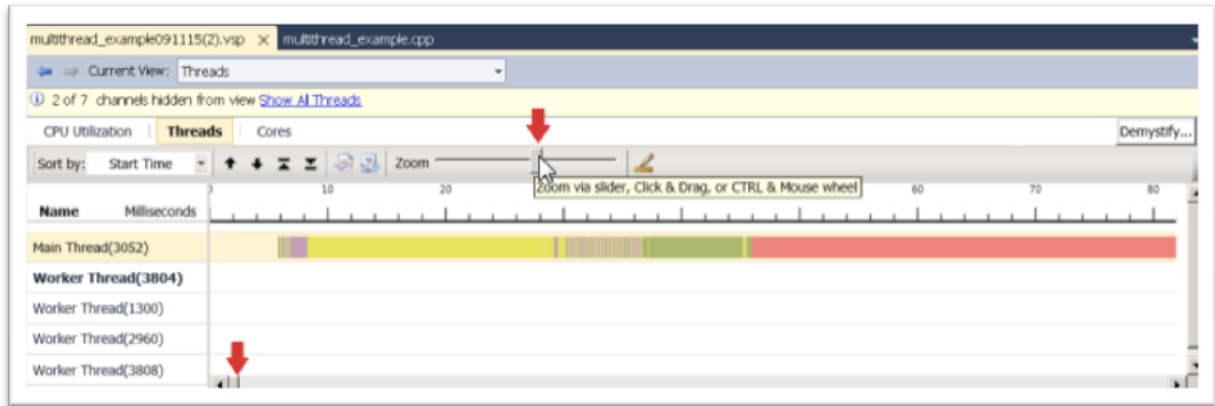


Figure 39

Threads View – Main thread zoomed in

Note: If necessary use the **horizontal scroll** to find the beginning of the main thread execution.

6. Notice how the legend on the down left changes data according to what is now being visualized:

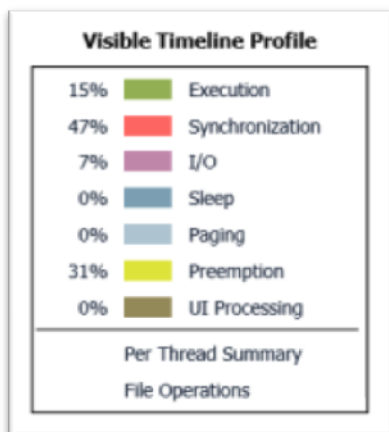


Figure 40

Threads View – Active Legend **Note:** On this view you can appreciate the main thread execution beginning, in green. Next, we get the input/output period in violet color. This is when the application waits for the user to press the Enter Key. During this time, we also have some preemption, in yellow color. Then the application executes again in order to create the four threads, and enter into the Synchronization period, in red color. This is when the threads start executing, and the main thread only waits for them to finish their respective jobs.

Note: Additionally, the blue color would be used to represent sleeping periods on the threads, if any. Light blue would represent Paging. And, brown would represent UI Processing. At any time you could use the mouse to click on these legend elements to obtain a detailed report in the **Profile Report** field:

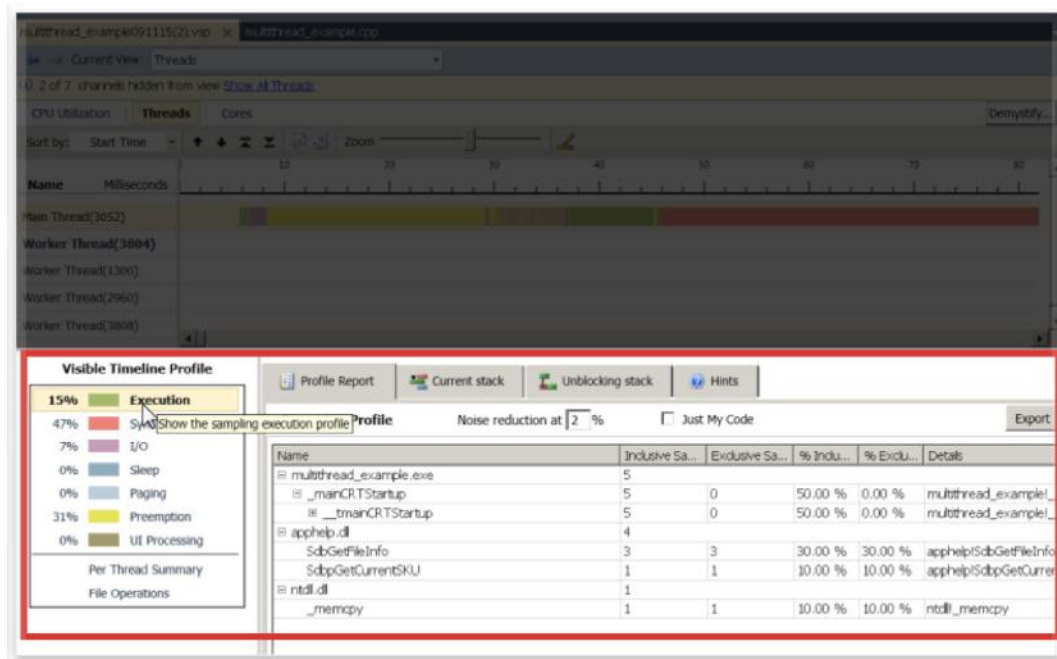


Figure 41

Clicking on Active Legend elements opens detail reports

Note: With all this information, this chart helped us to confirm that our application is really being parallelized. With this view we can identify possible optimizations for the analyzed code or confirm that the mental concept is being reproduced on the application.

Part 4: Core Execution / Thread Migration

- Next, click on the **Cores** view:

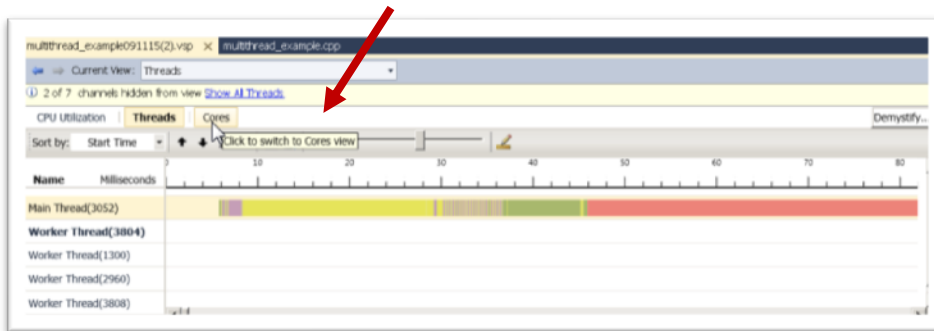


Figure 42

Selecting Cores View

Note: This will show the **Cores Execution View** of the report:

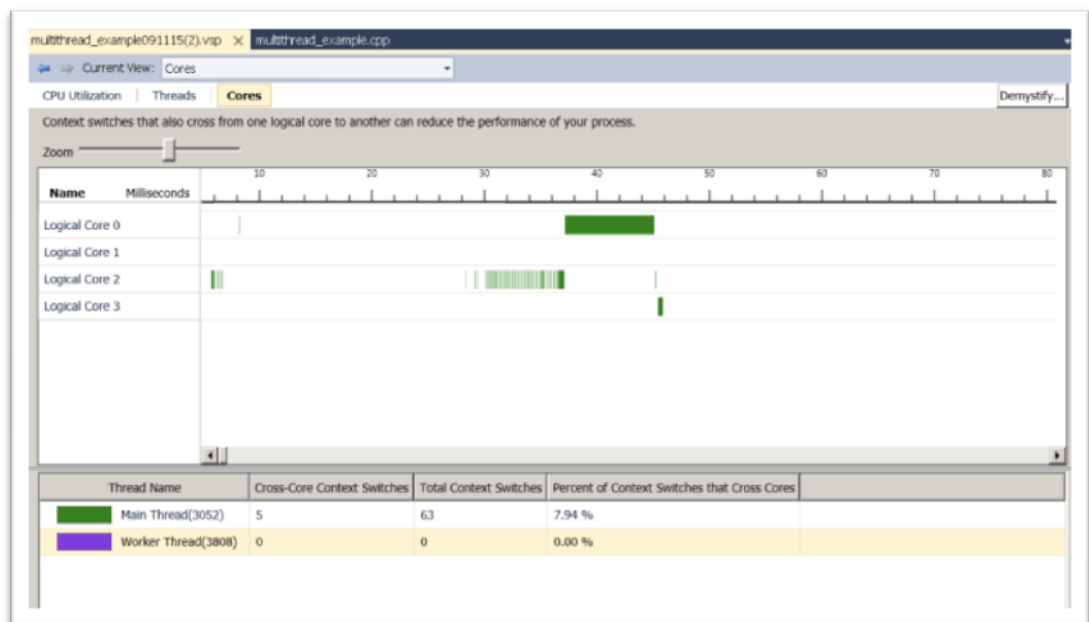


Figure 43

Cores Execution View

Note: On this chart, we have time going on the X-axis, and along the Y-Axis the four cores are listed. For each of these cores you can see which thread was running on that core at any particular moment in time. Notice that this green bars corresponds to the **Main Thread** execution.

- Use the **Zoom** bar to completely zoom out the chart.



Figure 44

Cores Execution View zoomed out

Note: This view helps us diagnose thread scheduling bugs and other thread migration issues.

Looking at this chart, we can see how the threads were executed on each core. For example, orange color represents a working thread running on it (Thread id 2960 on this particular execution, according to the legend). This thread seems to have been running on logical core 0 for a good amount of time. But then, it transits to logical core one, mainly, and also you can find this thread being execute in particular moments on logical core three and four.

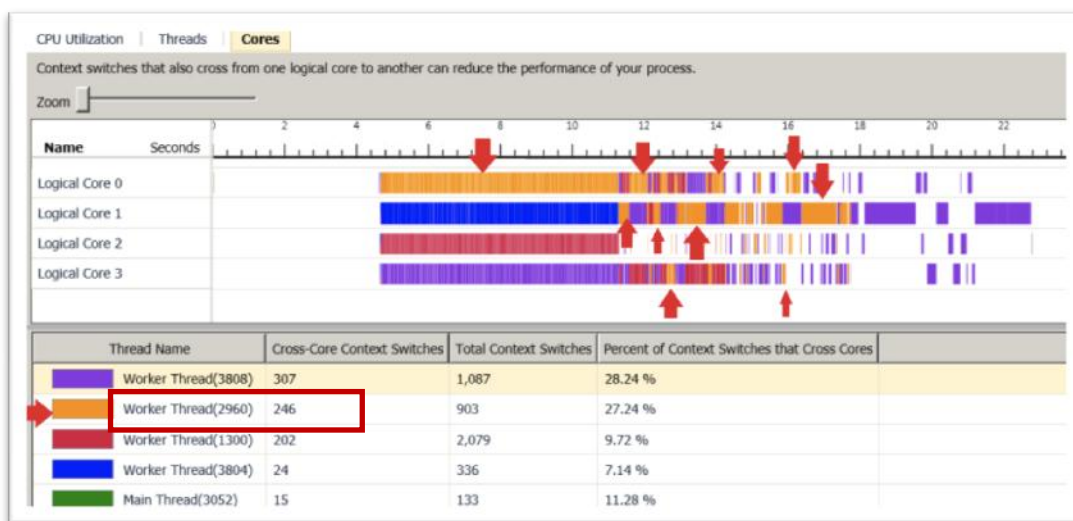


Figure 45

Cores Execution View – Worker thread transits

Note: According to this report, the four applications threads distribution over the four cores was very good, at the beginning, where for the first seconds each thread was running on a particular core. Then, it seems that after second eleventh, the threads started to bounce around between all four logical cores. This is something interesting to look at if you are worried about thread scheduling bugs.






Thread Name	Cross-Core Context Switches	Total Context Switches	Percent of Context Switches that Cross Cores
 Worker Thread(3808)	307	1,087	28.24 %
 Worker Thread(2960)	246	903	27.24 %
 Worker Thread(1300)	202	2,079	9.72 %
 Worker Thread(3804)	24	336	7.14 %
 Main Thread(3052)	15	133	11.28 %

Figure 46

Cores Execution View – context switches Statistics

Note: This analysis indicates that there were lots of **context switches** that cross from logical cores. These context switches that cross from one logical core to another reduce the performance of the process. Possible code optimizations could be studied here, thanks to these PPA reports.

Extra Credit

The **MSDN** site provides Code samples for the Concurrency Runtime and Parallel Pattern Library on Visual Studio 2010 that you can use to test your new abilities on the Parallel Performance Analysis tools:

<http://code.msdn.microsoft.com/concrtexttras/Release/ProjectReleases.aspx?ReleaseId=2776>

One of those code samples is an implementation of the five dining philosophers. The five dining philosophers problem is one of the most interesting illustration of the deadlock problem. The scenario describes five philosophers sitting at a round table. In front of each philosopher is a bowl of rice. Between each pair of philosophers is one chopstick. Before each philosopher can take a bite of rice, he must have two chopsticks; one taken from his left, and one taken from his right. The philosophers must find some way to share chopsticks such that they all get to eat. Analyze the behavior of this application with the techniques learned on the lab, using the code located on the following path:

C:\Server 2008 R2 Labs\Introduction to the PPA Analyzer\DiningPhilosophers_example

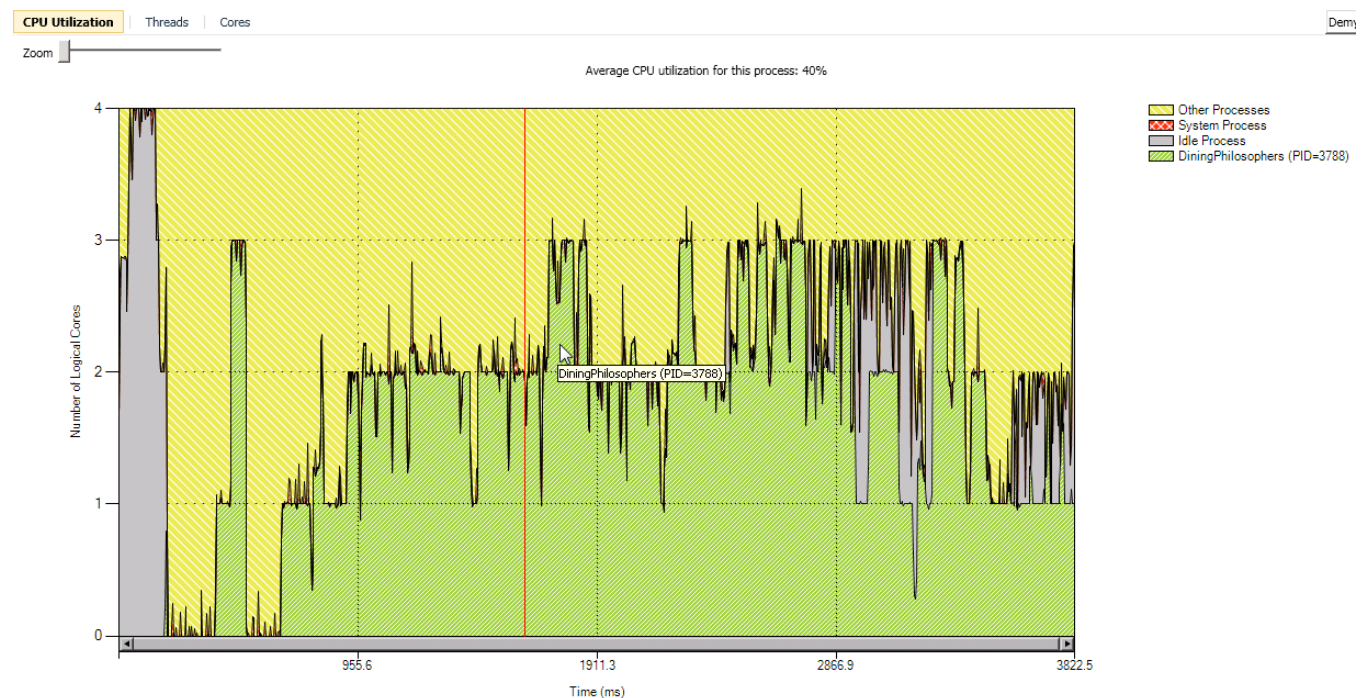


Figure 47

Dining Philosophers - CPU Utilization View



Figure 48
Dining Philosophers - Threads View

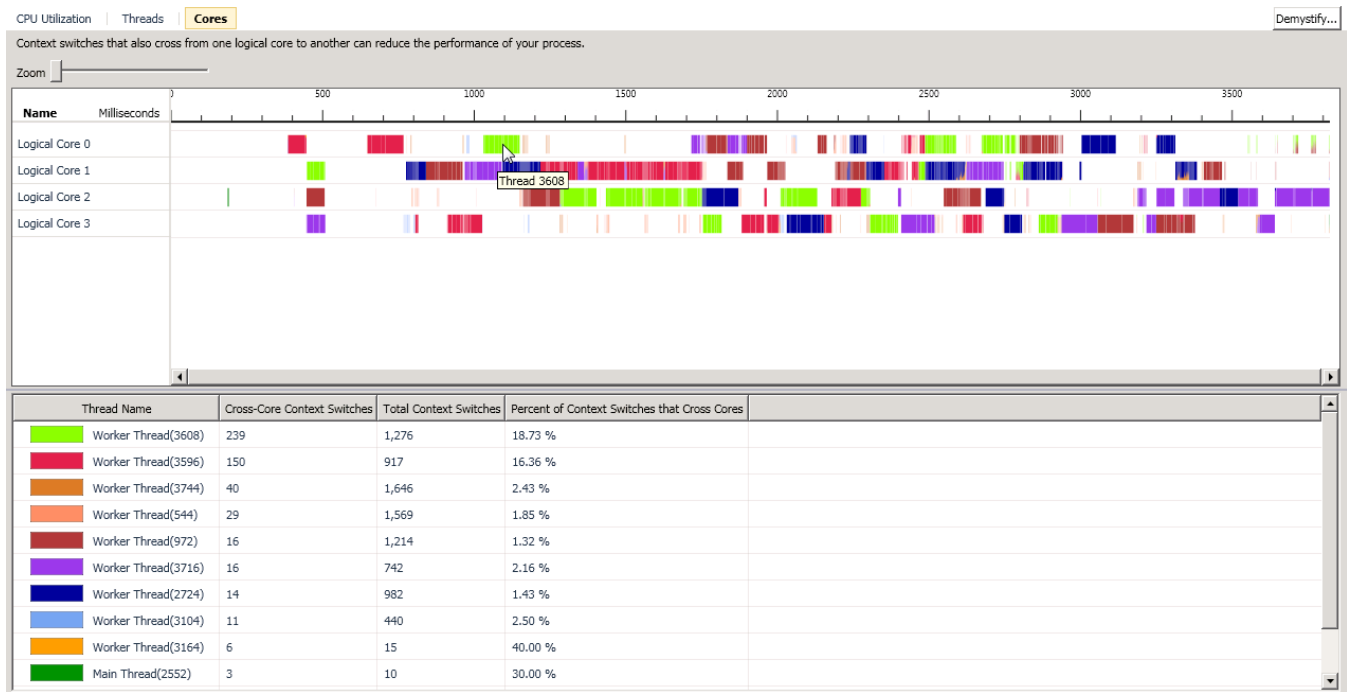


Figure 49
Dining Philosophers - Cores View

Lab Summary

The PPA Concurrency Visualization Tools included with Visual Studio 2010 allow us to visualize our mental concept of an application into a friendly visual representation of the integration our application has with the low level operating system politics. This makes it possible for us to easily see the what is going on “under the hood” when the operating system preempts or assigns core processor time to our application.

CPU Utilization View visually illustrates core utilization variations of our processes, the idle process, the System process, and other processes that are running on the system. This view helps us to quickly and easily find out if our application is making the best utilization of the CPUs on a machine.

The Cores view allows us to understand how application threads are scheduled on each logical core. This view eases makes it possible to visually understand the possible serialization over cores resulting from inappropriate threads use. If one of your goals in performance optimization is minimizing context switches, then this view will be of great use to you.

The threads view provides us with rich information about how our threads interacts between each other, with the operating system, and with other threads that execute at the time of our application. This view helps us to understand how the blocking threads wait for each other to release limited resources. Threads view helps us identify possible optimizations for the analyzed code or confirm that the mental concept is being reproduced on the application. When we are interested in performance improvements, this is going to be our main tool.

Being highly integrated with Visual Studio 2010, the Concurrency Visualizer tools efficiently helps us to locate performance bottlenecks, CPU underutilization, thread contention, thread migration, synchronization delays, and areas of overlapped I/O.