Appendix B: Debugging and Profiling Parallel Applications

The Visual Studio 2010 debugger includes two windows that assist with parallel programming: the Parallel Stacks window, and the Parallel Tasks window. In addition, the Premium and Ultimate editions of Visual Studio 2010 include a profiling tool. This appendix gives examples of how to use these windows and the profiler to visualize the execution of a parallel program and to confirm that it's working as you expect. After you gain some experience at this, you'll be able to use these tools to help identify and fix problems.

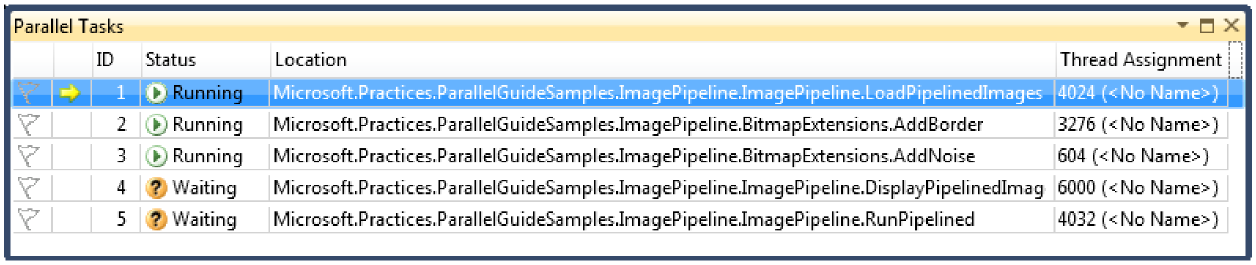
The Parallel Tasks and Parallel Stacks Windows

In Visual Studio, open the parallel guide samples solution. Set the ImagePipeline project from chapter 7 to be the Startup Project. Open **ImagePipeline.cs** and find the **LoadPipelinedImages** method. This is the method executed by the first task in the pipeline. Insert a breakpoint at the first statement in the body of the **foreach** loop. This is the loop that reads images from disk, and fills the pipeline as it iterates over the images.

Start the debugging process. You can either press F5, or, on the **Debug** menu, click **Start Debugging**. The ImagePipeline sample begins to run and opens its graphical user interface window on the desktop. Select the **Pipelined** option, then click **Start**. When execution reaches the breakpoint, all tasks stop and the familiar **Call Stack** window appears. From the **Debug** menu, point to **Windows**, then click **Parallel Tasks**.

When execution first reaches the breakpoint, the Parallel Tasks window shows that the first pipeline task, **LoadPipelinedImages**, is running and all the other tasks are waiting. This is because there are no images in the pipeline yet. Pressing F5 several times (or from the **Debug** menu, clicking **Continue**) causes several images to be loaded, so the pipeline starts to fill and other tasks can run. This is shown in Figure 1. Recall that each task runs in a thread. The Parallel Tasks window shows the assignment of tasks to threads. More than one task can run in a thread when task inlining occurs, so it's possible that a thread may be running when one of its tasks is blocked.

The Parallel Tasks Window



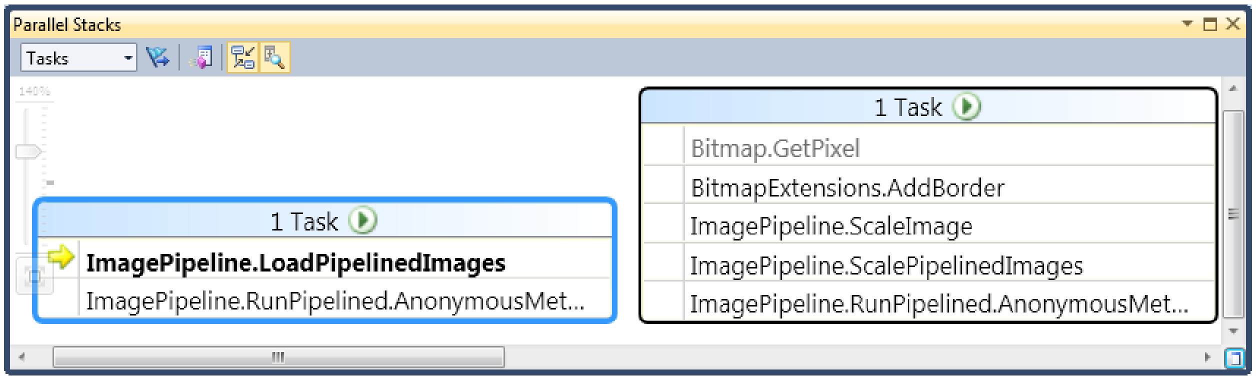
Parallel Tasks

ID	Status	Location	Thread Assignment
1	Running	Microsoft.Practices.ParallelGuideSamples.ImagePipeline.ImagePipeline.LoadPipelinedImages	4024 (<No Name>)
2	Running	Microsoft.Practices.ParallelGuideSamples.ImagePipeline.BitmapExtensions.AddBorder	3276 (<No Name>)
3	Running	Microsoft.Practices.ParallelGuideSamples.ImagePipeline.BitmapExtensions.AddNoise	604 (<No Name>)
4	Waiting	Microsoft.Practices.ParallelGuideSamples.ImagePipeline.ImagePipeline.DisplayPipelinedImag	6000 (<No Name>)
5	Waiting	Microsoft.Practices.ParallelGuideSamples.ImagePipeline.ImagePipeline.RunPipelined	4032 (<No Name>)

Figure 1

From the **Debug** menu, point to **Windows**, then click **Parallel Stacks**. In the Parallel Stacks window, from the drop-down menu in the upper left corner, click **Tasks**, then right-click on the background of the Parallel Tasks window and click **Show External Code**. (You may first need to disable the **Enable Just My Code (Managed only)** option. To locate this option, click the **Tools** menu, click **Options**, and click **Debugging**. This window shows the stack for each of the tasks. In Figure 2, the window contents have been enlarged (point to the zoom control in the left side of the window and use the slider) so only two stacks appear, but all stacks can be accessed.

The Parallel Stacks window



Parallel Stacks

Tasks

1 Task

ImagePipeline.LoadPipelinedImages

ImagePipeline.RunPipelined.AnonymousMet...

1 Task

Bitmap.GetPixel

BitmapExtensions.AddBorder

ImagePipeline.ScaleImage

ImagePipeline.ScalePipelinedImages

ImagePipeline.RunPipelined.AnonymousMet...

Figure 2

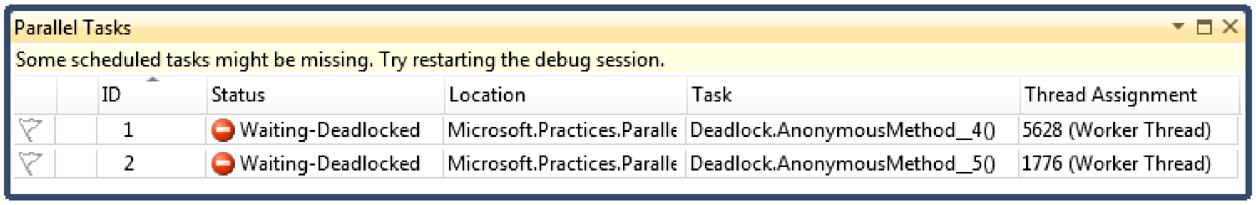
As you continue to press F5, the contents of each window changes as the buffers between pipeline stages empty and fill. This is the expected behavior, so no bugs are indicated. These windows can also reveal unexpected behavior that can help you identify and fix performance problems and synchronization errors. For example, the Parallel Tasks and Parallel Stacks windows can help to identify common concurrency problems such as deadlocks. The following code shows this problem.

```
static void Deadlock()
{
  object obj1 = new object();
  object obj2 = new object();

  Parallel.Invoke(
    () => {
      for (int i = 0; ; i++) {
        lock (obj1) {
          Console.WriteLine("Got 1 at {0}", i);
          lock (obj2) Console.WriteLine("Got 2 at {0}", i);
        }
      }
    },
    () => {
      for (int i = 0; ; i++) {
        lock (obj2) {
          Console.WriteLine("Got 2 at {0}", i);
          lock (obj1) Console.WriteLine("Got 1 at {0}", i);
        }
      }
    });
}
```

This code is a classic example of a deadlock. Two tasks each attempt to acquire locks in an order that leads to a cycle. This eventually results in deadlock. At some point while running this code, the application will stop making progress (this will be obvious because there will be no more new console output). At that point, if you use the **Break All** option on the **Debug** menu, and open the Parallel Tasks window, you'll see something like what's shown in Figure 3.

Parallel Tasks window showing deadlock


Parallel Tasks

Some scheduled tasks might be missing. Try restarting the debug session.

	ID	Status	Location	Task	Thread Assignment
	1	Waiting-Deadlocked	Microsoft.Practices.Paralle	Deadlock.AnonymousMethod__4()	5628 (Worker Thread)
	2	Waiting-Deadlocked	Microsoft.Practices.Paralle	Deadlock.AnonymousMethod__5()	1776 (Worker Thread)

Figure 3

The Concurrency Visualizer

The Visual Studio 2010 profiler includes the Concurrency Visualizer. It shows how parallel code uses resources as it runs: how many cores it uses, how threads are distributed among cores, and the activity of each thread. This information helps you to confirm that your parallel code is behaving as you intend, and can help you diagnose performance problems.

The Concurrency Visualizer has two stages: data collection and visualization. In the collection stage, you enable data collection and run your application. In the visualization stage, you examine the data you collected. This appendix uses the Concurrency Visualizer to profile the ImagePipeline sample from chapter 7 on a computer with two cores.

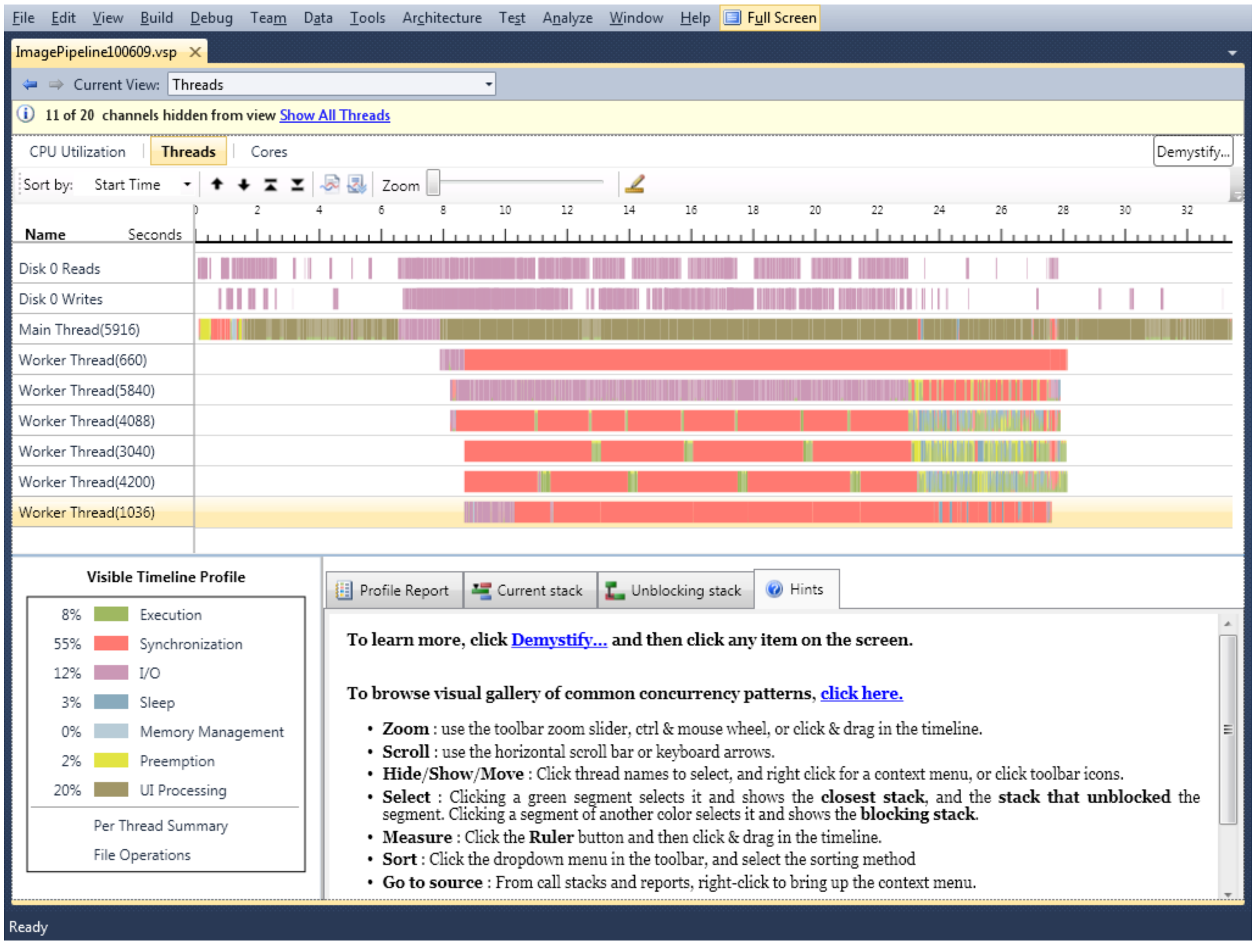
You first perform the data collection stage. To do this, you must run Visual Studio as an administrator because data collection uses kernel-level logging. Open the parallel guide samples solution in Visual Studio. There are several ways to start a data collection run. One way is to open the Visual Studio **Debug** menu, and click **Start Performance Analysis**. The Performance Wizard begins. Click **Concurrency**, and select **Visualize the behavior of a multithreaded application**. The next page of the wizard shows the solution that is currently open in Visual Studio. Select the project you want to profile, which is **ImagePipeline**. Click **Next**. The last page of the wizard asks if you want to begin profiling after the wizard finishes. This checkbox is selected by default. Click **Next**. The Visual Studio profiler window appears and indicates it's **Currently Profiling.** The ImagePipeline sample begins to run and opens its graphical user interface window on the desktop. In order to maximize processor utilitization, select the **Load Balanced** option, then click **Start**. In order to collect plenty of data to visualize, let the **Images** counter (on the graphical interface) reach at least 20. Then click **Stop Profiling** in the Visual Studio profiler window.

During data collection, the performance analyzer frequently takes a sample of data (called a snapshot) that records the state of your running parallel code. Each data collection run writes several data files, including a .vsp file. A single data collection run can write files that are hundreds of megabytes. Data collected during separate runs of the same program can differ because of uncontrolled factors such as other processes running on the same computer.

You can run the visualization stage whenever the files are available. You don't need to be running Visual Studio as an administrator to do this. There are several ways to begin visualization. You can request the Performance Wizard to start visualization as soon as data collection finishes. Alternatively, you can simply open any .vsp file in Visual Studio. If you select the first option, you'll see a summary report once the data is collected and analyzed. The summary report shows the different views you can see. These include a Threads view, a CPU Utilization view, and a Cores view.

Figure 4 shows the Threads view. Each task is executed in a thread. The Concurrency Visualizer shows the thread for each task (remember that there may be more than one task per thread because of inline tasks).

Threads view of the Concurrency Visualizer



8%	🟩	Execution
55%	🟥	Synchronization
12%	🟪	I/O
3%	🟦	Sleep
0%	⬜	Memory Management
2%	🟨	Preemption
20%	🟫	UI Processing

To learn more, click Demystify... and then click any item on the screen.

To browse visual gallery of common concurrency patterns, click here.

- **Zoom** : use the toolbar zoom slider, ctrl & mouse wheel, or click & drag in the timeline.
- **Scroll** : use the horizontal scroll bar or keyboard arrows.
- **Hide/Show/Move** : Click thread names to select, and right click for a context menu, or click toolbar icons.
- **Select** : Clicking a green segment selects it and shows the **closest stack**, and the **stack that unblocked** the segment. Clicking a segment of another color selects it and shows the **blocking stack**.
- **Measure** : Click the **Ruler** button and then click & drag in the timeline.
- **Sort** : Click the dropdown menu in the toolbar, and select the sorting method
- **Go to source** : From call stacks and reports, right-click to bring up the context menu.

Figure 4
The Concurrency Visualizer screens contain many details that may not be clear in these figures, which are reduced in size and are not in full color. A verson of this appendix that contains the full color screen shots are available on the CodePlex site at http://parallelpatterns.codeplex.com/.

Figure 5 illustrates the CPU Utilization view. The CPU Utilization view shows how many processors (logical cores) the entire application (all tasks) uses, as a function of time. On the computer used for this example, there are two logical cores. Other processes not related to the application are also shown. For each process, there's a graph that shows how many processors it's using at each point in time. To make the processes easier to distinguish, the area under each process's graph appears in a different color (some colors may not be reproduced accurately in this figure). Some data points show a fraction, not 0, 1, or 2, because each point represents an average calculated over the sampling interval.

During this particular data collection run, the graphical user interface showed that the first few images appeared slowly, and the remaining images appeared more rapidly. (This behavior didn't occur on every run). The view reflects the behavior. Early in the run (before about 20000 on the time scale) the application runs in bursts, and fills the pipeline as it loads images. When there are several images in the pipeline (after about 20000), pipeline tasks can run in parallel and the application uses two logical cores.

This view also shows intervals between bursts (before 20000) where the application gets no processors. During these intervals the application is blocked, or is preempted by other processes.

Detail of CPU Utilization view

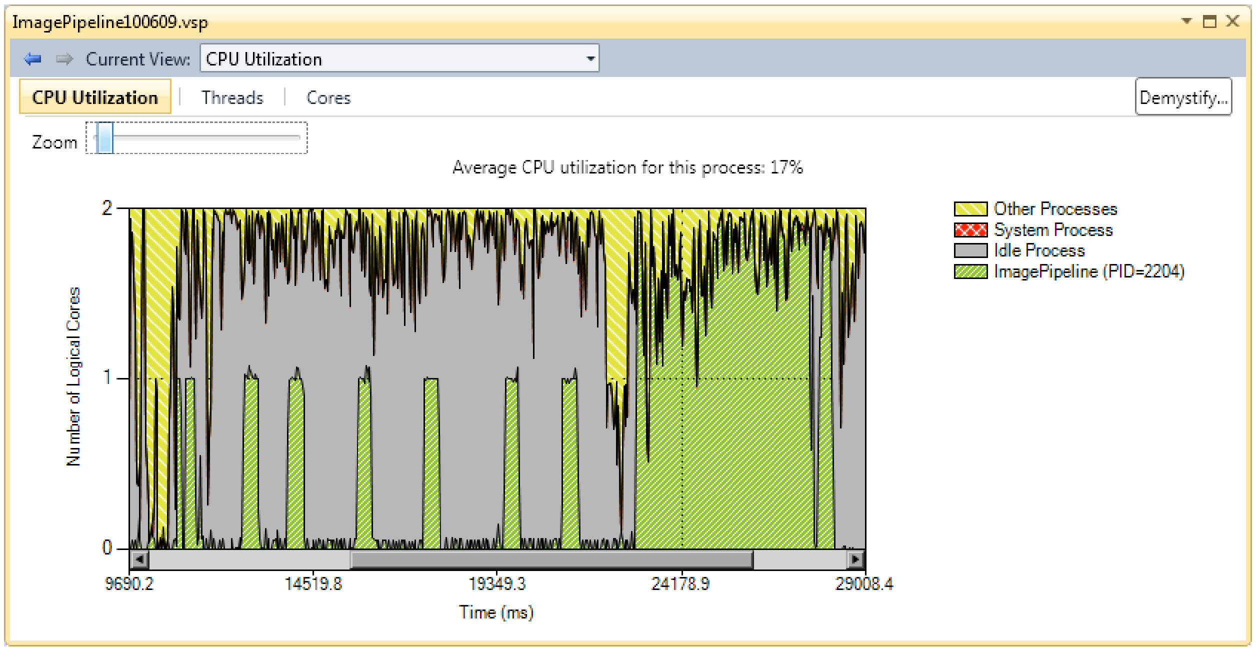
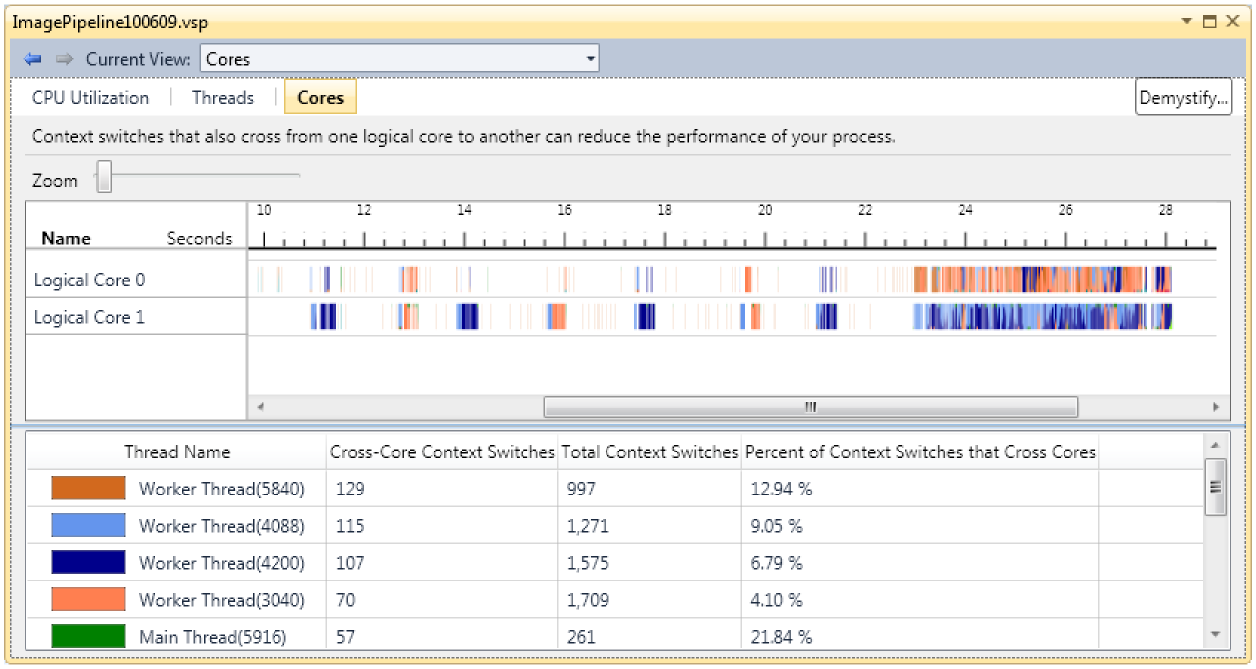


Figure 5

Figure 6 illustrates the Cores view. The Cores view shows how the application uses the available cores. There is a timeline for each core, with a color-coded band that indicates when each thread is running (a different color indicates each thread.) Between 10 and 22 on the time scale, the application runs in bursts and the empty intervals indicate when there was no work from this process running on either core. Between 22 and 28 the pipeline is filled and more tasks are eligible to run than there are cores. Several threads alternate on each core and the table beneath the graph shows that there is a great deal of context switching.

Detail of Cores view



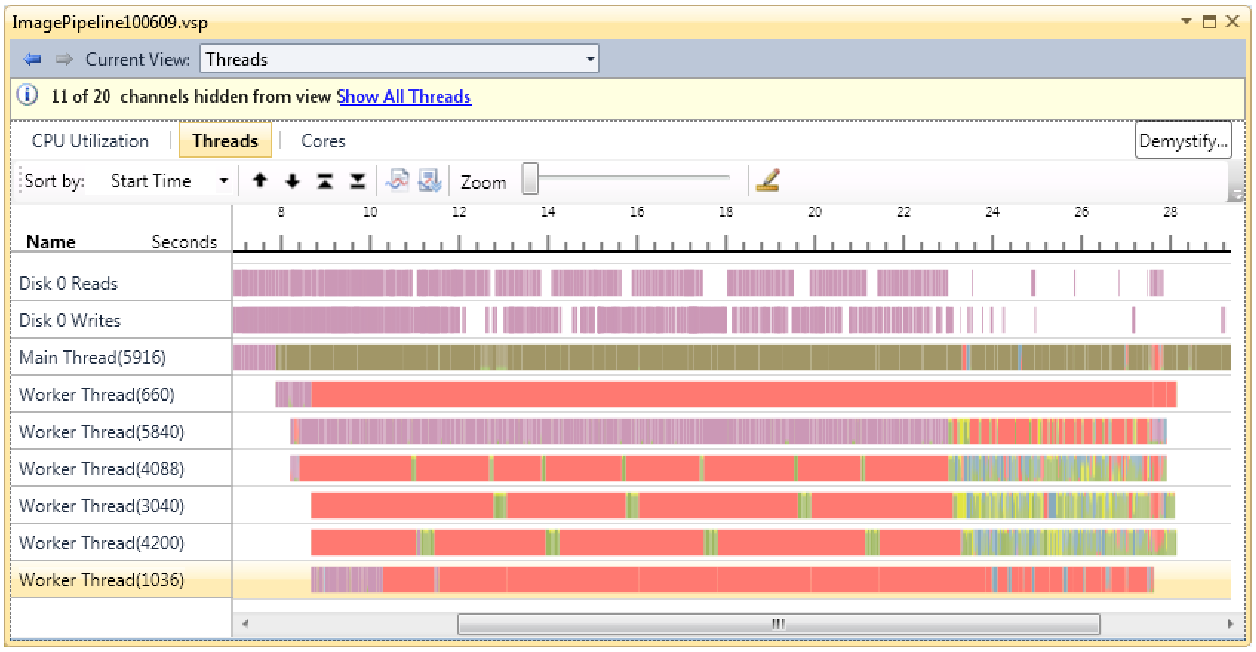
Context switches that also cross from one logical core to another can reduce the performance of your process.

Thread Name	Cross-Core Context Switches	Total Context Switches	Percent of Context Switches that Cross Cores
🟧 Worker Thread(5840)	129	997	12.94 %
🟦 Worker Thread(4088)	115	1,271	9.05 %
🟦 Worker Thread(4200)	107	1,575	6.79 %
🟧 Worker Thread(3040)	70	1,709	4.10 %
🟩 Main Thread(5916)	57	261	21.84 %

Figure 6

Figure 7 illustrates the Threads view. The Threads view shows how each thread spends its time. There is a timeline with color-coded bands that indicate different kinds of activity. For example, red indicates when the thread is synchronizing (waiting for something). This view initially shows idle threads in the thread pool. (You can hide them by right clicking on the view and selecting **Hide**). This view shows that the main thread is active throughout; the green color indicates user interface activity.

Detail of Threads view



11 of 20 channels hidden from view Show All Threads

Figure 7
The pipeline threads execute in bursts before time 22, alternating between running and synchronizing, as they wait for the pipeline to fill. After 22, some pipeline threads execute frequently and others execute almost continuously. There are more pipeline threads than cores, so some pipeline threads must alternate between running and being preempted.

You can use the Scenario library to mark different phases of complex applications. The following code shows an example. (The Scenario library is a free download on the MSDN Code Gallery web site. For more information, go to http://msdn.microsoft.com/en-us/library/dd984115.aspx.)

```
Scenario.Scenario myScenario = new Scenario.Scenario();
myScenario.Begin(0, "Main Calculation");

// Main calculation phase...

myScenario.End(0, "Main Calculation");
```

These markers will be displayed in the **Threads** view and CPU **Utilization** view. Don't use too many markers as they can easily swamp the visualization and make it hard to read. The tool may hide some markers to improve visibility. You can use the zoom feature to increase the magnification and see the hidden markers for a specific section of the view.

Visual Patterns

The patterns discussed in this book primarily focus on ways to express potential parallelism. However, there are other kinds of patterns that are useful in parallel development. The human mind is very good at recognizing visual patterns, and the Concurrency Visualizer takes advantage of this. You can learn to identify some common visual patterns that occur when an application has specific performance problems. This section describes visual patterns that will help you to recognize and fix oversubscription, lock contention, and load imbalances.

Oversubscription

Oversubscription occurs when there are more threads than logical processors to run them. Oversubscription can cause poor performance because of the high number of context switches, each of which takes some processing time and which can decrease the benefits provided by memory caches.

The Concurrency Visualizer makes it easy to recognize oversubscription because it causes there to be large numbers of yellow regions in the profiler trace. Yellow means that a thread was preempted (the thread was switched out). When traced, the following code yields a quintessential depiction of oversubscription.

```
static void Oversubscription()
{
    for (int i = 0; i < (Environment.ProcessorCount * 4); i++)
    {
        new Thread(() => {
            // Do work
            for (int j = 0; j < 1000000000; j++) ;
        }).Start();
    }
}
```

Figure 8 illustrates the Threads view from one run of this function on a quad-core system. It produces a very distinct pattern.

Threads view that shows oversubscription

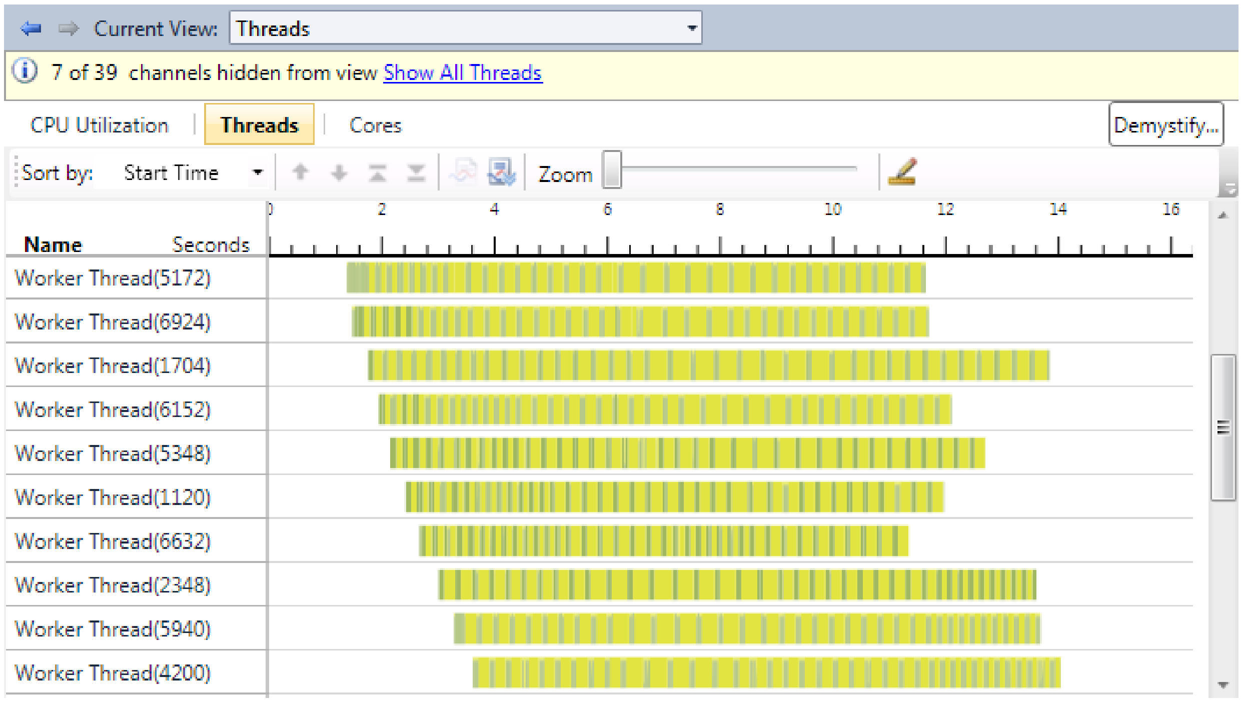


Figure 8

Lock Contention and Serialization

Contention occurs when a thread attempts to acquire a lock that is held by another thread. In many cases, this results in the first thread blocking until the lock is released. The Threads view of the Concurrency Visualizer depicts blocking in red. It is often a sign of decreased performance. In extreme cases, an application can be fully serialized by one or more locks, even though multiple threads are being used.

The following method produces a lock convoy, which leads to significant lock contention and serialization of the program even though multiple threads are in use. A lock convoy is a performance problem that occurs when multiple threads contend for a frequently shared resource.

```
static void LockContention()
{
    object syncObj = new object();
    for (int p = 0; p < Environment.ProcessorCount; p++)
    {
        new Thread(() => {
            for(int i=0; i<50; i++)
            {
                // Do work
                for (int j = 0; j < 1000; j++);

                // Do protected work
                lock (syncObj)
                    for (int j = 0; j < 100000000; j++);
            }
        }).Start();
    }
}
```

Figure 9 illustrates the pattern this code produced in the Threads view of the Concurrency Visualizer.

Threads view showing lock convoy

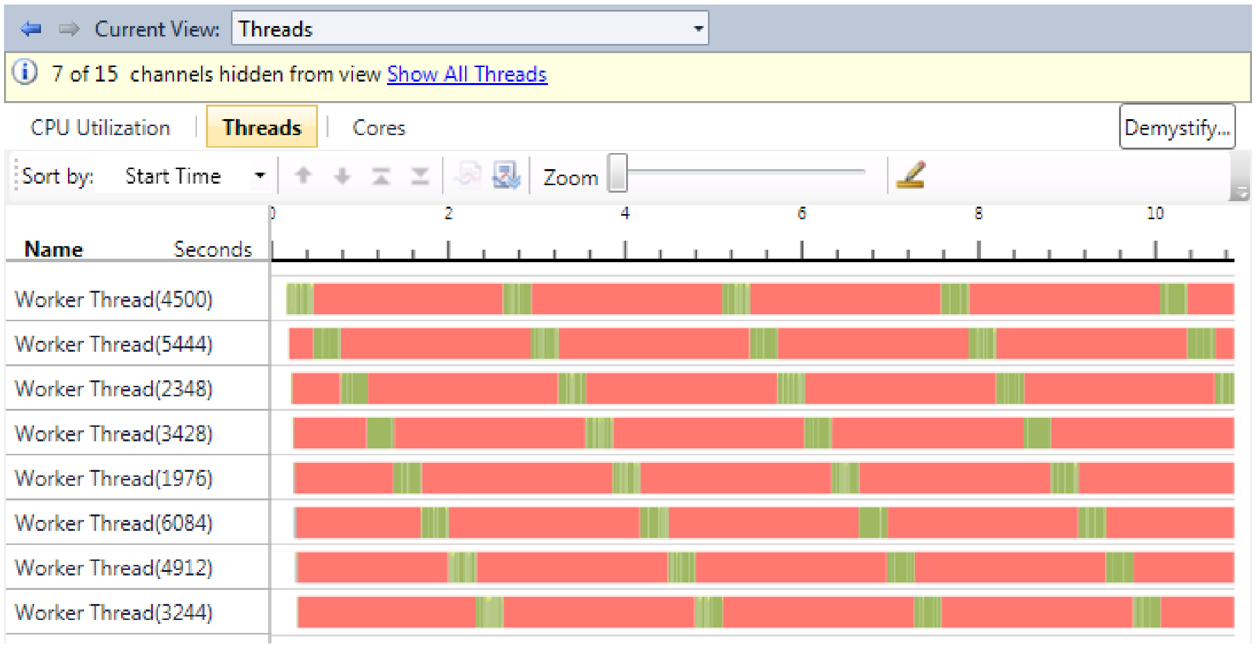


Figure 9

Load Imbalance

A load imbalance occurs when work is unevenly distributed across all of the worker threads that are involved in a parallel operation. Load imbalances mean that the system is underutilized, because some threads or cores sit idly while others finish processing the operation. The visual pattern produced by a load imbalance is recognizable in several of the Concurrency Visualizer views. The following code is designed to create a load imbalance.

```
static void LoadImbalance()
```

```
{
  const int loadFactor = 10;

  ParallelEnumerable.Range(0, 100000).ForAll(i =>
  {
    for (int j = 0; j < (i * loadFactor); j++) ;
  });
}
```

While most of the parallelism support in .NET 4 uses dynamic partitioning to apportion work between worker tasks, the **ParallelEnumerable.Range** method from PLINQ uses static partitioning. This example, on a system with eight logical cores, causes elements [0, 12499] to be processed by one task, and elements [12500, 24999] to be processed by another task, and so on. The body of the workload simply iterates from 0 to the current index value, which means that the amount of work to be done is proportional to the index. Workers that process lower ranges will have significantly less work to do than the workers that process the upper ranges. Figure 10, which is the CPU Utilization view in the Concurrency Visualizer, illustrates this.

CPU view that shows a load imbalance

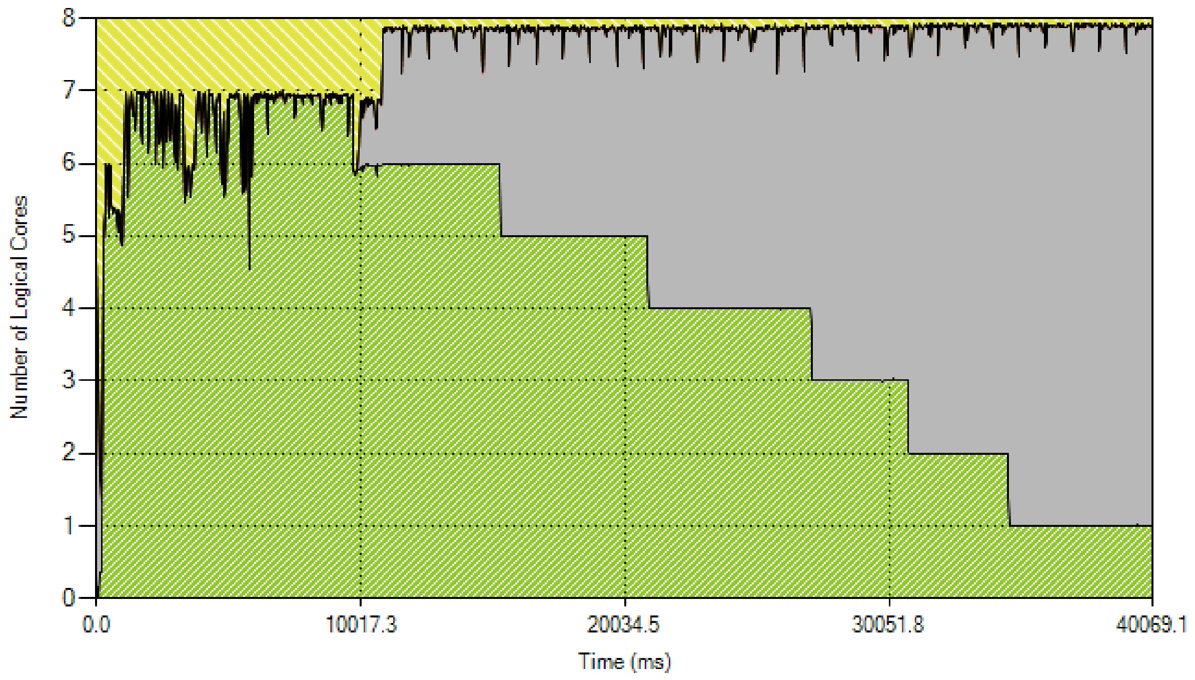


Figure 10
When the method begins to execute, seven logical cores on the system are being used. However, after a period of time, usage drops as each core completes its work. This yields a "stair-step pattern," as threads are dropped after they complete their portion of the work. The Threads view confirms this analysis. Figure 11 illustrates this.

Threads view that shows a load imbalance

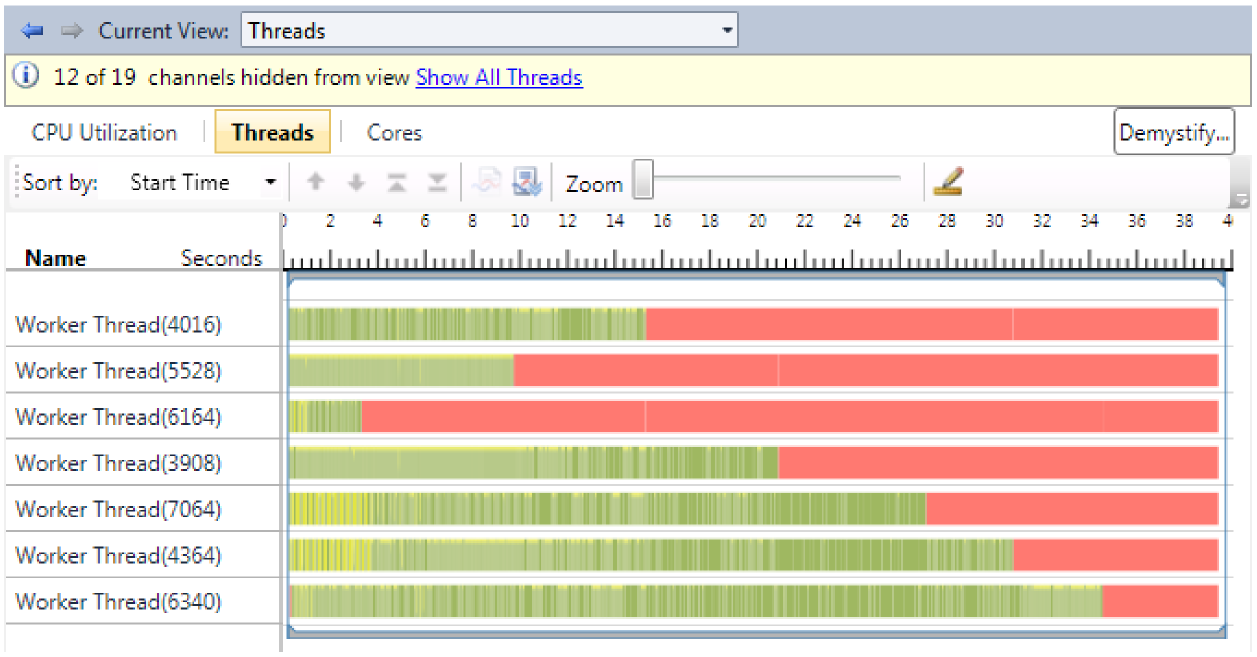


Figure 11
The Threads view shows that after completing a portion of the work, the worker threads were idle while they waited for CLR Worker Thread 6340 to complete the remaining work.

Further Reading

The Parallel Performance Analysis blog at MSDN discusses many techniques and examples. MSDN also provides information on the Scenario library.

Parallel Performance Analysis in Visual Studio 2010. http://blogs.msdn.com/b/visualizeparallel/.

Performance Tuning with the Concurrency Visualizer in Visual Studio 2010 http://msdn.microsoft.com/en-us/magazine/ee336027.aspx.

Scenario Home Page http://code.msdn.microsoft.com/scenario.