Hands-On Lab

Introduction to the Visual Studio 2010 Parallel Debugger

Lab version:	1.0.0
Last updated:	3/13/2010

Contents

Introduction 4
Lab Objectives 4
Prerequisites 4
An Overview of Multithreading 4
Exercise 1 – Examining the sample project 5
Exercise 2 – Understanding the Tool Windows 7
Exercise 3 – Viewing multiple call stacks 11
Exercise 4 – Tasks View of Parallel Stacks Window 15
Exercise 5 – Dealing with diverging Tasks 17
Exercise 6 – Flagging Threads of Interest 19
Exercise 7 – Dealing with Deadlocks and Other Statuses 21
Lab Summary 25

Introduction

In this lab, we'll be using the new Parallel Debugging support in Visual Studio 2010 to debug a simple task-based application. We will cover multithreading concepts, System.Threading.Task usage, and use the Parallel Stacks and Parallel Tasks windows.

Lab Objectives

After completing this lab, you'll better understand multithreaded debugging including:

1. Viewing the running threads in an application
2. Decomposing parallel work into tasks and debugging those tasks
3. How to gain understanding of the runtime workings of a multithreaded application

Prerequisites

This lab requires the following components:

- Visual Studio 2010 Professional Edition or higher (Express Edition does not contain Task debugging panes)
- Microsoft .NET Framework 4 (installed with VS 2010)

This lab assumes intermediate knowledge of threading concepts. This includes the concept of hardware and software threads, thread scheduling, and work partitioning. Familiarity with .NET and C#, though certainly helpful, is not strictly required. Some familiarity with Visual Studio 2005, 2008, or 2010 is also very helpful.

The following steps should be taken to ensure an experience that matches the screenshots in this lab:

To enable the **Just My Code** feature, click the **Tools | Options** menu command. Expand **Debugging** and then click **General**. Then check the **Enable Just My Code (Managed only)** check box.

To toggle whether external code is displayed, right-click the **Name** table header of the **Call Stack** window and then check or clear **Show External Code**. The screenshots in this lab are taken with external code showing.

An Overview of Multithreading

Writing powerful software has much to do with creating fast, efficient algorithms. Over the years, software performance has increased automatically as processor clock speeds have followed Moore's Law, which states that processor speeds double every 18 months. This automatic performance increase is sometimes called the free lunch. As the laws of physics have come to prevail though, clock speeds have been unable to keep up with that trend. Newer processors are adding multiple cores to the same chip, at the expense of clock speed.

Creating logic for one processor, that is, sequential code, is very different from code that runs well on multiple cores or processors: parallel code. Sequential code run on a quad-processor machine will only take advantage of one processor. To truly scale, it's imperative to think of how to create code that can run in parallel.

The downside to parallel development is the new way of thinking. This makes it more challenging to plan, but also more difficult to troubleshoot and debug. Instead of always knowing the state of your code, now there are multiple worker threads each at different points. The application might run without locking up, yet individual workers may be stuck. To make it worse, there are often unintended consequences where the actions of one thread influence the actions of another – generally in ways that were unforeseen, and almost always in ways that are challenging to deal with.

Visual Studio 2010 takes a major leap forward in providing support for developers creating parallel applications that are optimized for multiple cores. In addition to new framework additions for parallelism, you now have tools in the IDE that put the right information in front of you when you need it.

Exercise 1 – Examining the sample project

Before looking at the runtime and debugging experience, it is important to understand the sample solution:

1. Open the **TasksHOL** solution
2. In the **Solution Explorer**, double-click **OrderForm.cs**

This application doesn't perform any useful work, but it demonstrates how to use the new Task API's and how they appear in Visual Studio debugging tools. The purpose of a Task object is to complete some sort of action on a background thread. Though its use is flexible, a Task is typically designed to perform some sort of work and then exit. A limited pool of Task objects will be assigned units of work until all such items are processed. Tasks also provide a means to cancel operations and powerful methods of debugging.

The **OrderForm** class simply wraps a unit of work – a dummy order form such as would be created by any number of web sites.

3. In the **Solution Explorer**, double-click **FormTasks.cs**

All actions to perform on a task are encapsulated in the **Work** static class. All tasks begin in the **ProcessForm** method. This method checks for customer eligibility, then either processes or cancels the order. By different tasks branching into different methods, we have the opportunity to see parallel and diverging call stacks.

4. In the **Solution Explorer**, double-click **MainWindow.xaml**

The application uses a WPF **DataGrid** to present progress. A **BackgroundWorker** object works in the background using a LINQ query for creating the collection to view. In a more robust application, this would be in a ViewModel object.

5. Click the **View | Code** menu command (or press F7)
6. Look at the **MainWindow** constructor

The next block simply creates the **BackgroundWorker** for user interface updating. This operates on a background thread yet requires no knowledge of threading to use it. The entry point for the **BackgroundWorker** is the **w_DoWork** method. When this is called, it creates the tasks using the **CreateTasks** method, then waits for the tasks to complete. There are calls found throughout to the **ReportProgress** method. This is simply to cause the UI to refresh.

7. Locate the **CreateTasks** method:

```
for (int i = 0; i < numTasks; i++)
{
    OrderForm f = new OrderForm() {Name="Test #" + i, PostalCode="00000"};

    tasks.Add(new Task(Work.ProcessForm, f, src.Token));
    w.ReportProgress(prog++);
}
```

This code creates a collection of **Task** objects. A **Task** object is given an entry point and a work object. In this code, the work object is the **OrderForm** instance, and the entry point is the **ProcessForm** method.

Tasks are units of work to be performed, without information regarding how they are to be scheduled. A **Task** object can be useful in troubleshooting, and provides numerous properties for starting, status, and error handling. Since the work is separate from the thread, processor utilization and work scheduling can occur independently.

Tasks can also be started immediately using the **Task.Factory.StartNew** static method which is recommended for performance reasons. Creating a **Task** object directly just provides you with a reference to a work object. Scheduling and executing is a separate task. Using the **StartNew** method just lets you create and start it at the same time.

8. With the tasks all created, they can now be started:

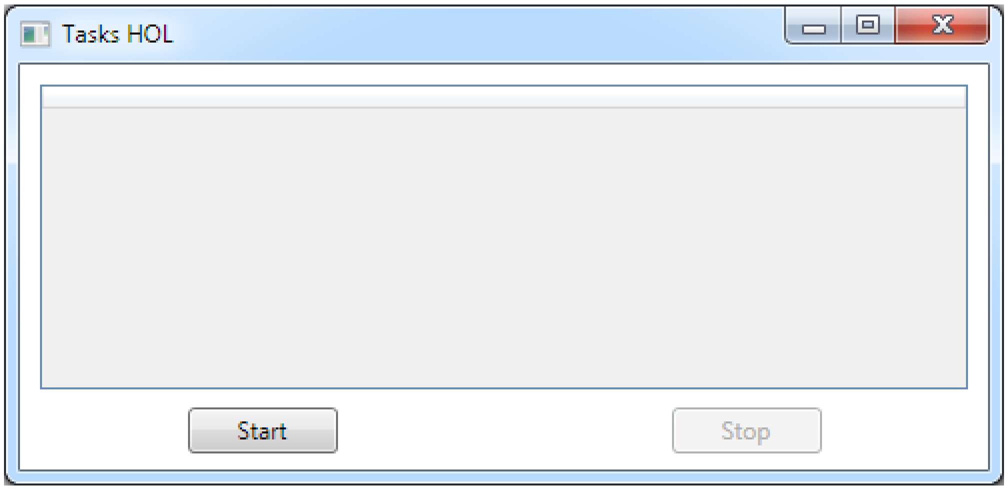
```
foreach (Task t in tasks)
{
    t.Start();
    w.ReportProgress(prog++);
}
```

It’s important to note that simply starting a task does not guarantee immediate execution of it. It simply signals readiness. It is up to the thread scheduler to determine when an idle thread should be assigned a given task.

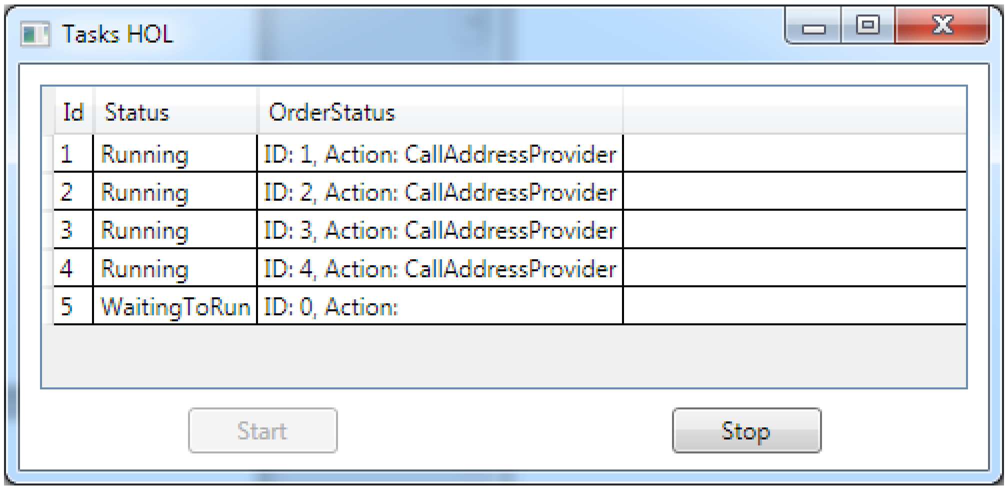
Exercise 2 – Understanding the Tool Windows

In this section, you will see how the tasks progress, how actions can be coordinated, and a way to keep simple operations thread-safe.

1. To start debugging, click the **Debug | Start Debugging** menu command, or **F5**.

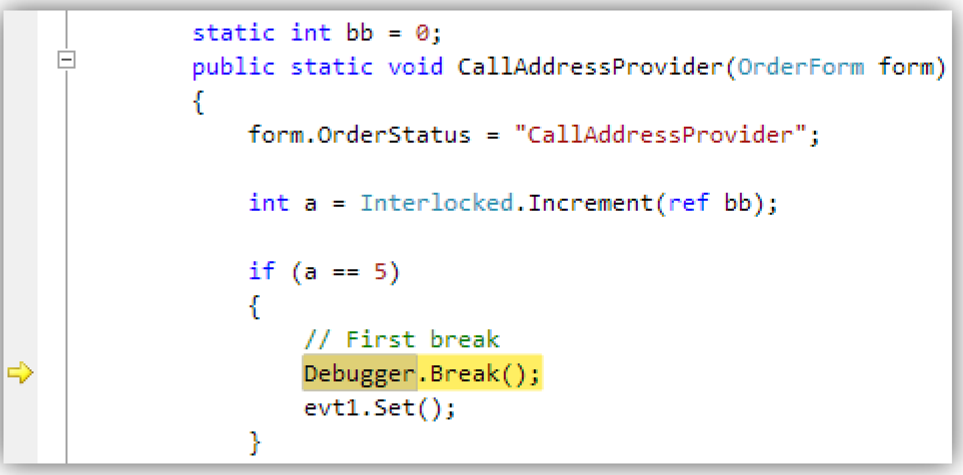


2. Click the **Start** button



Id	Status	OrderStatus
1	Running	ID: 1, Action: CallAddressProvider
2	Running	ID: 2, Action: CallAddressProvider
3	Running	ID: 3, Action: CallAddressProvider
4	Running	ID: 4, Action: CallAddressProvider
5	WaitingToRun	ID: 0, Action:

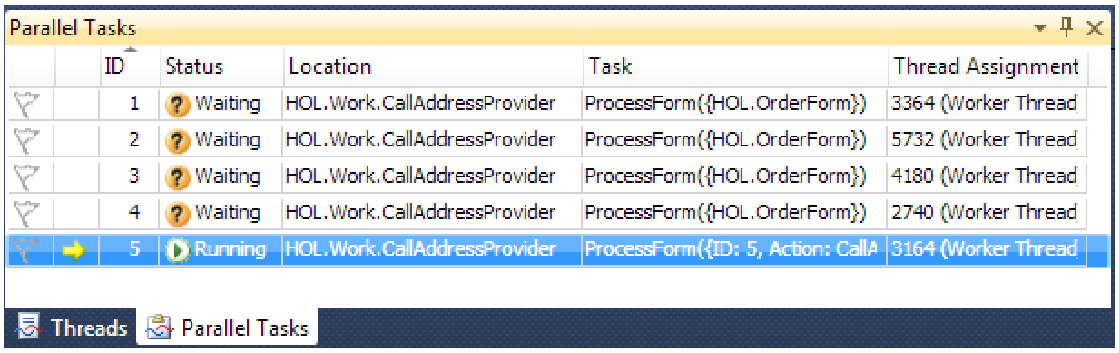
3. After a few seconds, the debugger automatically breaks. This is accomplished using the **Debugger.Break()** call in the **CallAddressProvider()** method.



```
        static int bb = 0;
        public static void CallAddressProvider(OrderForm form)
        {
            form.OrderStatus = "CallAddressProvider";

            int a = Interlocked.Increment(ref bb);

            if (a == 5)
            {
                // First break
                Debugger.Break();
                evt1.Set();
            }
```

As each task enters the **CallAddressProvider**, its **OrderStatus** property changes to "CallAddressProvider." This is a rough way to gauge the operation of the Task when it's active. This wouldn't make much sense in production code, but for demonstration purposes it makes it easier to monitor the flow. This is the property that the **BackgroundWorker** uses to show status in the DataGrid.
The second line is where it calls a statement to increase the **bb** variable. Since every thread/Task needs to increment **bb** and break if the new value is "5", you can't just make a call to "bb++". The **Interlocked.Increment** call performs this as a guaranteed atomic, thread-safe operation, placing the new value of **bb** into **a**. If **a** is equal to "5", then **Break** is called. If you are unfamiliar with thread-safety concepts and the **Interlocked** object, please see this article for more information.

4. If the **Parallel Tasks** window is not visible, click the **Debug | Windows | Parallel Tasks** menu command:



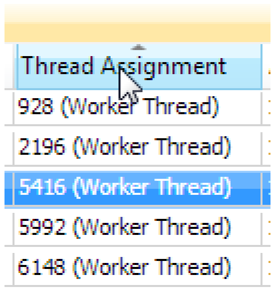
Parallel Tasks

		ID	Status	Location	Task	Thread Assignment
		1	Waiting	HOL.Work.CallAddressProvider	ProcessForm({HOL.OrderForm})	3364 (Worker Thread
		2	Waiting	HOL.Work.CallAddressProvider	ProcessForm({HOL.OrderForm})	5732 (Worker Thread
		3	Waiting	HOL.Work.CallAddressProvider	ProcessForm({HOL.OrderForm})	4180 (Worker Thread
		4	Waiting	HOL.Work.CallAddressProvider	ProcessForm({HOL.OrderForm})	2740 (Worker Thread
	➡	5	Running	HOL.Work.CallAddressProvider	ProcessForm({ID: 5, Action: CallA	3164 (Worker Thread

The **Parallel Tasks** window is similar to the **Threads** window, but instead of showing every thread object, it only shows **Task** objects that have been started. Tasks created using the **Task** constructor (such as in the lab code) are not shown until **Start** is called. The factory method **StartNew** creates and starts tasks so they show up immediately.
Tasks in the **Parallel Tasks** window could be waiting to run, in the **Scheduled** status, or executing in **Running** or **Waiting** status. The task's status is shown in the

Status column. At a glance you can see what each task is doing and identify any differences with what you were expecting.

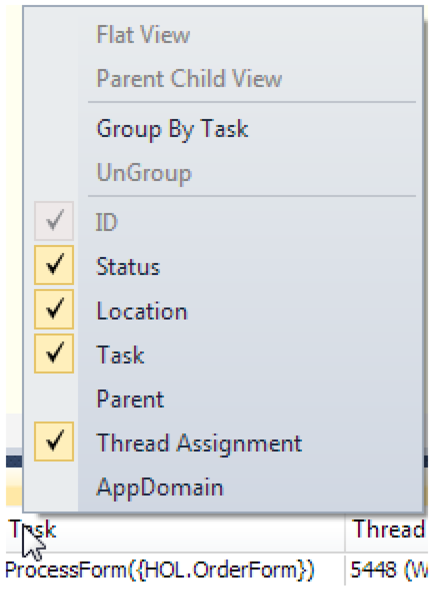
5. Click the **Thread Assignment** column header



Thread Assignment
928 (Worker Thread)
2196 (Worker Thread)
5416 (Worker Thread)
5992 (Worker Thread)
6148 (Worker Thread)

As with many data grids, you can order ascending or descending by any column by simply clicking on the column header. Try this for any column to create the desired view.

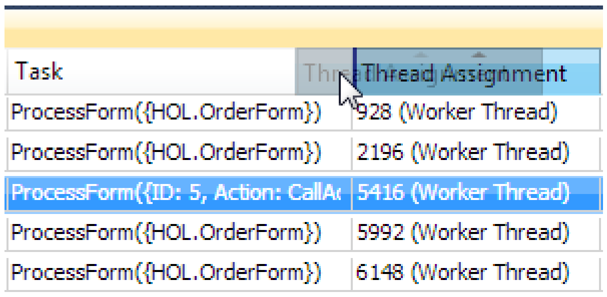
6. Right-click the **Task** column header



ProcessForm({HOL.OrderForm}) 5448 (W

By right-clicking column headers, you can choose which columns to show or hide. A column like **AppDomain** is not very useful in an application with a single application domain so can be hidden to conserve space.

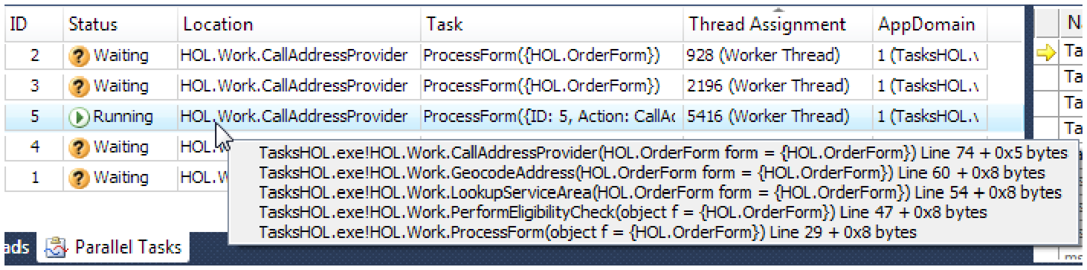
7. Click the **Thread Assignment** column header and drag it to the left. Release it between other columns to move it, or back over its original position to maintain the same spot.



Task	Thread Assignment
ProcessForm({HOL.OrderForm})	928 (Worker Thread)
ProcessForm({HOL.OrderForm})	2196 (Worker Thread)
ProcessForm({ID: 5, Action: CallAc	5416 (Worker Thread)
ProcessForm({HOL.OrderForm})	5992 (Worker Thread)
ProcessForm({HOL.OrderForm})	6148 (Worker Thread)

In some cases, you may prefer to see the columns in a different order. Simply drag-and-drop the columns to the desired locations.

8. Hover over the **Location** column

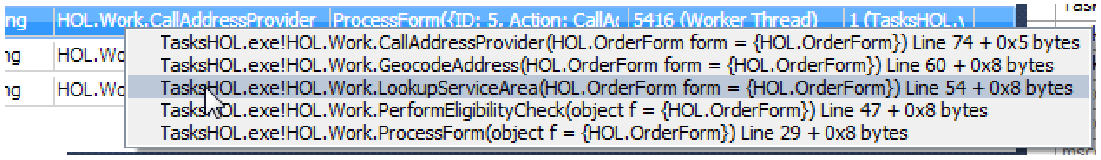


ID	Status	Location	Task	Thread Assignment	AppDomain
2	Waiting	HOL.Work.CallAddressProvider	ProcessForm({HOL.OrderForm})	928 (Worker Thread)	1 (TasksHOL.\
3	Waiting	HOL.Work.CallAddressProvider	ProcessForm({HOL.OrderForm})	2196 (Worker Thread)	1 (TasksHOL.\
5	Running	HOL.Work.CallAddressProvider	ProcessForm({ID: 5, Action: CallAc	5416 (Worker Thread)	1 (TasksHOL.\
4	Waiting	HOL.W			
1	Waiting	HOL.W			

```
TasksHOL.exe!HOL.Work.CallAddressProvider(HOL.OrderForm form = {HOL.OrderForm}) Line 74 + 0x5 bytes
TasksHOL.exe!HOL.Work.GeocodeAddress(HOL.OrderForm form = {HOL.OrderForm}) Line 60 + 0x8 bytes
TasksHOL.exe!HOL.Work.LookupServiceArea(HOL.OrderForm form = {HOL.OrderForm}) Line 54 + 0x8 bytes
TasksHOL.exe!HOL.Work.PerformEligibilityCheck(object f = {HOL.OrderForm}) Line 47 + 0x8 bytes
TasksHOL.exe!HOL.Work.ProcessForm(object f = {HOL.OrderForm}) Line 29 + 0x8 bytes
```

Hovering over the **Location** column for a given row causes a popup with the call stack to appear.

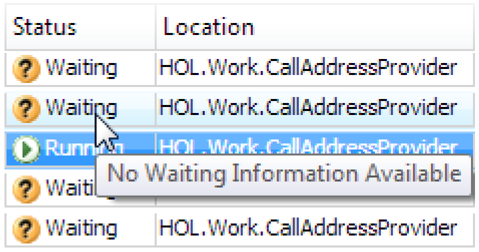
9. Hover over the actual stack frames in the popup and notice the highlight changes. Clicking a row would cause the current frame to change.



```
TasksHOL.exe!HOL.Work.CallAddressProvider(HOL.OrderForm form = {HOL.OrderForm}) Line 74 + 0x5 bytes
TasksHOL.exe!HOL.Work.GeocodeAddress(HOL.OrderForm form = {HOL.OrderForm}) Line 60 + 0x8 bytes
TasksHOL.exe!HOL.Work.LookupServiceArea(HOL.OrderForm form = {HOL.OrderForm}) Line 54 + 0x8 bytes
TasksHOL.exe!HOL.Work.PerformEligibilityCheck(object f = {HOL.OrderForm}) Line 47 + 0x8 bytes
TasksHOL.exe!HOL.Work.ProcessForm(object f = {HOL.OrderForm}) Line 29 + 0x8 bytes
```

ng
HOL.Work.CallAddressProvider
ProcessForm({ID: 5, Action: CallAc
5416 (Worker Thread)
1 (TasksHOL.\

10. Hover over the **Waiting** status

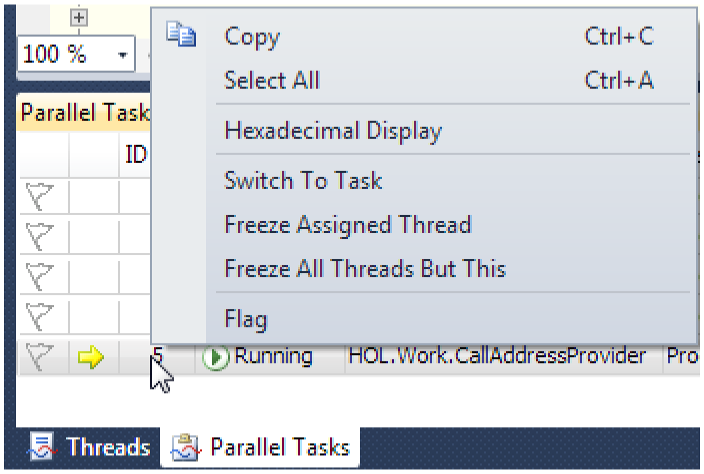


Status	Location
Waiting	HOL.Work.CallAddressProvider
Waiting	HOL.Work.CallAddressProvider
Running	HOL.Work.CallAddressProvider
Waiti	
Waiting	HOL.Work.CallAddressProvider

No Waiting Information Available

A thread can be waiting on a shared lock, an I/O operation, or a thread synchronization operation. If specific information is available, the tooltip for **Status** will display it.

11. Right-click the row with **ID** of 5



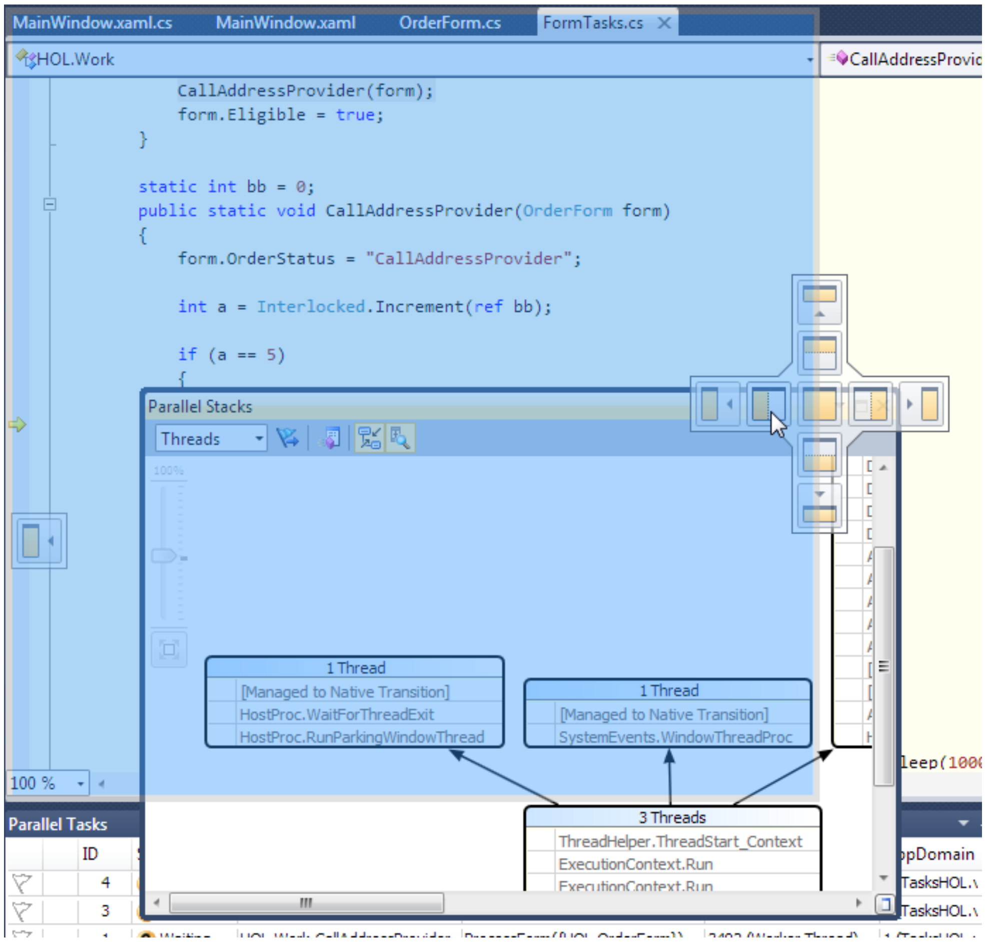
You can select all rows, copy the details of selected rows to the clipboard, display numeric values in hexadecimal form, switch to the selected task (like double-clicking), or freeze threads – an advanced topic out of scope in this session.

Exercise 3 – Viewing multiple call stacks

In this task, learn how Visual Studio displays tasks that share common stack frames, a sure indication of work being performed in parallel. These common stack frames are coalesced into single nodes as part of an overall execution flow. This feature makes it easier to focus on relevant information without being overloaded. This may not be immediately useful with just a few threads, but as works scales to more and more threads, the consolidation of information has a powerful benefit.

1. Click the **Debug | Windows | Parallel Stacks** menu command. The new window will appear. Now you can view multiple call stacks at the same time in a single view.
2. [Optional] The **Parallel Stacks** pane opens as a floating tool window. This is ideal for multi-monitor environments. You can actually maximize this window to a secondary monitory. On a single monitor, it works best docked to the side of the code editor. Dock it along-side other tabbed documents (like **MainWindow.xaml.cs**) by clicking and dragging the **Parallel Stacks** title bar and dragging until the docking

indicator appears, then dropping it onto the left of center region:

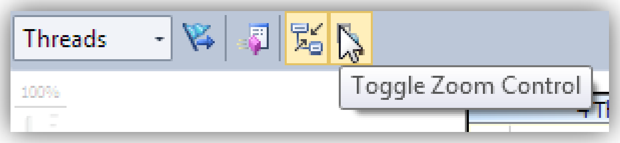


```
            CallAddressProvider(form);
            form.Eligible = true;
        }

        static int bb = 0;
        public static void CallAddressProvider(OrderForm form)
        {
            form.OrderStatus = "CallAddressProvider";

            int a = Interlocked.Increment(ref bb);

            if (a == 5)
            {
```

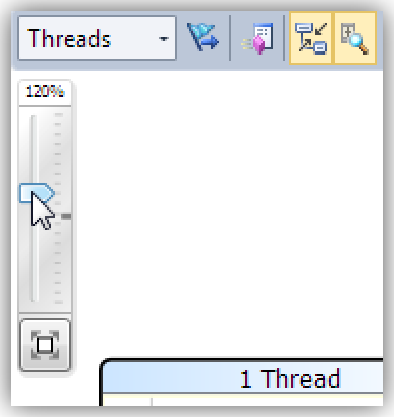
```
1 Thread
[Managed to Native Transition]
HostProc.WaitForThreadExit
HostProc.RunParkingWindowThread

1 Thread
[Managed to Native Transition]
SystemEvents.WindowThreadProc

3 Threads
ThreadHelper.ThreadStart_Context
ExecutionContext.Run
ExecutionContext.Run
```

ID	pDomain
4	TasksHOL.\
3	TasksHOL.\

3. For better visibility, you can change the scale of the graphics using a zoom control. To display the zoom control, click the **Toggle Zoom Control** button in the **Parallel Stacks** toolbar (fourth button):

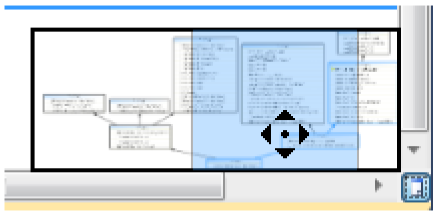


With it shown, you can drag it to the desired size to show everything or to zoom in on certain elements:

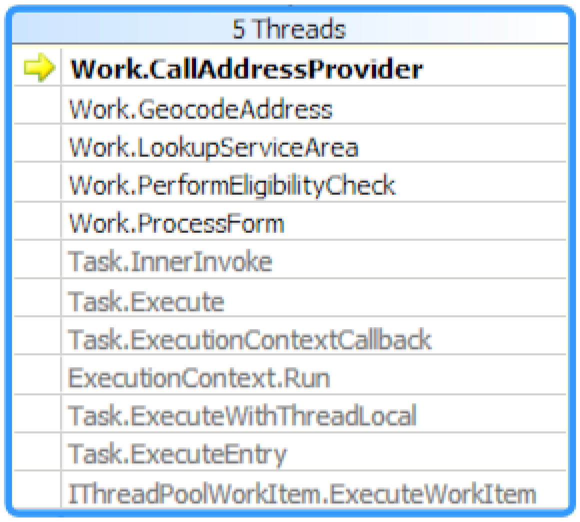


Use the scrollbars at the right and bottom edges of the window to pan if needed. Screenshots in this lab will be shown at 120% with the slider hidden. Click the **Toggle Zoom Control** button again to hide the zoom control. Alternatively, you can use your control key and mousewheel to zoom in and out.

4. For zooming, you can also use the bird's eye view button in the lower-right corner (between the scrollbars). Click this button to show or hide the view, then drag the view area around:



5. Notice the **5 Threads** node:



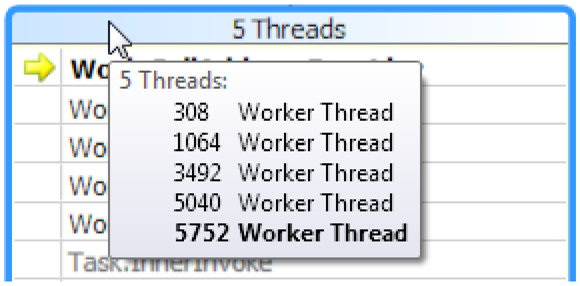
5 Threads
- **Work.CallAddressProvider**
- Work.GeocodeAddress
- Work.LookupServiceArea
- Work.PerformEligibilityCheck
- Work.ProcessForm
- Task.InnerInvoke
- Task.Execute
- Task.ExecutionContextCallback
- ExecutionContext.Run
- Task.ExecuteWithThreadLocal
- Task.ExecuteEntry
- IThreadPoolWorkItem.ExecuteWorkItem

This coalesced view makes it clear that all five threads started in **ProcessForm**, then called **PerformEligibilityCheck**, then called **LookupServiceArea**, and so on.

6. Notice the yellow arrow pointing at **Work.CallAddressProvider**
   Each row represents a method context (i.e. one or more stack frames in the same

method). Since the container is collapsing five different threads, the yellow arrow identifies the active stack frame of the current thread.

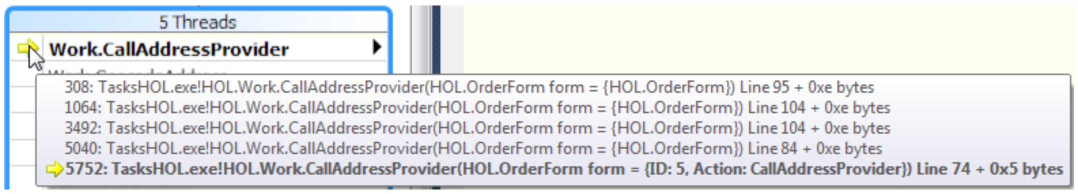
7. Hover over the header (**5 Threads**) and notice the thread ID's in the tooltip



```
5 Threads:
    308   Worker Thread
    1064  Worker Thread
    3492  Worker Thread
    5040  Worker Thread
    5752 Worker Thread
```

This tooltip makes it easier to see which specific threads are encompassed by this coalesced view. The bold row indicates the current thread (thread number 5752 in this screenshot, though it will vary on each execution).

8. Hover over the yellow arrow:

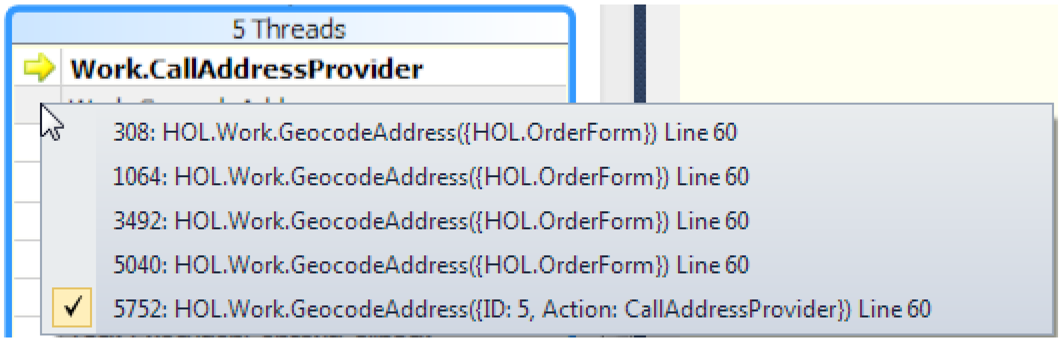


```
308: TasksHOL.exe!HOL.Work.CallAddressProvider(HOL.OrderForm form = {HOL.OrderForm}) Line 95 + 0xe bytes
1064: TasksHOL.exe!HOL.Work.CallAddressProvider(HOL.OrderForm form = {HOL.OrderForm}) Line 104 + 0xe bytes
3492: TasksHOL.exe!HOL.Work.CallAddressProvider(HOL.OrderForm form = {HOL.OrderForm}) Line 104 + 0xe bytes
5040: TasksHOL.exe!HOL.Work.CallAddressProvider(HOL.OrderForm form = {HOL.OrderForm}) Line 84 + 0xe bytes
5752: TasksHOL.exe!HOL.Work.CallAddressProvider(HOL.OrderForm form = {ID: 5, Action: CallAddressProvider}) Line 74 + 0x5 bytes
```

From the visual view you can already tell that five threads are in the **CallAddressProvider** method, but this tooltip shows the full stack frame details. This information can also be found in the **Call Stack** window, but it would require double-clicking on each thread, then remembering the details for mental comparison. The **Parallel Stacks** window makes it easy to identify the patterns with much less effort.

9. Notice the blue outline around the **5 Threads** node.
   The blue outlines around some of the nodes in the **Parallel Stacks** pane indicate the path of the current thread. Right now, the focused thread also has stack frames displayed in the **6 Threads** node, and the **9 Threads** node.

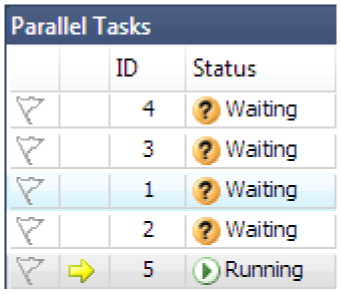
10. In the **5 Threads** node, double-click the second row from the top, **Work.GeocodeAddress**



```
308: HOL.Work.GeocodeAddress({HOL.OrderForm}) Line 60
1064: HOL.Work.GeocodeAddress({HOL.OrderForm}) Line 60
3492: HOL.Work.GeocodeAddress({HOL.OrderForm}) Line 60
5040: HOL.Work.GeocodeAddress({HOL.OrderForm}) Line 60
✓ 5752: HOL.Work.GeocodeAddress({ID: 5, Action: CallAddressProvider}) Line 60
```

When you double-click a row in the stack trace, you are switching the current stack frame and possibly the current thread of interest to the debugger. The checked row indicates which thread is already current, and the yellow arrow indicates the active

stack frame of the current thread. This affects other debugger windows such as **Locals** and **Call Stack**. Using these tools together you can pinpoint exactly what's going on at any point during code execution.
In this example, if you expected the code to be executing right now, you could easily see from the **Parallel Tasks** window that four of the tasks are waiting with only one running:

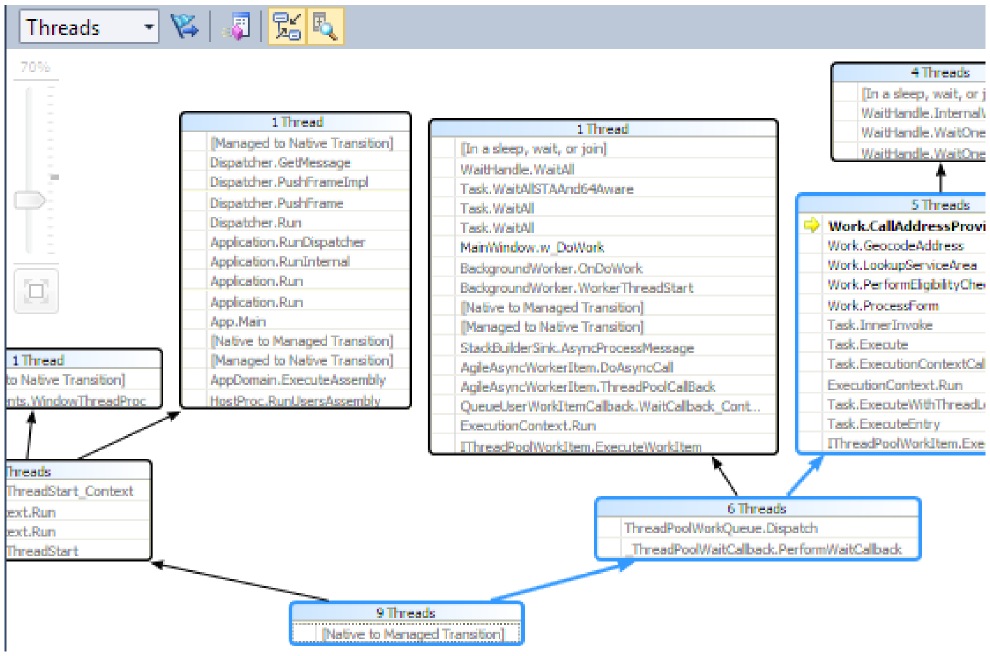

Parallel Tasks

		ID	Status
		4	Waiting
		3	Waiting
		1	Waiting
		2	Waiting
	⇨	5	Running

From the **Parallel Stacks** window, you can see exactly what path of execution each thread is taking. With these tools, you can visualize the parallel activities of your application. Instead of comparing call stack mentally or on paper, or printing out debug information, it's available in one place.

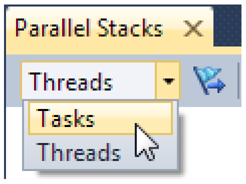
Exercise 4 – Tasks View of Parallel Stacks Window

Tasks are threads. This is oversimplified, but essentially true. If you are working with a given scheduled task, you are working with a thread. Ultimately though, working with Tasks is about simplifying your work. Threads are difficult to manage, expensive to create, and easy to get wrong. Tasks create a higher-level abstraction so you can concentrate on what you need to get done – not the low-level plumbing involved to make it happen. That being said, if you are still viewing the underlying Threads themselves, you may be looking at more than you need. In the previous exercises you viewed the call stacks based on threads. This added nodes to the view that aren't directly relevant to task-based debugging:

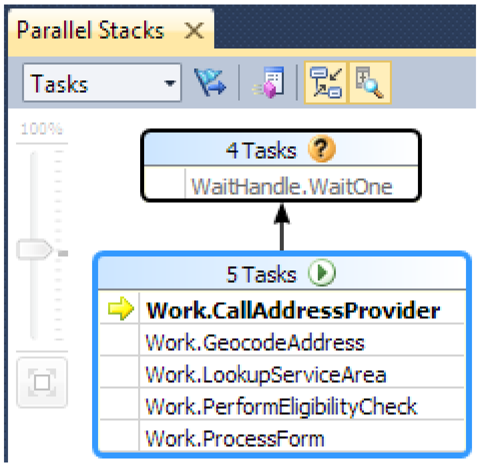


```
1 Thread
[Managed to Native Transition]
Dispatcher.GetMessage
Dispatcher.PushFrameImpl
Dispatcher.PushFrame
Dispatcher.Run
Application.RunDispatcher
Application.RunInternal
Application.Run
Application.Run
App.Main
[Native to Managed Transition]
[Managed to Native Transition]
AppDomain.ExecuteAssembly
HostProc.RunUsersAssembly

1 Thread
[In a sleep, wait, or join]
WaitHandle.WaitAll
Task.WaitAllSTAAnd64Aware
Task.WaitAll
Task.WaitAll
MainWindow.w_DoWork
BackgroundWorker.OnDoWork
BackgroundWorker.WorkerThreadStart
[Native to Managed Transition]
[Managed to Native Transition]
StackBuilderSink.AsyncProcessMessage
AgileAsyncWorkerItem.DoAsyncCall
AgileAsyncWorkerItem.ThreadPoolCallBack
QueueUserWorkItemCallback.WaitCallback_Cont...
ExecutionContext.Run
IThreadPoolWorkItem.ExecuteWorkItem

4 Threads
[In a sleep, wait, or j
WaitHandle.Internal\
WaitHandle.WaitOne
WaitHandle.WaitOne

5 Threads
Work.CallAddressProvi
Work.GeocodeAddress
Work.LookupServiceArea
Work.PerformEligibilityChe
Work.ProcessForm
Task.InnerInvoke
Task.Execute
Task.ExecutionContextCal
ExecutionContext.Run
Task.ExecuteWithThreadL
Task.ExecuteEntry
IThreadPoolWorkItem.Exe

1 Thread
to Native Transition]
nts.WindowThreadProc

Threads
ThreadStart_Context
ext.Run
ext.Run
ThreadStart

6 Threads
ThreadPoolWorkQueue.Dispatch
ThreadPoolWaitCallback.PerformWaitCallback

9 Threads
[Native to Managed Transition]
```

1. In the **Parallel Stacks** toolbar, click the drop-down box containing “Threads”.



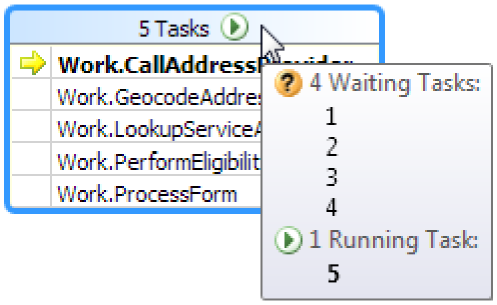
2. In the drop-down box, click **Tasks**:



```
4 Tasks
WaitHandle.WaitOne

5 Tasks
Work.CallAddressProvider
Work.GeocodeAddress
Work.LookupServiceArea
Work.PerformEligibilityCheck
Work.ProcessForm
```

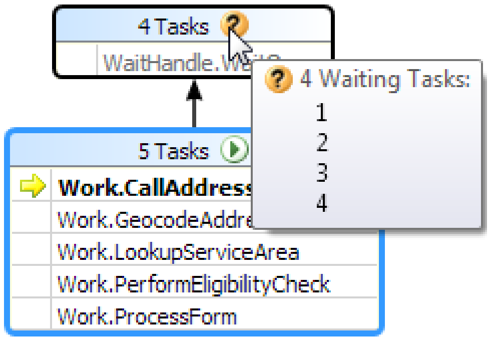
The view has just shrunk considerably. Threads responsible for system event queuing, for the **BackgroundWorker** object, and for underlying WPF management are no longer in view. You can also see the current status of a group of tasks by the icon to the right of the label, Waiting or Running in this view.

3. Hover over the **5 Tasks** node header:



The tooltip over all node headers shows you the state of the tasks that it represents. This summary makes it clear that four of the tasks are waiting, and one is running. The ID's are the same that you would see in other nodes for correlation.

4. Hover over the **3 Tasks** node header



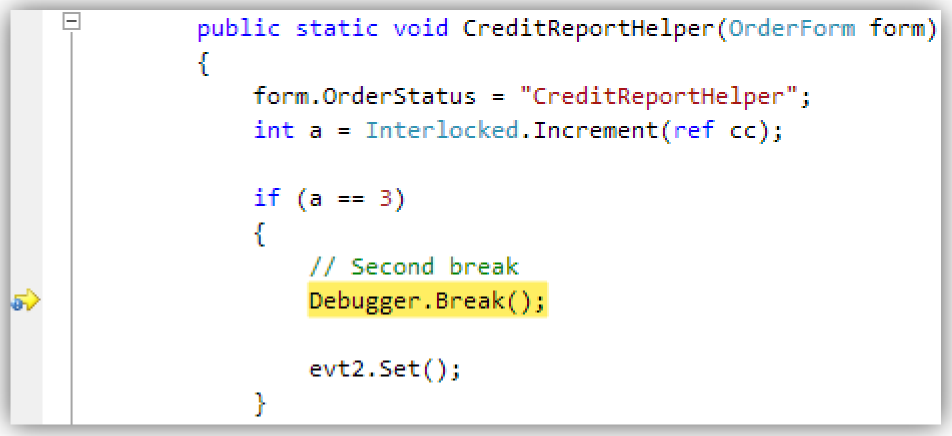
When you hover over a node which is at the top of the stack, the active stack frame, you will also see the icon for that task in the header. In other words, if the active frame yellow arrow is in a node, then the header will reflect its state in an icon.

Exercise 5 – Dealing with diverging stack frames

In Exercise 2, the code ran up to the first breakpoint. The code was written so that all threads/tasks would run in the same sequence of method calls. As the code continues, threads will move into different methods. Continue on to see how this is represented in the **Parallel Stacks** window.

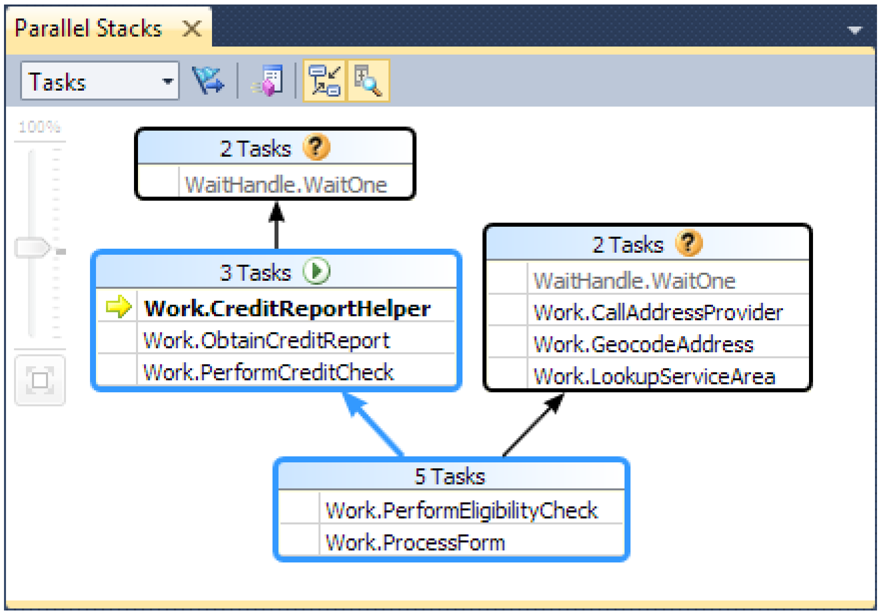
1. Continue debugging by clicking the **Debug | Continue** menu command (**F5**)
After a few moments, the debugger will break again, this time in the

CreditReportHelper method at another call to **Debugger.Break()**.



```
public static void CreditReportHelper(OrderForm form)
{
    form.OrderStatus = "CreditReportHelper";
    int a = Interlocked.Increment(ref cc);

    if (a == 3)
    {
        // Second break
        Debugger.Break();

        evt2.Set();
    }
```

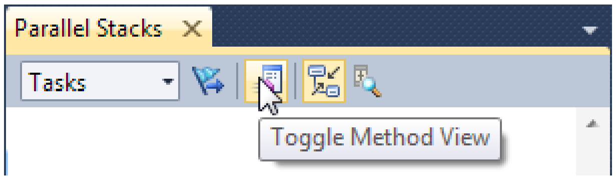
2. Switch to the **Parallel Stacks** window:



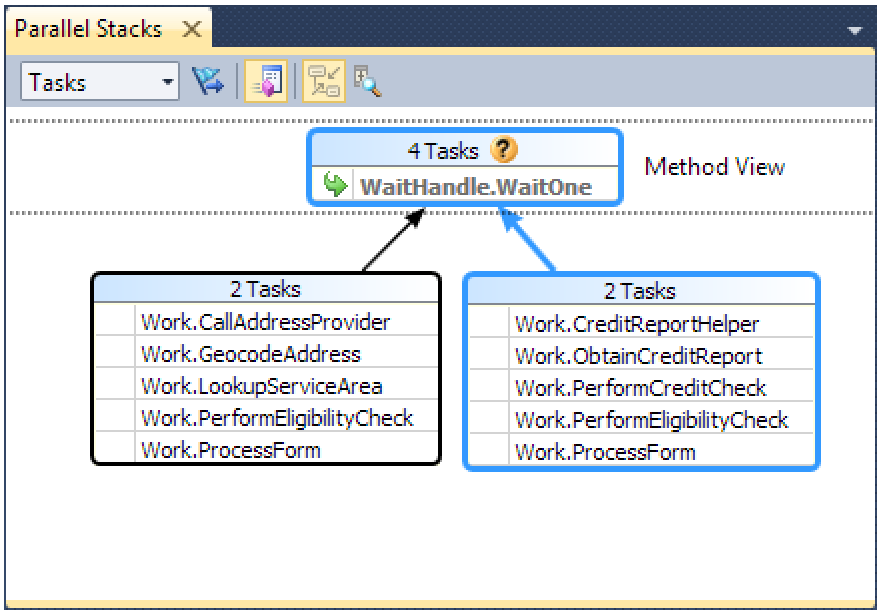
Two of the tasks are still in the **CallAddressProvider** method, while three of the them have unrolled their call stacks back to the **PerformEligibilityCheck** method, and then called into other methods. This is clear thanks to the coalesced nodes and the visual relationships between them. You can also see which threads are in a sleep/wait, or join state. In the case of tasks waiting on network connections or other resources with blocking operations, you may not be sure what is holding up execution without checking each Task's call stack and the code. The **Parallel Stacks** and **Parallel Tasks** windows both make it easier to identify this.

3. In the **2 Tasks** node, double-click **WaitHandle.WaitOne**,

4. In the toolbar, click the second button, **Toggle Method View.**



5. Notice that the **WaitHandle.WaitOne** is now centered in the diagram. The **Toggle Method View** button allows you to pivot the diagram to give you a view of your threads relative to the current method, including callers and callees.



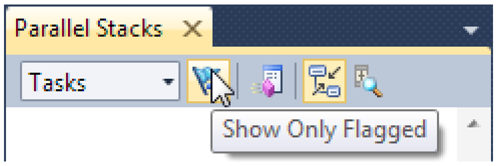
Now some of the tasks have disappeared because their call stacks don't include the **WaitHandle.WaitOne** method. Here we can see that four different call stacks called it from different methods.

6. Click **Toggle Method View** again to revert to the standard view

Exercise 6 – Flagging tasks of interest

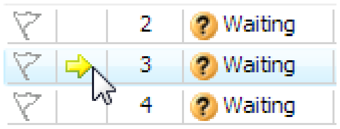
With many tasks in an application, it can be important to track a subset of one or more tasks of interest. Flagging tasks allows you to filter your view as needed.

1. In the **Parallel Stacks** toolbar, click the first button, **Show Only Flagged**



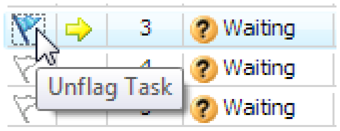
All nodes in the diagram will disappear, replaced by a message that there is nothing to show. Flagging tasks allows you to take an application with a large number of tasks and filter on the interesting ones. The next step is to actually flag some tasks.

2. In the **Parallel Tasks** window, locate the row with the yellow arrow indicating the current thread:



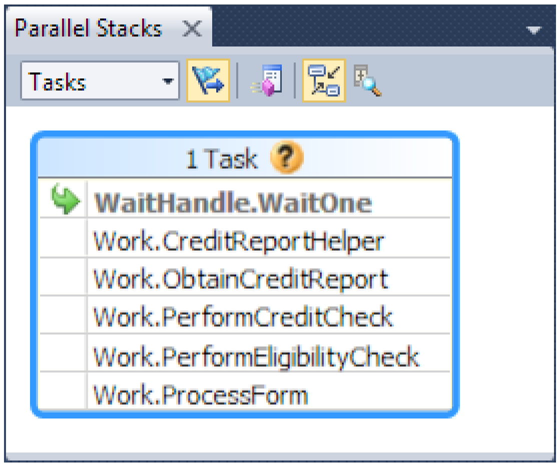
		2	Waiting
		3	Waiting
		4	Waiting

3. In the first column of that row, click the flag icon. A filled-in flag is considered to be in a "flagged" state. You can clear a flag by clicking it again.



		3	Waiting
		4	Waiting
		5	Waiting

4. In the **Parallel Stacks** window, there is now one node showing; labeled **1 Task**

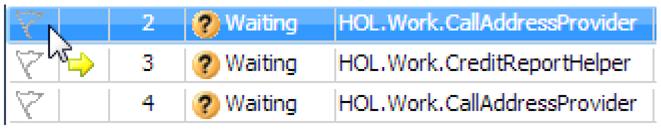


1 Task

WaitHandle.WaitOne
Work.CreditReportHelper
Work.ObtainCreditReport
Work.PerformCreditCheck
Work.PerformEligibilityCheck
Work.ProcessForm

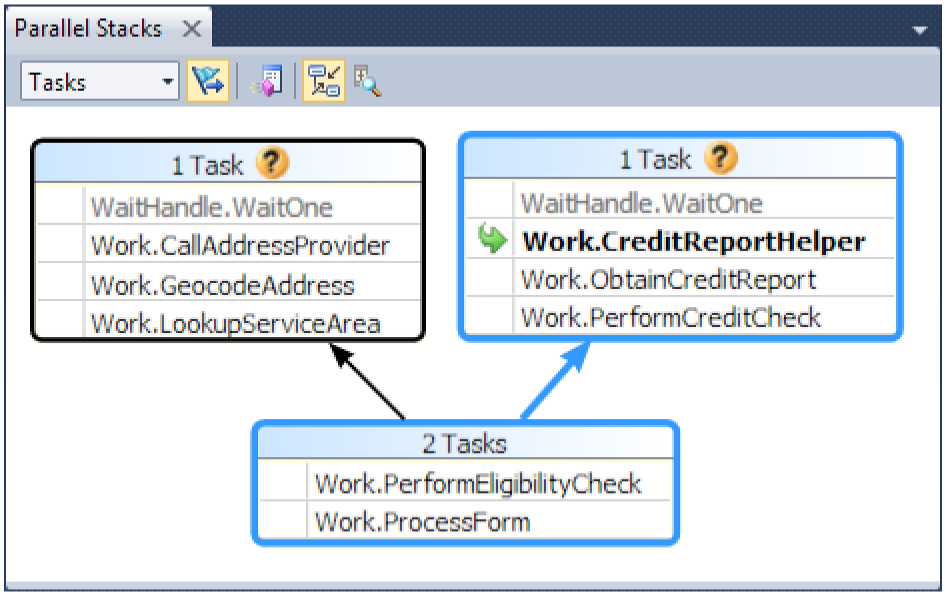
Because the diagram is only showing one task, there is no rich diagram to show – just the single call stack. This is the same information that would be seen in the **Call Stack** pane if you were to double-click the row in the **Tasks** window to focus on it.

5. In the **Tasks** window, flag either **Worker Thread** row with location at **HOL.Work.CallAddressProvider** (in this image, task 2 or 4):



		2	Waiting	HOL.Work.CallAddressProvider
		3	Waiting	HOL.Work.CreditReportHelper
		4	Waiting	HOL.Work.CallAddressProvider

6. Notice the addition of more call stack frames to the **Parallel Stacks** window:



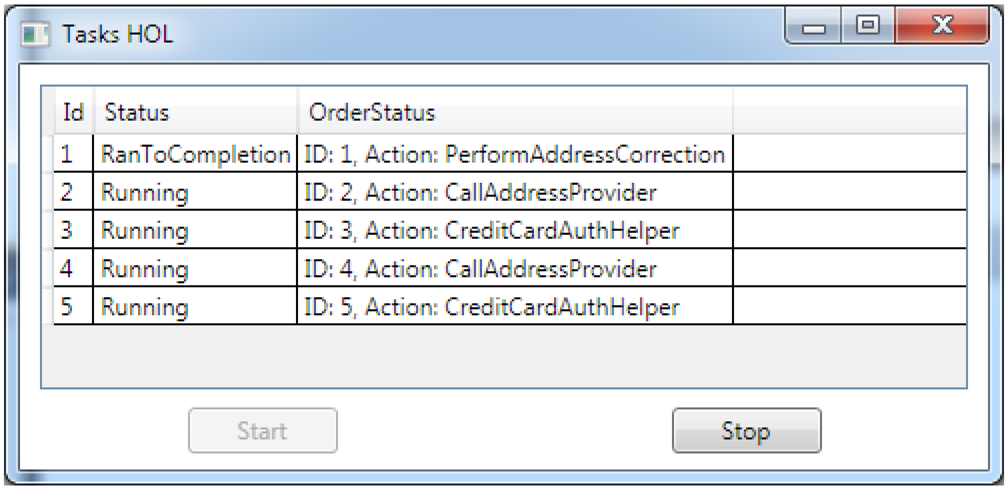
Though there are many threads and five tasks, by flagging tasks you have indicated a desire in only knowing information about these two. This is just one other way to create more order in a complex application.

7. In the **Parallel Stacks** pane, click **Show Only Flagged** again to revert to showing all the call stacks.

Exercise 7 – Dealing with Deadlocks and Other Statuses

When using the **Task** object, threads are almost a secondary consideration. Being able to see the state of each **Task** object can be very handy.

1. Resume debugging by clicking the **Debug | Continue** (**F5**) menu command
The application will continue to run, but you will notice that there are no changes to the DataGrid. All threads have stopped showing changes.

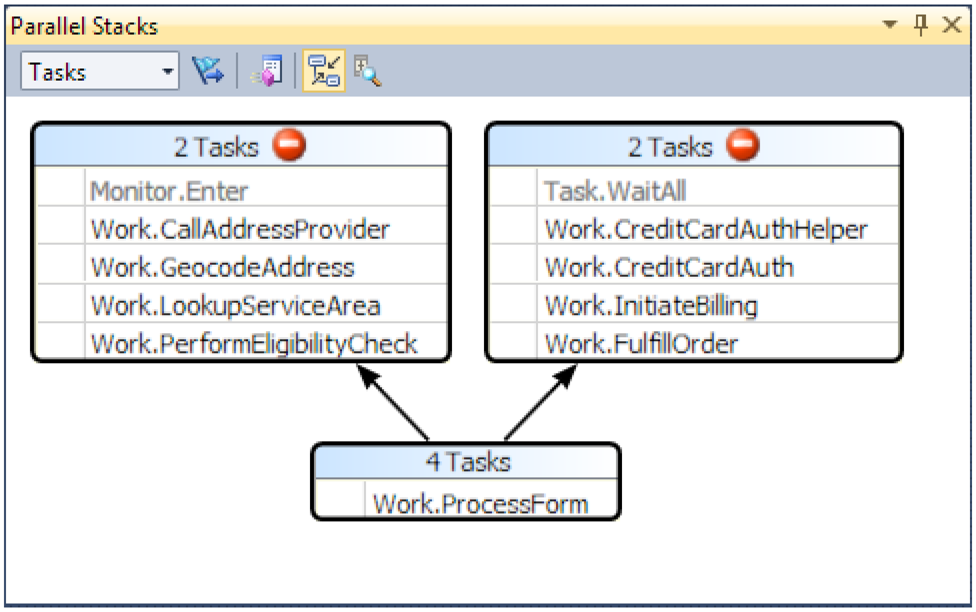


Id	Status	OrderStatus
1	RanToCompletion	ID: 1, Action: PerformAddressCorrection
2	Running	ID: 2, Action: CallAddressProvider
3	Running	ID: 3, Action: CreditCardAuthHelper
4	Running	ID: 4, Action: CallAddressProvider
5	Running	ID: 5, Action: CreditCardAuthHelper

One thread has run to completion, but the other four still show a Status of Running.

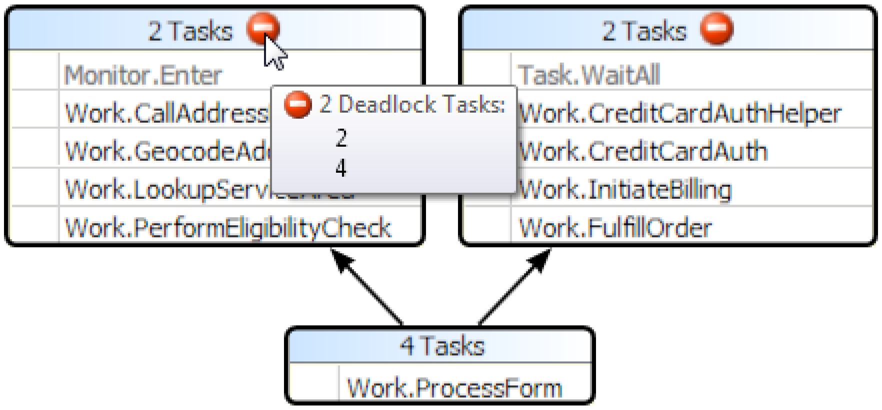
You might be tempted to wait for a possibly longer operation here, but it won't ever continue.

2. Click **Stop**
   The **Stop** button should signal the other tasks to stop, but only if they are waiting on it or checking for the indicator.
3. Click the **Debug | Break All** menu command
   Since the tasks were stalled, the only option was to break execution manually. In a robust application, you could setup monitor threads to check for such problems.
4. Notice a new status indicator on the node headers -- a red circle with a white horizontal bar. These tasks aren't running, but rather are waiting on locks that won't ever be available – a deadlock – and this icon represents this.

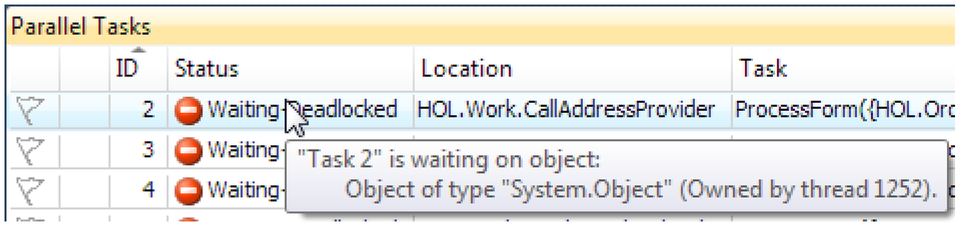


5. Hover over either of the deadlock indicators.
   The tooltip makes it clear that debugger has detected a deadlock. A deadlock is a condition that occurs when one or more threads are waiting on a lock held by another thread, at the same time that the other thread is waiting on a lock held by the first thread. These can be very tricky to detect, often only occurring at seemingly random intervals since the condition is dependent upon the ordering of threads for

each execution.



6. In the **Parallel Tasks** window, hover over the Status cell of the first task:

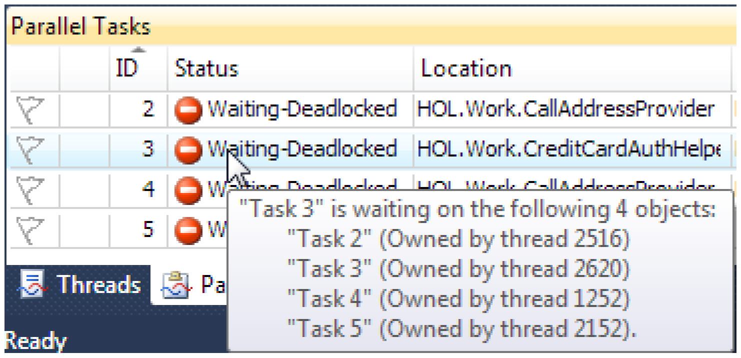


ID	Status	Location	Task
2	Waiting-Deadlocked	HOL.Work.CallAddressProvider	ProcessForm({HOL.Orc
3	Waiting-		
4	Waiting-		

"Task 2" is waiting on object:
Object of type "System.Object" (Owned by thread 1252).

There are only four tasks now and each are in a deadlocked state. The other task ran to completion so no longer shows in this view.

7. Hover over the second task:

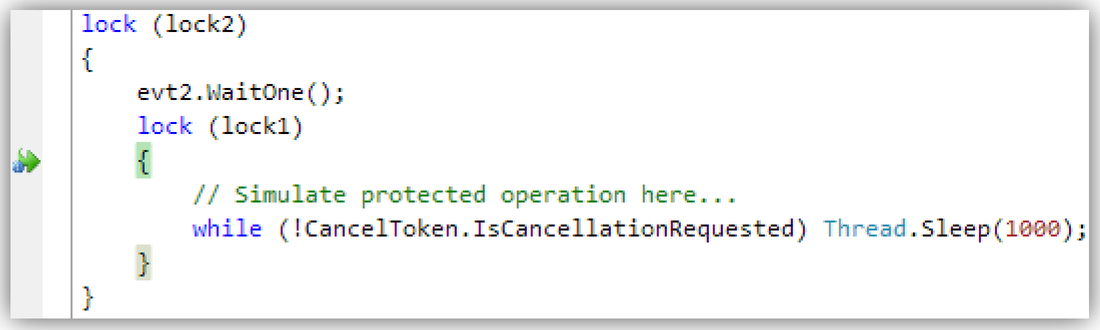


ID	Status	Location
2	Waiting-Deadlocked	HOL.Work.CallAddressProvider
3	Waiting-Deadlocked	HOL.Work.CreditCardAuthHelpe
4	W	
5	W	

"Task 3" is waiting on the following 4 objects:
"Task 2" (Owned by thread 2516)
"Task 3" (Owned by thread 2620)
"Task 4" (Owned by thread 1252)
"Task 5" (Owned by thread 2152).

The tooltips make it clear what each task's thread is waiting on – two of the tasks are waiting on the other tasks, and two of them are waiting on locks held by the other. Though there are no direct steps to take from here, in an application of a large number of tasks, you may not be aware of a deadlock when it happens ("it just locks up sometimes…"). The visual indicator and tooltip information makes this much easier to track down.

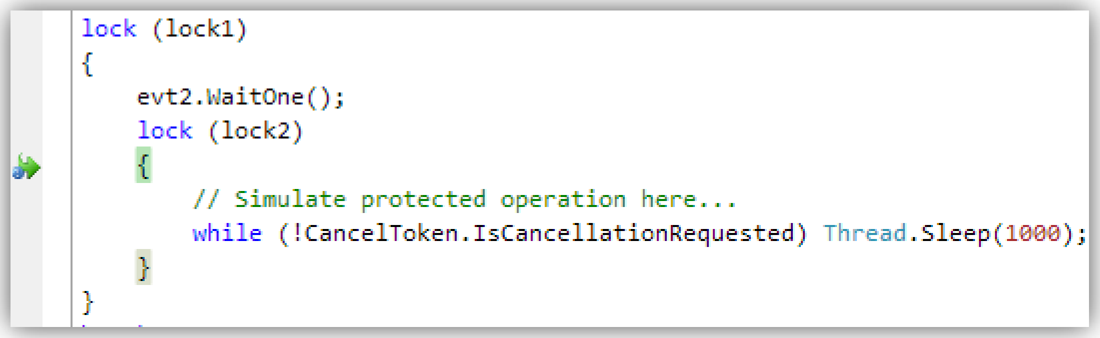
8. In the **Parallel Tasks** window, double-click the first row (ID=2):



```
lock (lock2)
{
    evt2.WaitOne();
    lock (lock1)
    {
        // Simulate protected operation here...
        while (!CancelToken.IsCancellationRequested) Thread.Sleep(1000);
    }
}
```

First the task locks **lock2**. Then it waits for the **WaitHandle** to be set before proceeding to lock **lock1**. This is a very artificial case for demonstration purposes.

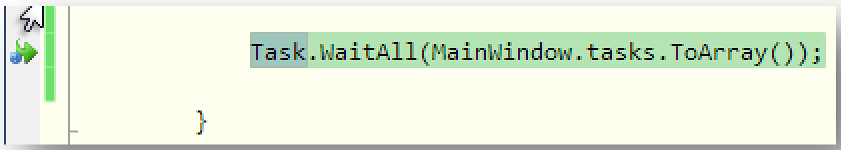
9. In the **Parallel Tasks** window, double-click the third row (ID=4):



```
lock (lock1)
{
    evt2.WaitOne();
    lock (lock2)
    {
        // Simulate protected operation here...
        while (!CancelToken.IsCancellationRequested) Thread.Sleep(1000);
    }
}
```

Each lock section is in different order. Task #2 locks **lock2**, then Task #4 locks **lock1**. This is typically not so predictable, but waiting on the WaitHandle makes it extremely likely that both will tasks will perform their locks at the same time. The next step for each task is to obtain the other lock. Of course since the other task has already done so, both are stuck. Since both are waiting on each other's lock neither will ever proceed. The fix in real-world code is to make sure that locks always occur in the same order or provide other synchronization measures.

10. In the **Parallel Tasks** window, double-click the second row (ID=3):



```
            Task.WaitAll(MainWindow.tasks.ToArray());

        }
```

The two other tasks are actually waiting on the collection of tasks to complete. Since this means that they are actually waiting on themselves and on tasks already in a deadlock, they will never recover. The Visual Studio debugger was able to detect these conditions, thereby making the fatal condition easier to deal with.

Lab Summary

In this lab, we took advantage of some of the new features of Visual Studio 2010 to improve the debugging experience for parallel applications. The .NET Framework 4 includes many enhancements to simplify parallel processing, but without appropriate debugging and troubleshooting tools, it would only make it more difficult to effectively use them. The **Parallel Stacks** window takes advantage of the fact that parallel processing threads will typically share the same call stack, thus coalescing related information into a usable form. Being able to view tasks only or all threads makes the view even more useful as you see all of the work being executed.

The **Parallel Tasks** window provides a view similar to the **Threads** window, but specializing in the **Tasks** themselves. Your choice of view will depend on the issues you are facing and the insight that you need.

Using the new framework and Visual Studio features will make it easier than ever to implement and troubleshoot parallelized applications. Only by understanding these changes can you ensure that your computationally-intensive applications will perform at their best now, and into the future.