Microsoft
Parallel Computing Platform

Hands-On Lab

Introduction to the Visual Studio 2010 Parallel Debugger

Lab version: 1.0.0

Last updated: 3/13/2010

developer & platform **evangelism**

Contents

Introduction....3
Debugging Parallel Tasks....4
Regular Debugging Tools....4
Threads Window....4
Call Stack Window....5
Variables Windows....5
What's new?....6
Parallel Tasks....6
Parallel Stacks....7
Exercise 1: Working with Parallel Debugging Tools....9
Part 1: Reviewing the Code....10
Part 2: Exploring the Parallel Tasks window....13
Part 3: Exploring the Parallel Stacks window....17
Lab Summary....22

Introduction

The Parallel Patterns Library (PPL) provides fine-grained control over how computing tasks are executed on parallel hardware.

It has never been easier for developers to take advantage of multi-core architectures. Threads have been around for years, and as platforms have gotten stronger, their usage has been easier overtime; but threads did not offer true concurrency, it was more of a “better spent” time approach for a one-core solution.

With multi-core processors, the difficulty of handling processes that would selectively run procedures on different cores has been decreasing over time. By now, **Microsoft Visual Studio 2010 Beta 2** comes with a new implementation of the Parallel Patterns Library that makes it a lot easier to create multiple tasks that are truly concurrent, working together or separately. The platform itself removes most of the concurrency implementation complexity off your hands, and lets you focus on your true work, your actual business logic.

But what happens during the development process when you are running truly concurrent procedures, that might even be sharing variables, and you want to see how they interact at runtime by debugging your application. It sure would be nice if you could place breakpoints and see how each of those tasks is doing at any given point. With **Microsoft Visual Studio 2010 Beta 2** you can, and it is as easy as it can be.

Debugging Parallel Tasks

When an application is working with true concurrency by distributing the workloads through multiple cores in one system, debugging your application can become a different task than what you are used to with an application that does not implement concurrency.

If an application does not use concurrency or threading, there is only one instruction being executed at any given time, although that is also technically true for threads. However, when an application creates and uses two or four concurrent tasks, these tasks can potentially use up to two or four cores, respectively. In these cases the application is actually running two or four instructions at any given time, not just one.

Have you ever seen that pause button when you are debugging an application in Visual Studio? When you click on that button, or when you place a breakpoint in your code and the execution pauses for you, all threads and tasks pause too, giving you the chance to watch the values of variables, and the call stack helps you witness the order in which the instructions have been executed.

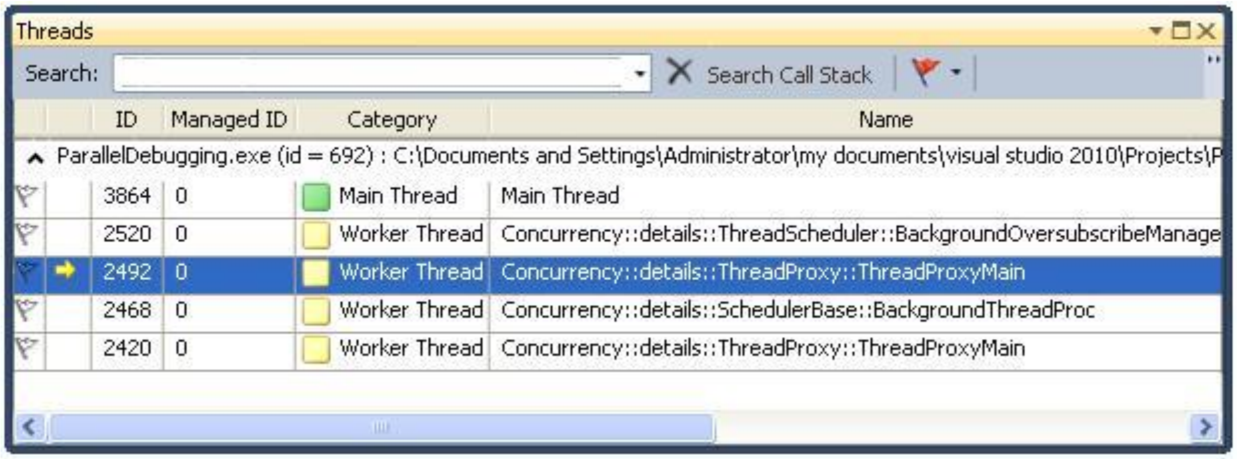
Regular Debugging Tools

The history of software development has demonstrated time and time again that programming models benefit greatly from exemplary debugging support, and **Visual Studio 2010** delivers in this regard by providing two new debugging tool windows to assist with task-based parallel programming. But before we look at these new features, let's review the debugging experience in Visual Studio today to set the stage.

The main debugging windows that developers use in Visual Studio are the Threads window, the Call Stack window, and the variable windows (Locals, Autos, Watch).

Threads Window

Displays a list of all the threads in your process, including information such as the thread ID and thread priority and an indication (a yellow arrow) of the current thread, which by default is the thread that was executing when the debugger broke into the process. Probably the most important information about a thread is where it was executing when the debugger halted its execution, shown by the call stack frame in the Location column. Hovering your cursor over that column reveals the equally important call stack -- the series or method calls that the thread was in the process of executing before reaching the current location.



ParallelDebugging.exe (id = 692) : C:\Documents and Settings\Administrator\my documents\visual studio 2010\Projects\P

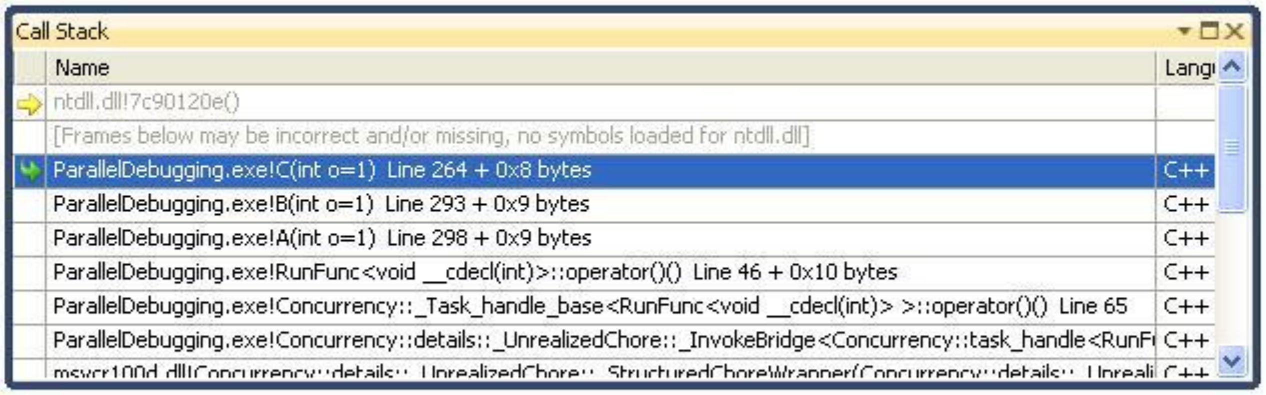
		ID	Managed ID	Category	Name
		3864	0	🟩 Main Thread	Main Thread
		2520	0	🟨 Worker Thread	Concurrency::details::ThreadScheduler::BackgroundOversubscribeManage
	➡	2492	0	🟨 Worker Thread	Concurrency::details::ThreadProxy::ThreadProxyMain
		2468	0	🟨 Worker Thread	Concurrency::details::SchedulerBase::BackgroundThreadProc
		2420	0	🟨 Worker Thread	Concurrency::details::ThreadProxy::ThreadProxyMain

Figure 1

Threads window with yellow arrow signaling current thread

Call Stack Window

Displays the call stack of the current thread, provides much richer information about the call stack, including interaction opportunities. To see the call stack of another thread; simply click on that thread in the Threads window. At the top of the call stack you will find the instruction that was being executed at the moment of the break.



Call Stack

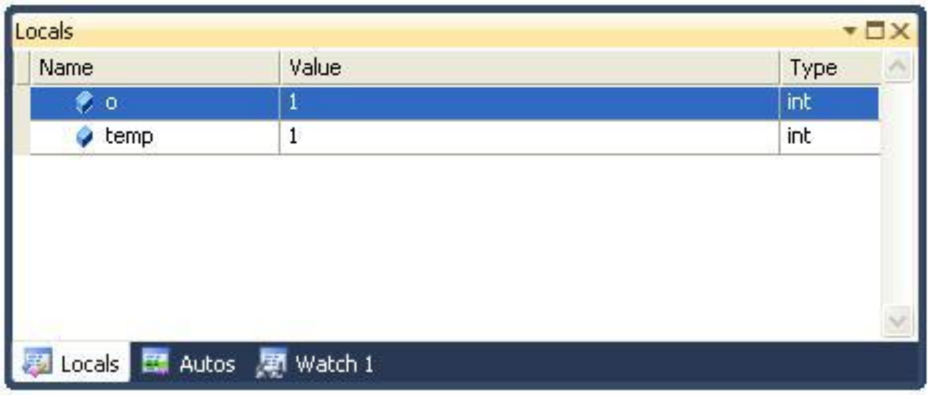
	Name	Langı
⇨	ntdll.dll!7c90120e()	
	[Frames below may be incorrect and/or missing, no symbols loaded for ntdll.dll]	
➜	ParallelDebugging.exe!C(int o=1) Line 264 + 0x8 bytes	C++
	ParallelDebugging.exe!B(int o=1) Line 293 + 0x9 bytes	C++
	ParallelDebugging.exe!A(int o=1) Line 298 + 0x9 bytes	C++
	ParallelDebugging.exe!RunFunc<void __cdecl(int)>::operator()() Line 46 + 0x10 bytes	C++
	ParallelDebugging.exe!Concurrency::_Task_handle_base<RunFunc<void __cdecl(int)> >::operator()() Line 65	C++
	ParallelDebugging.exe!Concurrency::details::_UnrealizedChore::_InvokeBridge<Concurrency::task_handle<RunFı	C++
	msvcr100d.dll!Concurrency::details::_UnrealizedChore::_StructuredChoreWrapper(Concurrency::details::_Unreali	C++

Figure 2

Call Stack signaling the instruction being executed for the current thread

Variables Windows

The variable windows are used to inspect the values of variables in your application. The variables of local methods are usually browsed in the **Locals** and **Autos** windows; global state (variables not declared in a method) can be examined by adding them to the **Watch** window. To examine variables that were in scope earlier in the call stack of the thread, you need to change the current stack frame by double-clicking the stack frame you want to examine in the Call Stack window.



Locals

Name	Value	Type
o	1	int
temp	1	int

Figure 3

Variables windows with locals, autos, and watch; exposing values for variables of the current scope

In summary, when you break into your process in the debugger, you can very easily inspect the variables in scope at the executing method of one of the threads. However, to create a complete picture of where all your threads are executing, you need to individually examine the calls stack of each thread by double-clicking each thread to make it current, looking at the Call Stack window, and then creating the holistic picture mentally. Furthermore, to examine variables on various stack frames of various threads, two levels of indirection are needed again: switch threads and then switch frames.

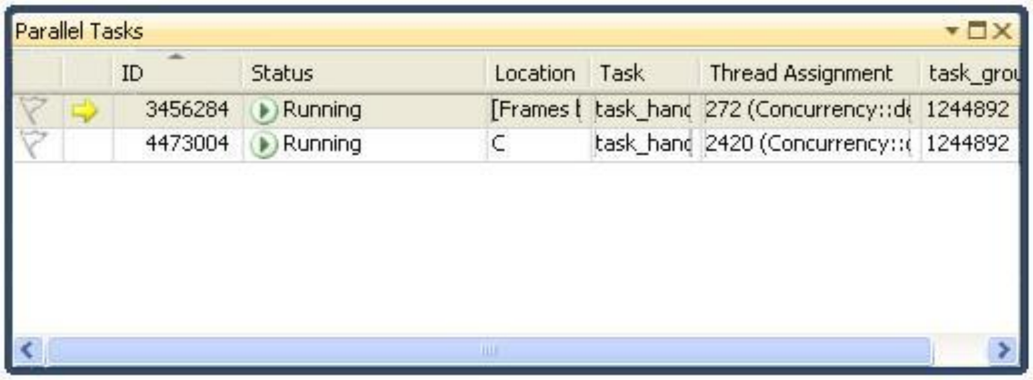
What's new?

So this all sounds very good and dandy, but you have probably seen it before in previous versions of Visual Studio. Nowadays, the multi-core approach has changed the paradigm in which CPU consuming applications can gain a better performance, and therefore, more developers out there would struggle with the complexity of having to debug their applications through several concurrent tasks.

Suddenly the idea of creating a mental picture out of the Threads and Call Stack windows seems to be unnecessarily insufficient. **Microsoft Visual Studio 2010** launches a new set of windows that can help you see a graphic representation of those tasks, and how they interact and evolve over time.

Parallel Tasks

If your application creates tasks rather than threads, you can switch to a task-centric view. This new debugger window exposes additional information about tasks, including the task ID, the thread assigned to the task, the current Location, and the entry point (the delegate) passed to the task at creation. This window, called the Parallel Tasks window, exposes features similar to the Threads window, such as indicating the current task (the top-most task running on the current thread), the ability to switch the current task, flagging of tasks, and freezing and thawing threads.



		ID	Status	Location	Task	Thread Assignment	task_grou
	⇨	3456284	Running	[Frames t	task_hanc	272 (Concurrency::de	1244892
		4473004	Running	C	task_hanc	2420 (Concurrency::(	1244892

Figure 4

Parallel Tasks window listing two tasks running off the same task group.

You can double click on any of the listed tasks, the yellow arrow points to the task that you wish to debug. The Threads and Call Stack windows will also change accordingly.

Each Task has a unique **Id** which can be retrieved programmatically by calling the Id property (for C++ tasks, this will be the memory address). This is useful so you can map the task you are seeing in the window with diagnostic output you may send to some stream; it is also useful for cross-referencing tasks between the Parallel Tasks and the Parallel Stacks windows (which I'll describe in another post).

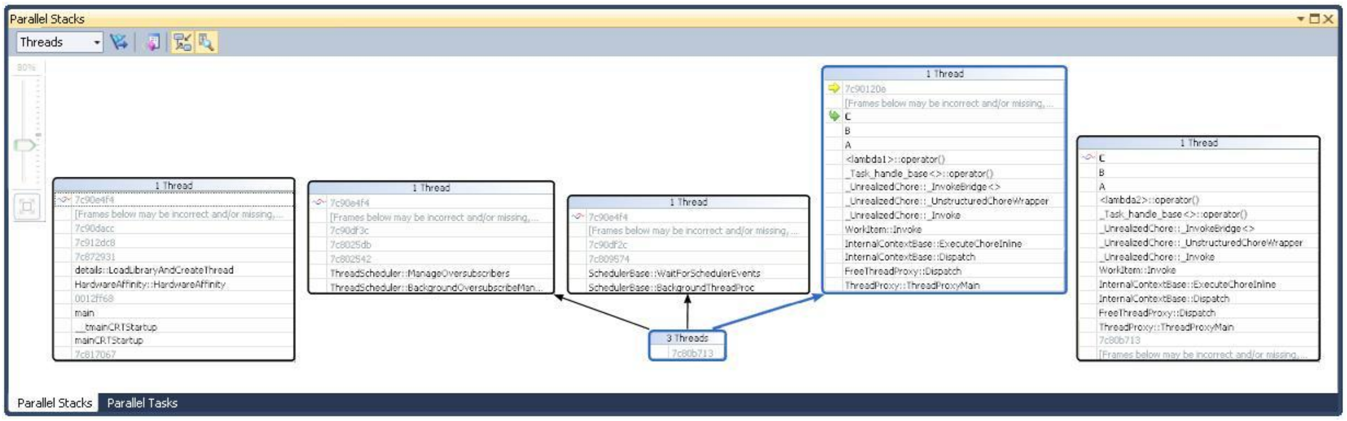
As for the Status, the 4 potential values here are Running, Scheduled, Waiting and Waiting-Deadlocked.

- Running: tasks that are executing code at the moment your app breaks in the debugger (they are at top of stack of a running thread).
- Scheduled: tasks that are sitting in some queue but have not been executed by a thread yet.
- Waiting: are the ones that have run, but are now blocked on something, e.g. a Monitor, a critical section.
- Waiting-Deadlocked: tasks that are "Waiting" and additionally we have detected a wait chain with other tasks in the list.

The Location column displays the method that the task is currently executing (i.e. the top most user frame of the task's call stack). Hovering over the cell displays in a stacktip the entire call stack for the task (not the entire call stack for the thread!) and from the stacktip you can even switch the current stack frame by double clicking on your chosen one. Naturally, Scheduled tasks do not have a value in this column.

Parallel Stacks

Much in the same way that the Parallel Tasks window provides a parallel-based version of the original Tasks window, the Parallel Stacks window also presents a specialized version of the Call Stack window. In this view, call stacks of threads not executing tasks are omitted. Additionally, a single-thread call stack, as perceived from the original Call Stack window could include two or three tasks; tasks that you may want to split out and view separately. A special feature of the Parallel Stacks window allows you to pivot the diagram on a single method and clearly observe the callers and callees of that method context.



```
1 Thread
7c90e4f4
[Frames below may be incorrect and/or missing,...
7c90dacc
7c912dc8
7c872931
details::LoadLibraryAndCreateThread
HardwareAffinity::HardwareAffinity
0012ff68
main
__tmainCRTStartup
mainCRTStartup
7c817067
```

```
1 Thread
7c90e4f4
[Frames below may be incorrect and/or missing,...
7c90df3c
7c8025db
7c802542
ThreadScheduler::ManageOversubscribers
ThreadScheduler::BackgroundOversubscribeMan...
```

```
1 Thread
7c90e4f4
[Frames below may be incorrect and/or missing,...
7c90df2c
7c809574
SchedulerBase::WaitForSchedulerEvents
SchedulerBase::BackgroundThreadProc
```

```
1 Thread
7c90120e
[Frames below may be incorrect and/or missing,...
C
B
A
<lambda1>::operator()
_Task_handle_base<>::operator()
_UnrealizedChore::_InvokeBridge<>
_UnrealizedChore::_UnstructuredChoreWrapper
_UnrealizedChore::_Invoke
WorkItem::Invoke
InternalContextBase::ExecuteChoreInline
InternalContextBase::Dispatch
FreeThreadProxy::Dispatch
ThreadProxy::ThreadProxyMain
```

```
3 Threads
7c80b713
```

```
1 Thread
C
B
A
<lambda2>::operator()
_Task_handle_base<>::operator()
_UnrealizedChore::_InvokeBridge<>
_UnrealizedChore::_UnstructuredChoreWrapper
_UnrealizedChore::_Invoke
WorkItem::Invoke
InternalContextBase::ExecuteChoreInline
InternalContextBase::Dispatch
FreeThreadProxy::Dispatch
ThreadProxy::ThreadProxyMain
7c80b713
[Frames below may be incorrect and/or missing,...
```

Figure 5

Parallel Stacks window showing all the threads of the application.

Do you see too many unnecessary threads in the Parallel Stacks window above? The window shows all threads of your application by default. To narrow it down there is a couple of things that you can do. Threads can be flagged in the Threads window, and you can make the Parallel Stacks window show you only flagged threads.

But in the spirit of a more task-centric approach, you can also choose to see tasks only, instead of threads in general.

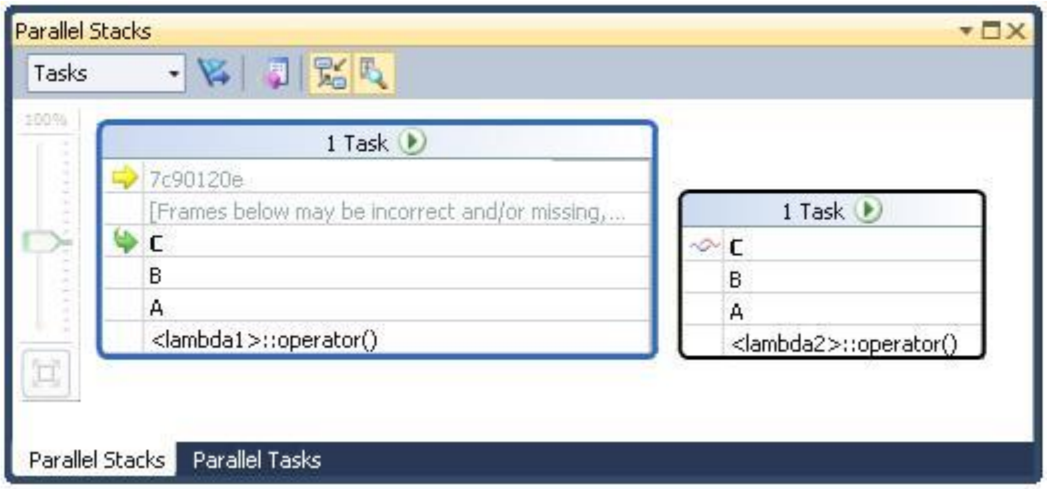


Figure 6

Parallel Stacks window showing tasks only.

When placed in *Tasks* mode, the Parallel Stacks window presents tasks with their respective call stacks.

This is a much cleaner view of the tasks running in your application, it excludes threads that may fall out of the scope of what you want to debug. This closer look also let's us appreciate the fact that this window not only shows the tasks, but it also presents a single view of the "call stacks" of each task or thread. Do you notice any differences between the Call Stacks from Figures 5 and 6 above? When in *Threads* mode of the Parallel Stacks window, you can see your regular call stacks in the diagram, while the *Tasks* mode does not show the calls that were not part of the tasks themselves.

You can also flag tasks in the Parallel Tasks window as a way of selecting which tasks to show in the diagram.

Exercise 1: Working with Parallel Debugging Tools

In this exercise you will use **Microsoft Visual Studio 2010 Beta 2** to debug an application that runs multiple parallel tasks. This improved version of Visual Studio has a new set of tools that will help you easily visualize the call stacks for threads, and tasks.

Throughout the following steps you will also see the different options that can be used to speed up the development process by inspecting and manipulating those tasks, changing the current task, inspecting the variables and how they have changed overtime, sorting the tasks, and grouping them to help you better understand the ultimate workflow of your application.

The C++ code that will be used for this exercise does nothing in particular. It is simply a way of showing the different options available from the Parallel Tasks and Parallel Stacks windows. For simplicity purposes, and to go through all the steps, we need specific break points on specific pieces of the code; but you will not have to add those breakpoints. Instead, the code launches its own break points from code through the **DebugBreak** method.

Part 1: Reviewing the Code

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. Click **Yes** when prompted by **UAC**.
5. Once Visual Studio is opened, select:

```
File -> Open -> Project/Solution…
```

6. Open the **ParallelDebugging** solution found on the following path:

```
C:\Server 2008 R2 Labs\Introduction to Parallel Debugging\ParallelDebugging
```

7. From the Solution Explorer, in Visual Studio 2010, click on the **Source Files** node to expand it.

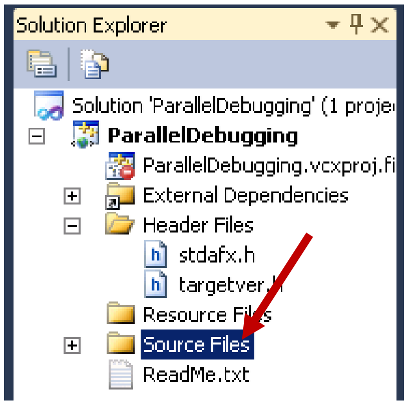


Figure 7

Solution Explorer - Source Files node

8. Double click on the **ParallelDebugging.cpp** source file to open the code that we are going to analyze:

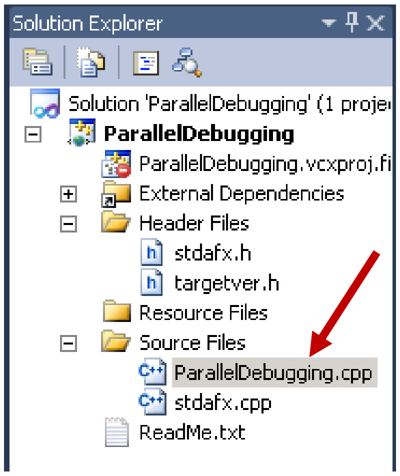
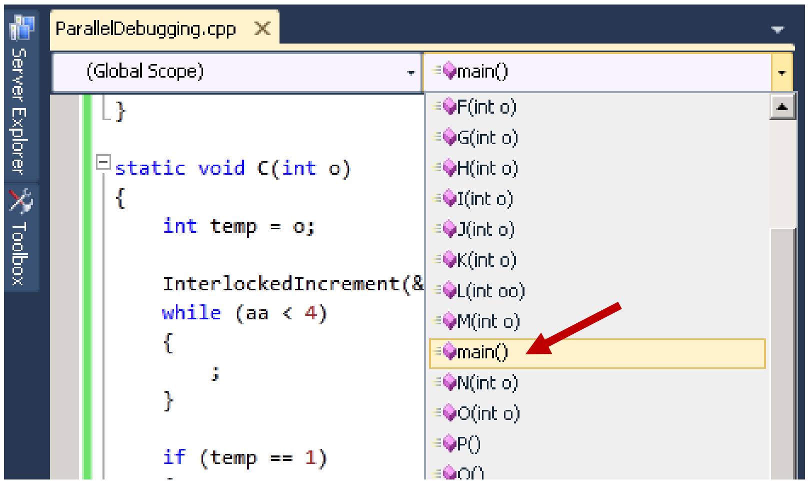


Figure 8

Solution Explorer - ParallelDebugging.cpp source file

Note: The displayed code contains a main thread that creates four tasks. The file contains its own code built in breakpoints, and these tasks will flow through a series of methods whose only purpose is to create an execution path or call stack for each of those tasks.

9. Right above the code you should see a couple of drop downs that can help you navigate the code more easily. Click on the second drop down and select the **main** method.



```
}

static void C(int o)
{
    int temp = o;

    InterlockedIncrement(&
    while (aa < 4)
    {
        ;
    }

    if (temp == 1)
```

Figure 9

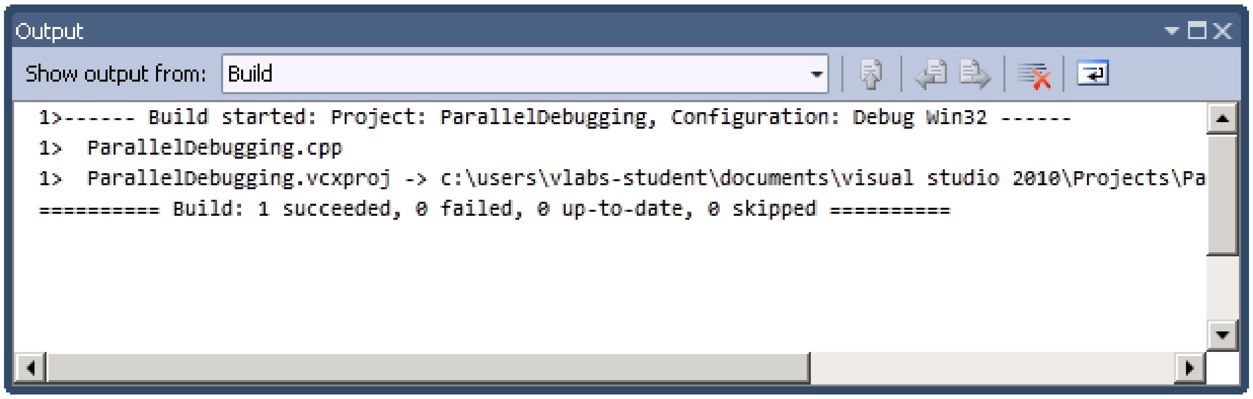
Using navigation options in Visual Studio to locate the main method

10. The cursor should jump to the **main** method. In it you will find a few lines that create four tasks and runs them:

```
task_group tasks;
task_handle<RunFunc<decltype(A)>> t1(RunFunc<decltype(A)>(A,1));
tasks.run(t1);
task_handle<RunFunc<decltype(A)>> t2(RunFunc<decltype(A)>(A,2));
task2.run(t2);
task_handle<RunFunc<decltype(A)>> t3(RunFunc<decltype(A)>(A,3));
task3.run(t3);
task_handle<RunFunc<decltype(A)>> t4(RunFunc<decltype(A)>(A,4));
task4.run(t4);
```

Note: All four tasks start their execution by calling the **A** method, which takes an integer parameter referred to as **o**. Depending on the value of this variable, the execution of each task will differ, but they will all stop at breakpoints that have been entered into the code itself, so you can look at the different capabilities of the Parallel Tasks, and Parallel Stacks windows. We encourage you to review the methods and study their implementation; however, it is not completely required for this exercise, as the Parallel Tasks window will paint the picture for you.

11. Compile the project by selecting **Build Solution** from the **Build** main menu.
12. Wait for the code to be compiled; by the end of it you should see results similar to the following in the **Output** window:



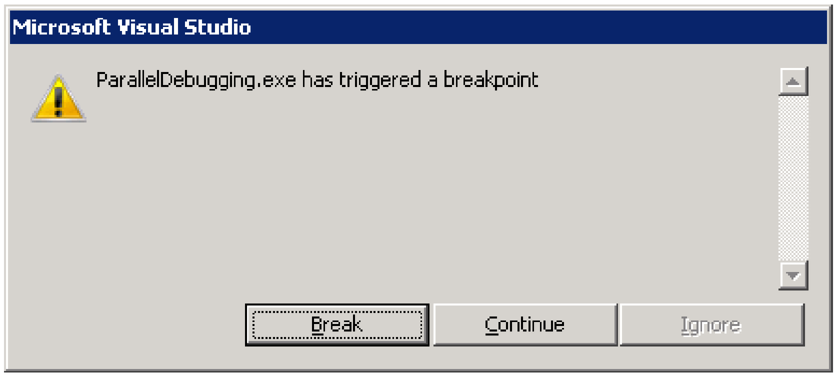
```
1>------ Build started: Project: ParallelDebugging, Configuration: Debug Win32 ------
1>  ParallelDebugging.cpp
1>  ParallelDebugging.vcxproj -> c:\users\vlabs-student\documents\visual studio 2010\Projects\Pa
========== Build: 1 succeeded, 0 failed, 0 up-to-date, 0 skipped ==========
```

Figure 10

Output messages for the successful compilation of the project

Part 2: Exploring the Parallel Tasks window

1. Run the application in debug mode by selecting the **Start Debugging…** option from the **Debug** menu.
2. This is a command prompt application. A short time after it starts running you should see a message similar to the following:

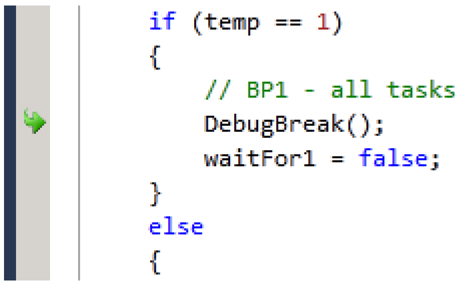

Microsoft Visual Studio

ParallelDebugging.exe has triggered a breakpoint

Figure 11

Confirmation box for break point added in code through the DebugBreak method

3. Click on the **Break** button to confirm that you do want to pause the execution.
4. You should see a green arrow pointing to the DebugBreak call that triggered this break.

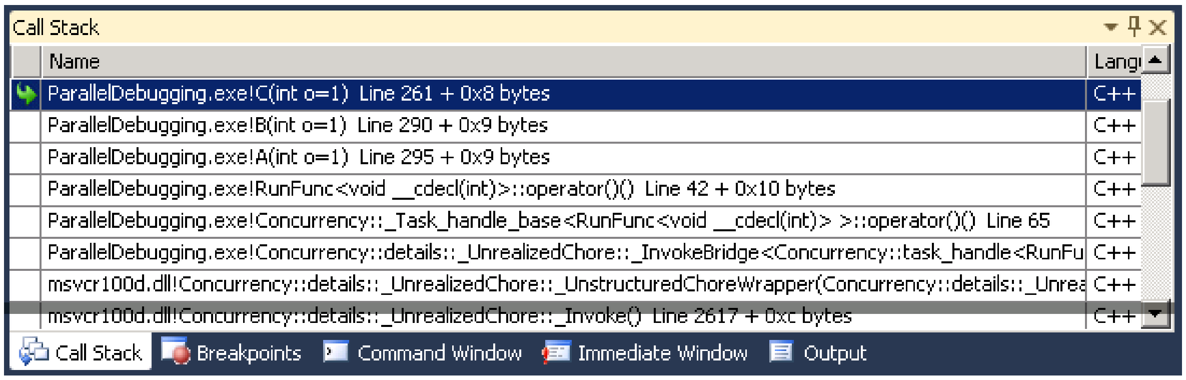


```
if (temp == 1)
{
    // BP1 - all tasks
    DebugBreak();
    waitFor1 = false;
}
else
{
```

Figure 12

Green arrow pointing to the DebugBreak call that triggered the break.

5. Make sure that the **Call Stack** window is open. To do this, go the **Debug** menu, expand on the **Windows** submenu, and then select the **Call Stack** option.


Call Stack

Name	Lang
ParallelDebugging.exe!C(int o=1) Line 261 + 0x8 bytes	C++
ParallelDebugging.exe!B(int o=1) Line 290 + 0x9 bytes	C++
ParallelDebugging.exe!A(int o=1) Line 295 + 0x9 bytes	C++
ParallelDebugging.exe!RunFunc<void __cdecl(int)>::operator()() Line 42 + 0x10 bytes	C++
ParallelDebugging.exe!Concurrency::_Task_handle_base<RunFunc<void __cdecl(int)> >::operator()() Line 65	C++
ParallelDebugging.exe!Concurrency::details::_UnrealizedChore::_InvokeBridge<Concurrency::task_handle<RunFu	C++
msvcr100d.dll!Concurrency::details::_UnrealizedChore::_UnstructuredChoreWrapper(Concurrency::details::_Unrea	C++
msvcr100d.dll!Concurrency::details::_UnrealizedChore::_Invoke() Line 2617 + 0xc bytes	C++

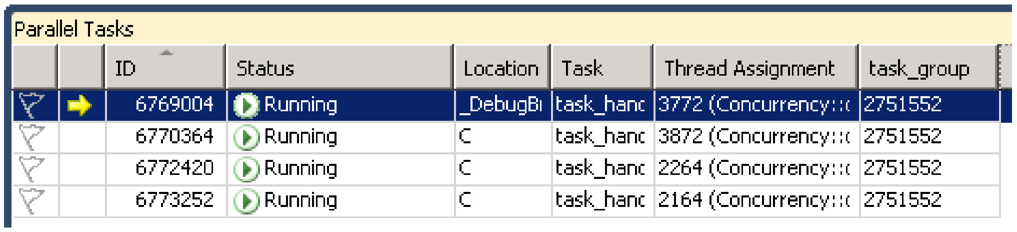
Figure 13

Call Stack window showing the active call at the moment of the break.

Note: You should see a green arrow again. The arrow is pointing to the same DebugBreak instruction as seen before. Also notice that it tells us the name of the method that contains this instruction. In this case the execution stopped with the DebugBreak call in the **C** method. And digging a little deeper you can see that the **o** parameter taken by the **C** method is equal to 1, which means that the breakpoint was requested by the first task created in the main method, which we will refer to as task #1.

6. In the **Debug** menu, expand on the **Windows** submenu and select the **Parallel Tasks** option.

Note: You should see the Parallel Tasks window, and it should show the four tasks that were launched from the **main** method.

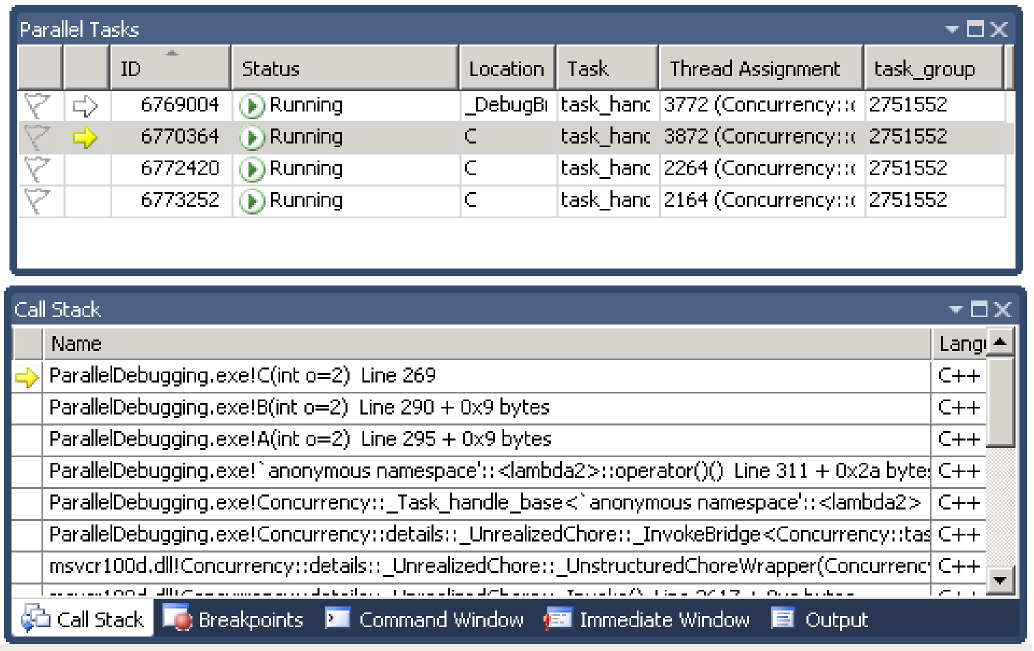

Parallel Tasks

		ID	Status	Location	Task	Thread Assignment	task_group
		6769004	Running	_DebugBı	task_hanc	3772 (Concurrency::(	2751552
		6770364	Running	C	task_hanc	3872 (Concurrency::(	2751552
		6772420	Running	C	task_hanc	2264 (Concurrency::(	2751552
		6773252	Running	C	task_hanc	2164 (Concurrency::(	2751552

Figure 14

Parallel tasks showing four running tasks.

7. Double click on each of the tasks in the Parallel Tasks window, and while you do, look at the changes in the Call Stack window.


Parallel Tasks

		ID	Status	Location	Task	Thread Assignment	task_group
		6769004	Running	_DebugBı	task_hanc	3772 (Concurrency::(	2751552
		6770364	Running	C	task_hanc	3872 (Concurrency::(	2751552
		6772420	Running	C	task_hanc	2264 (Concurrency::(	2751552
		6773252	Running	C	task_hanc	2164 (Concurrency::(	2751552

Call Stack

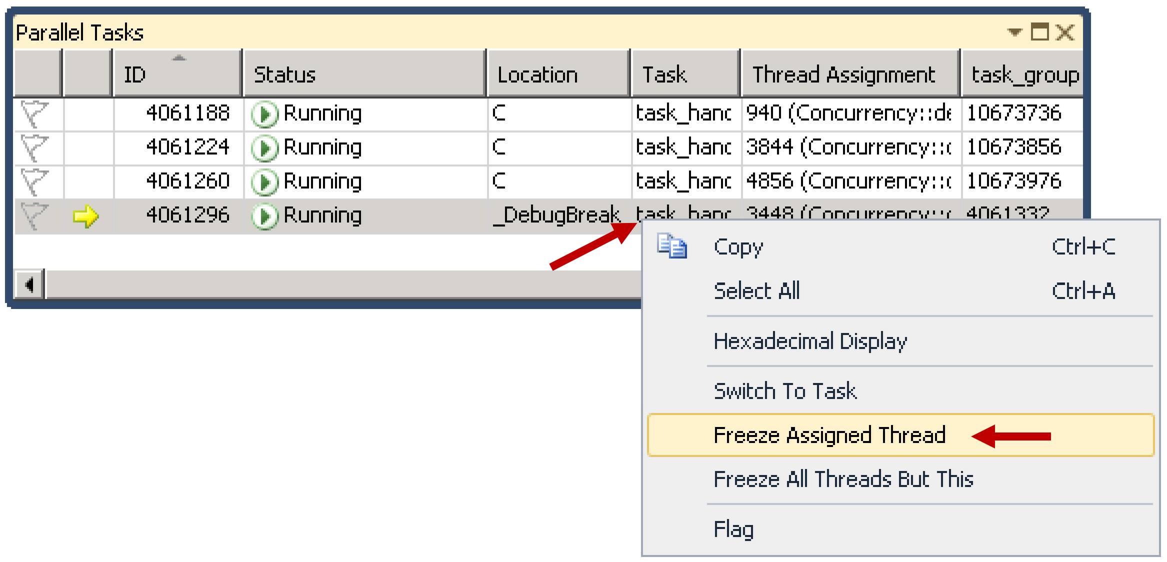
	Name	Langı
	ParallelDebugging.exe!C(int o=2) Line 269	C++
	ParallelDebugging.exe!B(int o=2) Line 290 + 0x9 bytes	C++
	ParallelDebugging.exe!A(int o=2) Line 295 + 0x9 bytes	C++
	ParallelDebugging.exe!`anonymous namespace'::<lambda2>::operator()() Line 311 + 0x2a byte:	C++
	ParallelDebugging.exe!Concurrency::_Task_handle_base<`anonymous namespace'::<lambda2>	C++
	ParallelDebugging.exe!Concurrency::details::_UnrealizedChore::_InvokeBridge<Concurrency::tas	C++
	msvcr100d.dll!Concurrency::details::_UnrealizedChore::_UnstructuredChoreWrapper(Concurrenc	C++

Figure 15

Call Stack changes while switching the currently selected task.

Note: All tasks started on **A**, then **B**, and then they all stopped in the **C** method. The code has been written so that all tasks hit the break point on this method. Task #1 triggers the break, but the rest are just waiting for task #1 to react after the break point. You should also notice how the **o** parameter that is passed to all the methods represents the task number as specified in the main method. If you kept running the application all threads would continue to their next respective calls, out of the C method.

8. Right click on the task that has the **_DebugBreak** location, and select the **Freeze Assigned Thread**.



Parallel Tasks

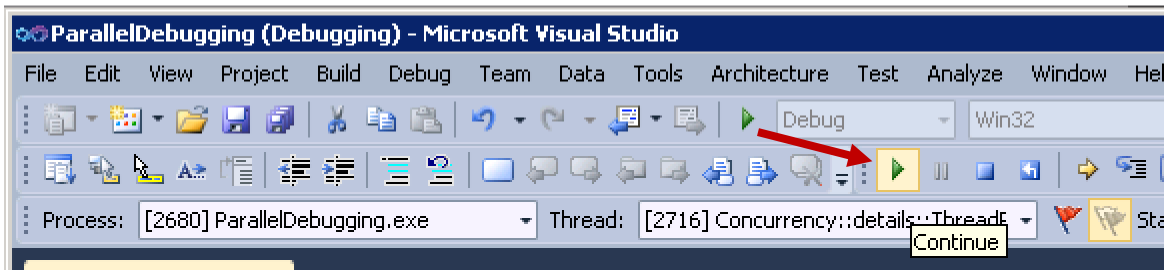
		ID	Status	Location	Task	Thread Assignment	task_group
		4061188	Running	C	task_hanc	940 (Concurrency::de	10673736
		4061224	Running	C	task_hanc	3844 (Concurrency::c	10673856
		4061260	Running	C	task_hanc	4856 (Concurrency::c	10673976
		4061296	Running	_DebugBreak	task_hanc	3448 (Concurrency::c	4061332

Figure 16

Task options during execution.

Note: With this action you are freezing Task #1, which is the task that triggered the break point. Remember that all other tasks are simply waiting for Task #1 to resume its execution.

9. Locate and click on the **Continue** option from the buttons on top, as signaled in the picture below.



ParallelDebugging (Debugging) - Microsoft Visual Studio

Process: [2680] ParallelDebugging.exe Thread: [2716] Concurrency::details::ThreadP

File Edit View Project Build Debug Team Data Tools Architecture Test Analyze Window Hel
Debug
Win32
Continue
Sta

Figure 17

Locating the Continue button.

Note: If you had not frozen task #1, by now the application would have hit a second breakpoint. The console application got focused and the remaining tasks are simply waiting for task #1 to continue.

10. Go back to the Visual Studio window.

11. Locate and click on the **Break All** option from the buttons on top, as signaled in the picture below.

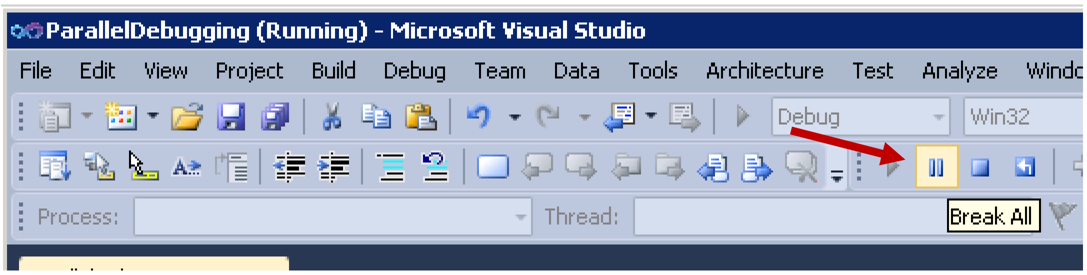

ParallelDebugging (Running) - Microsoft Visual Studio
File Edit View Project Build Debug Team Data Tools Architecture Test Analyze Windo
Debug
Win32
Process:
Thread:
Break All

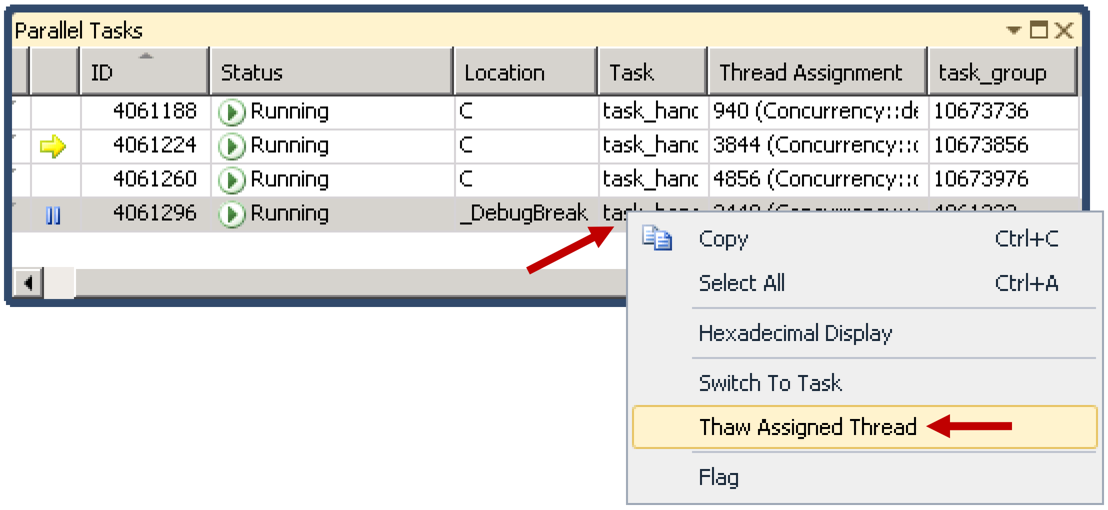
Figure 18

Locating the Break All button.

Note: That should pause the execution, and you should see the currently active task pause in one of the lines of the following segment of code. As you can see, the code does nothing except for waiting for task #1 to react and set the waitFor1 variable so that all the tasks can move on.

```
while (waitFor1)
{
    ;
}
```

12. In the Parallel Tasks window, right click on the paused task and select the **Thaw Assigned Thread** option.


Parallel Tasks

ID	Status	Location	Task	Thread Assignment	task_group
4061188	Running	C	task_hanc	940 (Concurrency::de	10673736
4061224	Running	C	task_hanc	3844 (Concurrency::c	10673856
4061260	Running	C	task_hanc	4856 (Concurrency::c	10673976
4061296	Running	_DebugBreak	ta		

Figure 19

Thawing the thread for task #1.

13. Click on the **Continue** button as shown in Figure 17.

Note: Now that task #1 has been thawed, the application should hit the second break point.

14. Click on the **Break** button for the breakpoint confirmation message as shown in Figure 11.

Part 3: Exploring the Parallel Stacks window

1. In the **Debug** menu, expand on the **Windows** submenu, and select the **Parallel Stacks** option.

> **Note:** You should see the Parallel Stacks window, which shows a graphic representation of the workflow of the application so far. It lets you see where each thread started, and it summarizes and groups the stacks for all of their calls.

2. Make sure to select the Tasks mode in the near the top-left corner of the window.

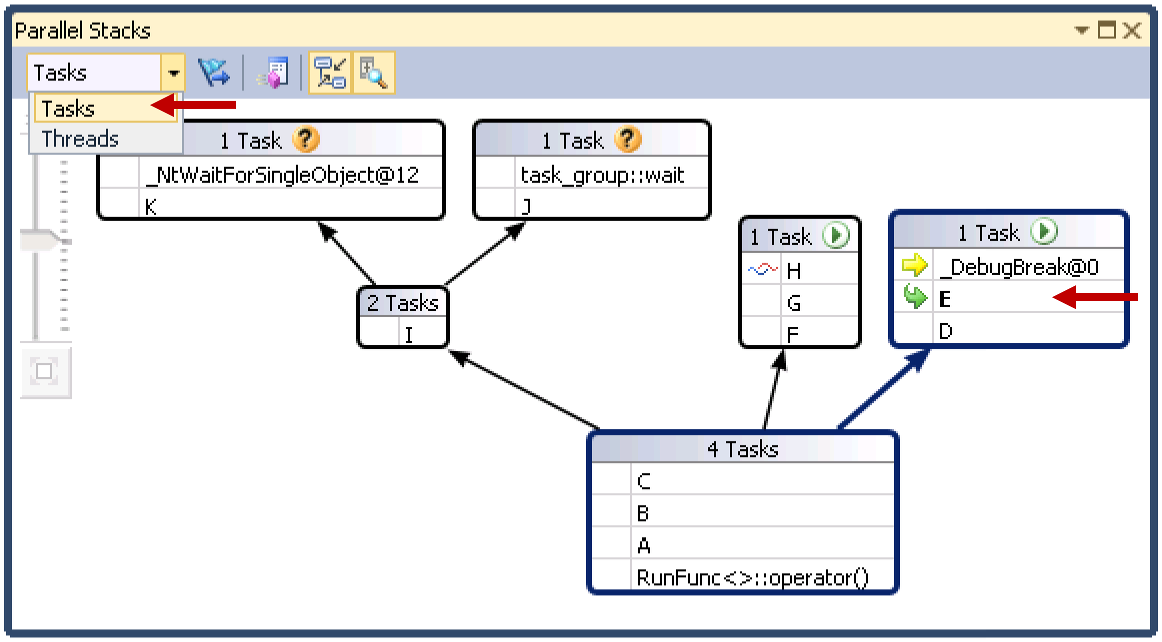
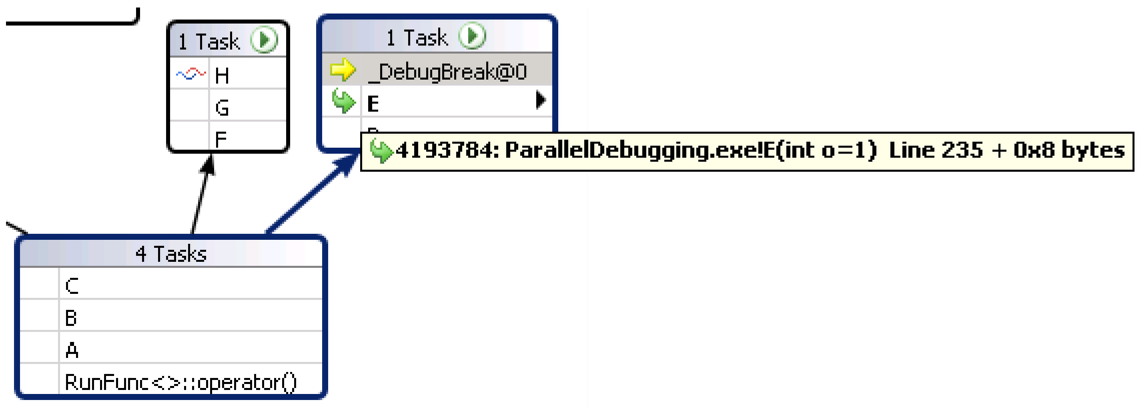


Figure 20

Parallel stacks of tasks up to the second breakpoint.

> **Note:** In this very simple diagram you can see how all four tasks started with calls to methods **A**, **B**, and **C**, as seen in the previous part of this exercise. From then one, two of those tasks went the **I** method, another one went to **F**, and another one went to **E**. Also notice the blue highlight around some of the boxes, and the arrows connecting them; this signals the current thread.

3. Locate the green arrow in the diagram; that is the method where the break point was triggered. Hover your mouse over that call.



4193784: ParallelDebugging.exe!E(int o=1) Line 235 + 0x8 bytes

Figure 21

Parallel stacks of tasks up to the second breakpoint.

Note: This tells you not only that the break point was triggered in the **E** method, but also, thanks to an easy view of the parameters with which it was called, you can also see that its **o** parameter is equal to 1, which means that this break point was also triggered by task #1.

4. Right click on the call that was made to the **J** method, point to the **Switch To Frame** submenu and select the first available frame.

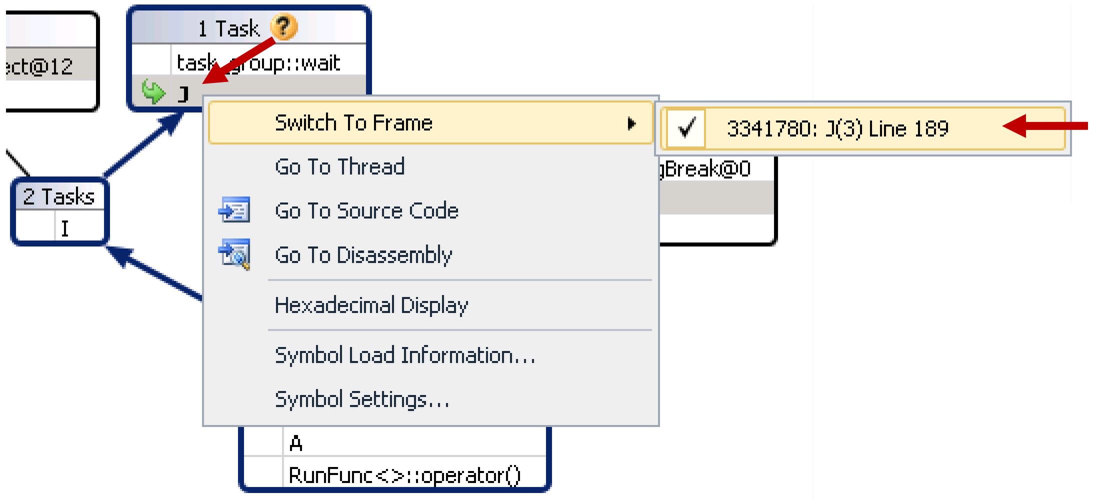
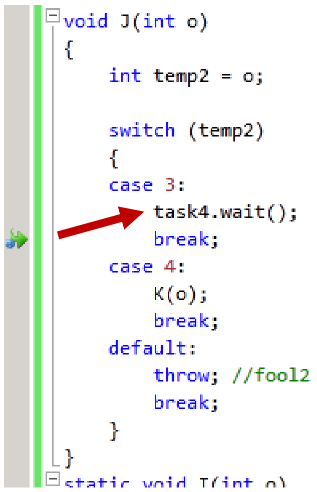


Figure 22

Parallel stacks of tasks up to the second breakpoint.

Note: The cursor will jump to the code for the selected frame. The instruction pointed by the green arrow would be next in execution, and the code demonstrates that task #3 is waiting for task #4 to complete.

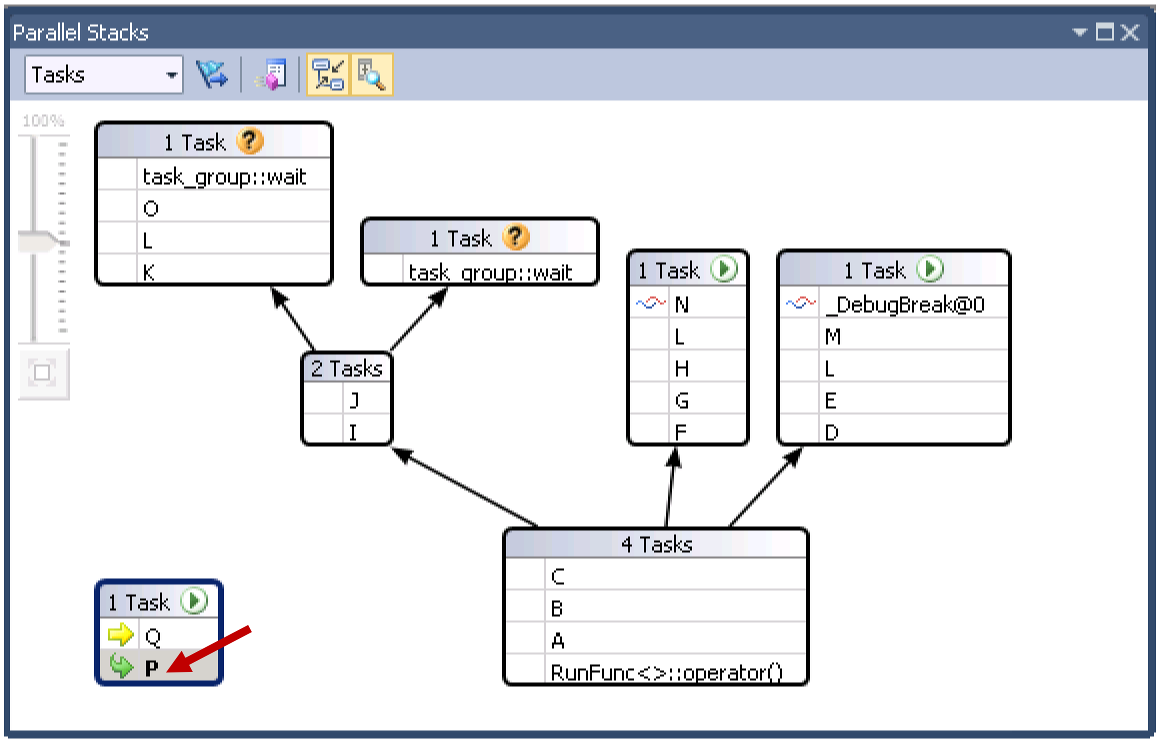


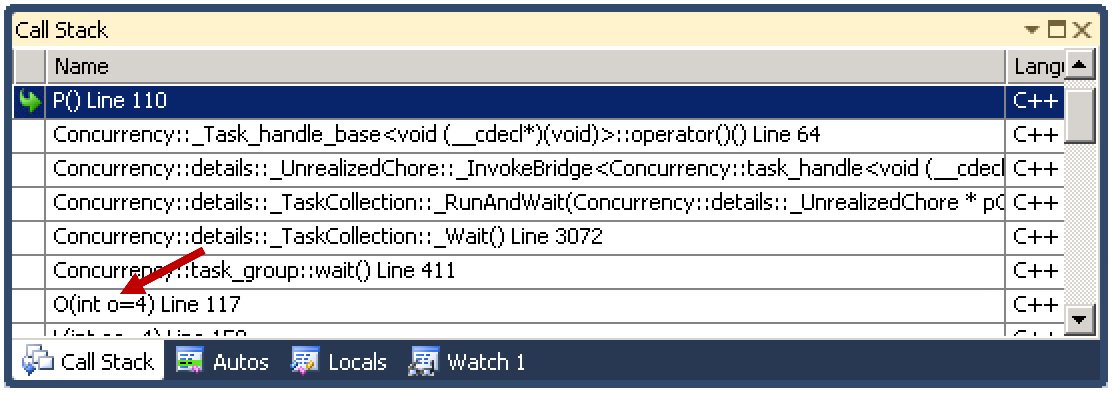
```
void J(int o)
{
    int temp2 = o;

    switch (temp2)
    {
    case 3:
        task4.wait();
        break;
    case 4:
        K(o);
        break;
    default:
        throw; //fool2
        break;
    }
}
static void I(int o)
```

Figure 22

Code showing recent execution of Task #3 while in second breakpoint.

5. Click on the **Continue** button as shown in Figure 17.
6. Click on the **Break** button for the breakpoint confirmation message as shown in Figure 11.
7. In the **Parallel Stacks** window, double click on the first call to the **P** method in the new task.





Name	Lang
P() Line 110	C++
Concurrency::_Task_handle_base<void (__cdecl*)(void)>::operator()() Line 64	C++
Concurrency::details::_UnrealizedChore::_InvokeBridge<Concurrency::task_handle<void (__cdecl	C++
Concurrency::details::_TaskCollection::_RunAndWait(Concurrency::details::_UnrealizedChore * p(	C++
Concurrency::details::_TaskCollection::_Wait() Line 3072	C++
Concurrency::task_group::wait() Line 411	C++
O(int o=4) Line 117	C++

Figure 23

*Exploring the call to **P** method in the call stack.*

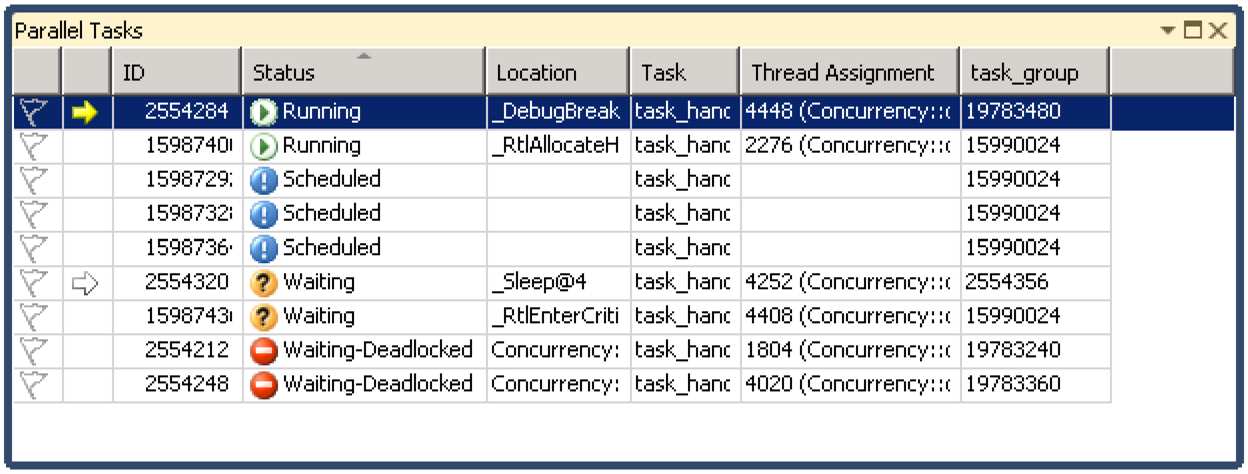
8. In the **Debug** menu, expand the **Windows** submenu, and select the **Call Stack** option.

Note: Previous steps in the call stack show that this new task was created by task #4 in the **O** method. As you can see, at this point there are only three tasks that are running, and two are waiting.

9. Click on the **Continue** button as shown in Figure 17.
10. Click on the **Break** button for the breakpoint confirmation message as shown in Figure 11.

Note: As you can see in the diagram, by the fourth breakpoint there have been a few changes for the tasks. But there is more than what you see in the diagram. When a task has been started in code, but it has not really run yet, this task's status is labeled as **Scheduled**, and these tasks are not displayed in the **Parallel Stacks** window.

11. In the **Debug** menu, expand the **Windows** submenu and select the **Parallel Tasks** option.



		ID	Status	Location	Task	Thread Assignment	task_group
		2554284	Running	_DebugBreak	task_hanc	4448 (Concurrency::(	19783480
		1598740	Running	_RtlAllocateH	task_hanc	2276 (Concurrency::(	15990024
		1598729;	Scheduled		task_hanc		15990024
		1598732	Scheduled		task_hanc		15990024
		1598736	Scheduled		task_hanc		15990024
		2554320	Waiting	_Sleep@4	task_hanc	4252 (Concurrency::(	2554356
		1598743	Waiting	_RtlEnterCriti	task_hanc	4408 (Concurrency::(	15990024
		2554212	Waiting-Deadlocked	Concurrency:	task_hanc	1804 (Concurrency::(	19783240
		2554248	Waiting-Deadlocked	Concurrency:	task_hanc	4020 (Concurrency::(	19783360

Figure 24

Multiple tasks with different statuses.

Note: There are two tasks that have a status of **Waiting-Deadlocked**. This means that these tasks are waiting on each other.

Lab Summary

In the past, it was feasible to improve the performance of a processor. The paradigm has changed, and improving the performance of a processor is not as feasible and cost effective as it has been for the past few years.

One of the ways that is more commonly used now to improve the performance of a computer is the addition of multiple cores. But this is no longer “transparent” to the developer; the code must be adapted to take advantage of these resources by logically dividing the workload among physical cores. The Parallel Programming Library removes most of the complexity of parallel programming away from your code, and lets you work primarily on the business logic of your application.

However, as simplified as it can be, most applications will hide scenarios that were not considered by developers; add to that the complexity of handling multiple true concurrent tasks, and the task of finding an error could become a real burden to the developer.

To complement the Parallel Programming Library, **Microsoft Visual Studio 2010 Beta 2** includes a set of tools that will help developers better inspect and manipulate threads, tasks, call stacks, and variables. Debugging an application with parallel tasks has been deeply simplified by offering a graphic representation of the workflow of any active tasks through the **Parallel Stacks** window, and the Parallel Tasks window tells the full story, by letting you filter task-threads, with more specific data, and with new built-in options to manage them.