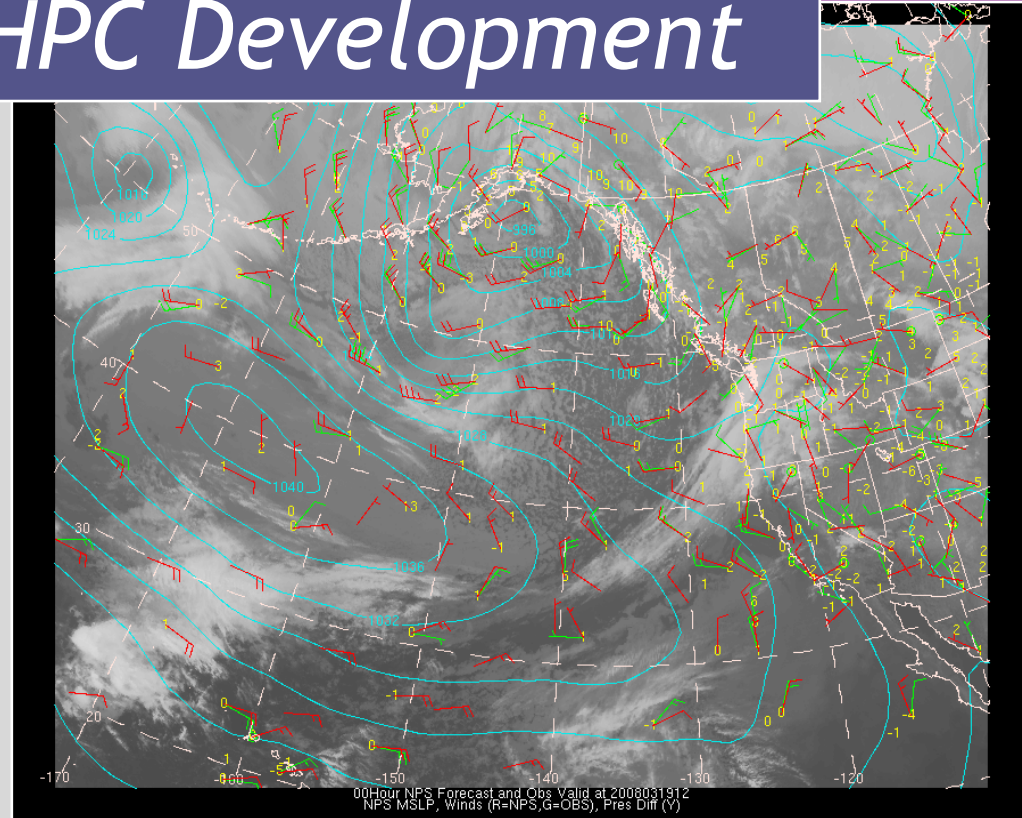


Sequential to Parallel HPC Development



© 2008 Microsoft Corporation

Developed by Pluralsight LLC

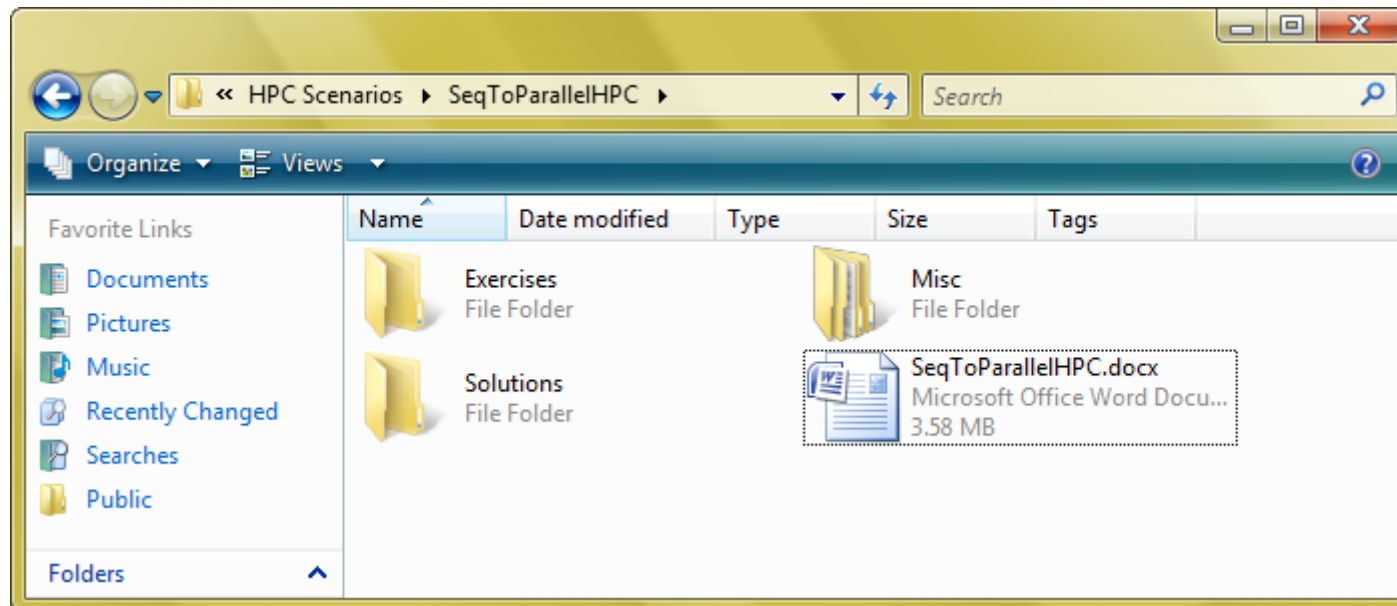
Table of Contents

Preface.....	4
1. Problem Domain	5
2. Task Parallelism and the Mandelbrot Set.....	6
3. A Sequential Version of the Mandelbrot Application	7
3.1 Architecture of Sequential Version (VC++)	7
3.2 Architecture of Sequential Version (C#).....	8
3.3 Concurrency and Parallelism in Traditional Applications	9
3.4 Lab Exercise!.....	11
4. Shared-Memory Parallel Programming — Multi-threading for Performance	13
4.1 Explicit Multi-threading using the .NET Thread Class	13
4.2 Multi-threading in VC++ with OpenMP	16
4.3 Configuring Visual Studio and VC++ to use OpenMP.....	18
4.4 Lab Exercise!.....	18
4.5 Multi-threading in C# with the TPL	20
4.6 Lab Exercise!.....	22
5. A Client-Server Version of the Mandelbrot Application.....	26
5.1 Client-Server Mandelbrot.....	26
5.2 Running the Sequential Client-Server Mandelbrot App.....	27
5.3 The Visual C++ and Visual C# versions of the Sequential Client-Server Mandelbrot App.....	28
5.4 Lab Exercise!.....	32
6. Working with Windows HPC Server 2008	33
6.1 Submitting a Job to the Cluster.....	33
6.2 Lab Exercise — Parallelization with Parametric Sweep!	38
7. Parallelizing the Client-Server Mandelbrot Application using Windows HPC Server 2008	42
7.1 Communicating with Windows HPC Server 2008 — The Job Scheduler API.....	43

7.2	Executing a Job on the Cluster.....	43
7.3	Configuring the Client-Server Mandelbrot Application for Cluster Execution	44
7.4	Implementing the Cluster-based Client-Server Mandelbrot Application	48
8.	Shared-memory Parallelization using Parametric Sweep, Thread Class, OpenMP, and the TPL	52
8.1	Lab Exercise — Shared-memory Parallelization on the Cluster!	52
9.	SOA-based Parallelization using WCF.....	56
9.1	Creating a WCF Service	56
9.2	Configuring the Broker Node.....	57
9.3	Installing a WCF Service under Windows HPC Server 2008	58
9.4	Calling the Service.....	59
9.5	Behind the Scenes	61
9.6	Amortizing Overhead and Executing in Parallel	61
9.7	Design Considerations	62
9.8	Lab Exercise!.....	63
10.	Conclusions.....	67
10.1	References.....	67
10.2	Resources	67
Appendix A: Summary of Cluster and Developer Setup for Windows HPC Server 2008.....		68
Appendix B: Troubleshooting Windows HPC Server 2008 Job Execution		70
Appendix C: Screen Snapshots		72
Feedback.....		74
More Information and Downloads		74

Preface

This document is a tutorial on Windows® HPC Server 2008. In particular, it presents an HPC scenario where the sequential C++/C# developer is redesigning their application to take advantage of parallel processing. We'll discuss a number of high-performance, parallel solutions using a range of technologies — OpenMP, PFX, WCF, and Windows HPC Server 2008. The complete tutorial includes lab exercises, program solutions, and miscellaneous support files. Installation of the tutorial yields a folder with the following structure:



This document presents a common HPC development scenario — the sequential developer looking to take advantage of parallel processing. Written for the C++ or C# developer, this tutorial walks you through the steps of designing and developing parallel applications for Windows® HPC Server 2008. This tutorial is designed to provide you with the skills and expertise necessary to deliver high-performance applications for Windows HPC Server 2008.

1. Problem Domain

Scientific computation is an obvious candidate for high-performance computing. The *Mandelbrot set*, shown graphically on the right, is an example of a simple mathematical definition leading to complex behavior. The Mandelbrot set is interesting for its connection with *Chaos Theory* and *Fractals*. The set contains an infinite number of elements, which is not surprising. However, the elements themselves appear random, or “chaotic”. This is elegantly conveyed by the border of the image. As you expand the border expecting the image to end, the more you realize the image goes on forever in new and intricate ways.

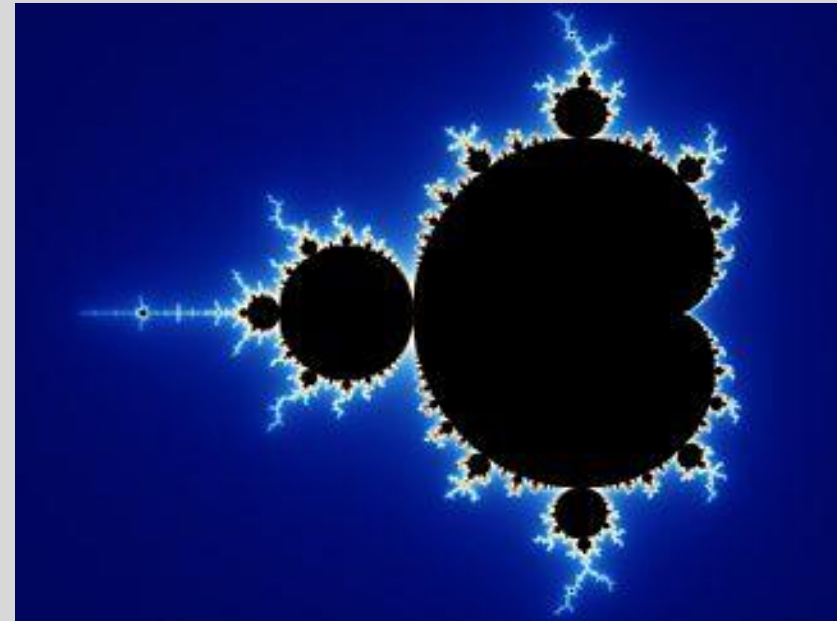
The algorithm for generating the image is straightforward. Let’s assume we want to produce a square image of size *pixels*. The values *x* and *y* denote the center of the generated image, and *size* represents the amount of magnification (larger values zoom out, smaller values zoom in). For example, the image on the top-right is generated by the following values:

```
pixels = 600;  
x      = -0.7;  
y      = 0.0;  
size   = 2.5;
```

Shifting (*x*, *y*) and reducing *size* yields images similar to the bottom-right. Given these values, the image is produced as follows:

```
for (yp = 0; yp < pixels; yp++)  
    for (xp = 0; xp < pixels; xp++)  
        image[yp, xp] = MandelbrotColor(yp, xp, y, x, size, pixels);
```

The *MandelbrotColor* function generates a color reflecting the time taken to determine whether the pixel is an element of the Mandelbrot set — the darker the color, the longer the computation. Black pixels represent algorithm termination before a result is known; these pixels are likely not to be members of the Mandelbrot set. Here’s a definition of *MandelbrotColor*, based on our own coloring scheme and the mathematical definition of the Mandelbrot set:



```

int MandelbrotColor(yp, xp, y, x, size, pixels)
{
    // Compute pixel position:
    ypos = y + size * (yp - pixels/2) / ((double) pixels);
    xpos = x + size * (xp - pixels/2) / ((double) pixels);

    // Now setup for color computation:
    y = ypos;
    x = xpos;

    y2 = y*y;
    x2 = x*x;

    color = 1;

    // Repeat until we know pixel is not in set, or until a max # of iterations has been
    // reached --- in which case pixel is probably in set (and colored MAXCOLOR).
    while ((y2 + x2) < 4 && color < MAXCOLOR)
    {
        y = 2*x*y + ypos;
        x = x2-y2 + xpos;

        y2 = y*y;
        x2 = x*x;

        color++;
    }

    return color;
}

```

For more information, Wikipedia contains a detailed discussion of the Mandelbrot set, including algorithms and issues for the developer¹

2. Task Parallelism and the Mandelbrot Set

Visualizing the Mandelbrot set is a classic example of *task parallelism* — where the *computation* is viewed as a set of tasks operating on independent *data streams*. In this case, generating a pixel of the Mandelbrot image is the computation, and the pixel position is the data stream. Each task is thus a call to MandelbrotColor(yp, xp, y, x, size, pixels).

¹ http://en.wikipedia.org/wiki/Mandelbrot_set.

In fact, visualizing the Mandelbrot set is considered an *embarrassingly parallel* computation, since the tasks are completely independent of one another — generation of the pixel at (y1, x1) is in no way dependent upon the generation of any other pixel (y2, x2). This greatly simplifies parallelization of the algorithm. Given P pixels and N execution units, we can assign the P tasks to the N execution units in whatever manner we want, and expect a factor of N increase in performance. For example, if generating 360,000 pixels (a 600x600 image) takes 60 seconds on a single-core machine, it should take 1/4 the time on a quad-core machine — i.e. 15 seconds. And on a cluster with 60 cores, it should take just 1 second.

3. A Sequential Version of the Mandelbrot Application

An important first step in developing a parallel version of an application is to create a sequential version. A sequential version allows us to gain a better understanding of the problem, provides a vehicle for correctness testing against the parallel versions, and forms the basis for performance measurements. Performance is often measured in terms of *speedup*, i.e. how much faster the parallel version executed in comparison to the sequential version. More precisely:

$$speedup = \frac{Sequential_{time}}{Parallel_{time}}$$

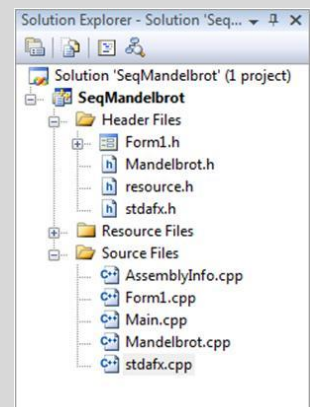
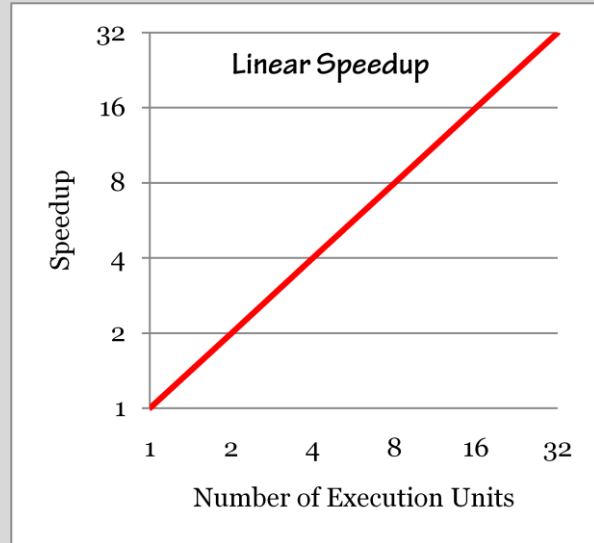
For example, if the sequential version runs in 60 seconds and the parallel version runs in 15 seconds, then the speedup is 4. If the parallel version was run on 4 execution units, this is a very good result — the sequential workload was perfectly parallelized across the execution units with no measurable overhead. If, however, the parallel version was run on 16 execution units, then the quality of the result depends on your expectations. A time of 15 seconds might be considered good if the application was difficult to parallelize. If, on the other hand, the application was considered well-written and highly-parallel, then it's a bad result — the total execution time should have been around 4 seconds.

Why 4 seconds? Generally, the goal of parallelizing an application is *linear* speedup: given N execution units, the parallel version should run N times faster than the sequential one (see graph upper-right). Linear speedup implies the parallel application is fully utilizing all N execution units in meaningful computation. Given a sequential app running in 60 seconds, the parallel version should run in 30 seconds on 2 execution units, 15 seconds on 4 units, 7.5 seconds on 8 units, and 3.75 seconds on 16 units. Linear speedup is one of the holy grails of HPC.

3.1 Architecture of Sequential Version (VC++)

Let's take a look at the architecture of the sequential VC++ version of the Mandelbrot application. [For the C# version, skip to the [next section](#).] Open the solution (.sln file) in VS 2008, found in [Solutions\Sequential\SeqMandelbrot](#). This is a .NET application written in Visual C++®, consisting of 4 include files and 5 source files:

Form1.h	definition of main WinForm class for user interface
Mandelbrot.h	definition of Mandelbrot class for computation
resource.h	<empty>



stdafx.h application-wide, pre-compiled header file

AssemblyInfo.cpp managed code information relevant to .NET
Form1.cpp implementation of main WinForm class
Main.cpp main program
Mandelbrot.cpp implementation of Mandelbrot class
stdafx.cpp support for pre-compiled header file

Let's ignore the implementation details, and simply gain a broad overview of the application. First, open "Form1.h" and you'll see the design of the UI (shown on the right). Next, open "Main.cpp" and notice it creates an instance of Form1 and then "runs" the form — this launches the UI when the app starts. Finally, open "Mandelbrot.h", and you'll see the design of the Mandelbrot class where the computation is performed. That's enough for now.

Let's run the program to get a feeling for how it behaves. First, select the platform for your local workstation (Win32 or x64) via the drop-down in the Visual Studio® standard toolbar:



Now run via F5, which starts the app and launches the UI. Click the "Go" button to start generation of the Mandelbrot image. Notice that the initial rows generate quickly, since it is easily determined the pixels are in the Mandelbrot set (lighter colors mean fewer iterations of the MandelbrotColor function). As set determination becomes more costly (requiring more iterations and yielding darker colors), the application slows down significantly.

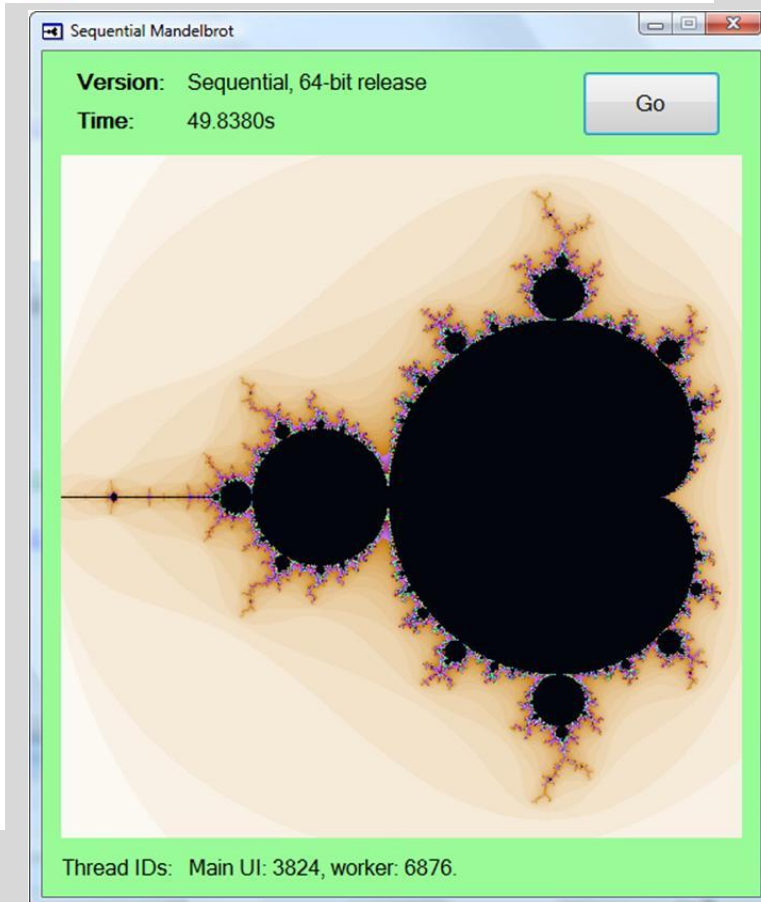
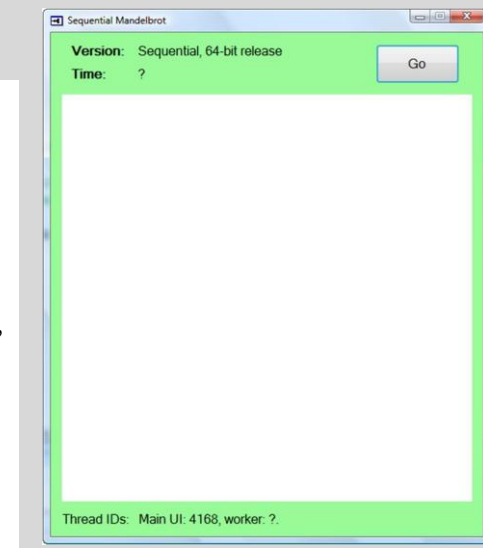
Repeat, and notice how the "Go" button becomes a "Cancel" button. Cancel the computation, and image generation stops. This simple feature is harder to implement than one might think, and significantly impacts the design of the application. The good news is that it introduces concepts of parallelism, even in a "sequential" version of the application. We'll [continue this discussion](#) in a moment after we present the C# version of the app.

3.2 Architecture of Sequential Version (C#)

The C# version of the Mandelbrot application shares the same architecture as the Visual C++ version. Start by opening the sequential C# solution (.sln file) in VS 2008, found in [Solutions\Sequential.NET\SeqDotNetMandelbrot](#). This is a .NET application written in C#, consisting of 3 source code files:

Form1.cs WinForm class for user interface
Mandelbrot.cs Mandelbrot class for computation
Program.cs main program

Let's ignore the implementation details, and simply gain a broad overview of the application. First, open "Form1.cs" and you'll see the design of the UI (upper-right). Next, open "Program.cs" and notice it creates an instance of Form1 and then "runs" the form — this



launches the UI when the app starts. Finally, open “Mandelbrot.cs”, and you’ll see the design of the Mandelbrot class where the computation is performed. That’s enough for now.

Let’s run the program to get a feeling for how it behaves. Press F5, which starts the app and launches the UI. Click the “Go” button to start generation of the Mandelbrot image. Notice that the initial rows generate quickly, since it is easily determined the pixels are in the Mandelbrot set (lighter colors mean fewer iterations of the MandelbrotColor function). As set determination becomes more costly (requiring more iterations and yielding darker colors), the application slows down significantly. An application snapshot is shown on the previous page, bottom-right.

Repeat, and notice how the “Go” button becomes a “Cancel” button. Cancel the computation, and image generation stops. This simple feature is harder to implement than one might think, and significantly impacts the design of the application. The good news is that it introduces concepts of parallelism, even in a “sequential” version of the application.

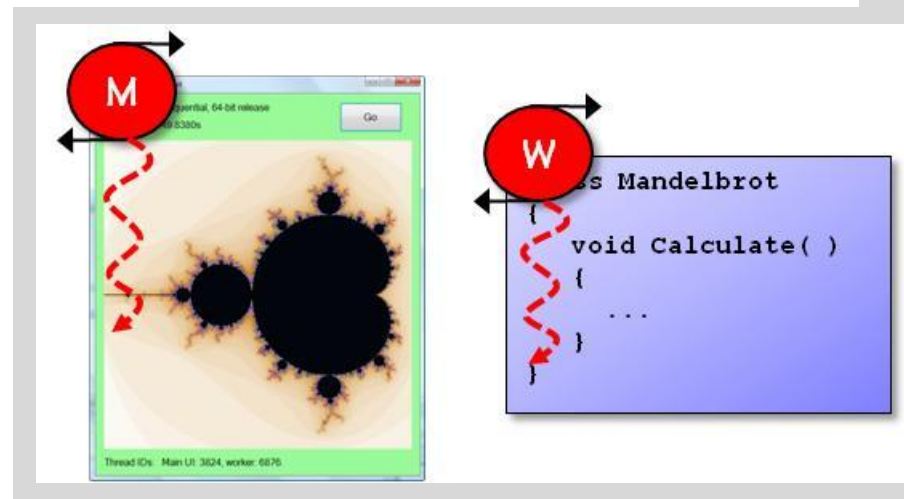
3.3 Concurrency and Parallelism in Traditional Applications

In both versions of the application (VC++ and C#), *multi-threading* is used to maintain a responsive user interface during the computation. In particular, a *worker* thread is used to generate the Mandelbrot image, while the *main* thread processes events and draws the image. These are depicted as red “cyclones” (M and W) in the image to the right, conveying that two execution agents are in fact running through the program at the same time. On today’s multi-core hardware, these agents / threads may execute simultaneously.

While various workarounds exist, multi-threading is often the best approach for maintaining responsive user interfaces, especially when offering “cancel” functionality. On the other hand, multi-threading increases the complexity of an application, since concurrent execution introduces potential problem areas in the form of critical sections, race conditions, and deadlock. In the case of WinForm (i.e. GUI) applications, .NET adds an additional constraint: the only thread that may touch the UI is the main thread M. Otherwise a run-time exception occurs.

In .NET, every application starts with a single thread, known as the *main* thread. Additional *worker* threads must be created explicitly, or implicitly through the use of language features (such as *asynchronous delegates*) or classes (such as *BackgroundWorker*). When creating worker threads for high-performance computing, the best approach is either explicit creation, or using higher-level techniques such as OpenMP² or the TPL³. In the case of our existing Mandelbrot application, this is a traditional app focused more on usability than high performance. For this reason, the design uses the *BackgroundWorker* class, a class provided by .NET expressly for background computation interacting with a GUI.

If you haven’t already, open either version of the sequential Mandelbrot app: VC++ ([Solutions\Sequential\SeqMandelbrot](#)) or C# ([Solutions\Sequential.NET\SeqDotNetMandelbrot](#)). View the code behind Form1, in particular the code behind the “Go” button’s Click event (*Form1::GoButton_Click* or *_goButton_Click*). When it comes time to perform the computation, a new Mandelbrot object is created, followed by a new *BackgroundWorker* object to do the work on a separate thread:



² Open Multi-Processing: <http://www.openmp.org/>.

³ Task Parallel Library: <http://msdn.microsoft.com/en-us/concurrency/default.aspx>.

```
//
// C# version:
//
_mandelbrot = new Mandelbrot(x, y, size, pixels); // create object that contains computation to perform:

.
.
.

_worker = new BackgroundWorker();
_worker.DoWork += new DoWorkEventHandler( _mandelbrot.Calculate ); // hook computation into worker thread:

.
.
.

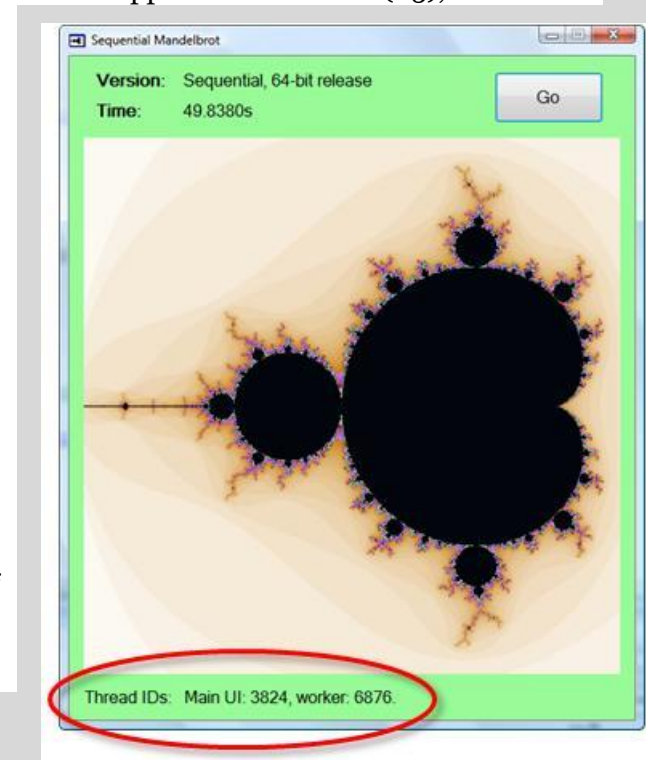
_worker.RunWorkerAsync(null); // tell worker to start computation!
```

The BackgroundWorker class adheres to an event-based version of the Asynchronous design pattern⁴, where the work — *_mandelbrot.Calculate* — is written as an event handler and hooked into the worker’s *DoWork* event. Nothing happens until the worker is told to “RunWorkerAsync”, at which point the handler(s) behind the DoWork event are executed concurrently by the worker thread. Run the app in Visual Studio (F5), and note that the Windows thread ids are displayed along the bottom of the GUI (see snapshot below-right).

The primary advantage of the BackgroundWorker class is that it encapsulates the creation of the worker thread, handling the details for you. It also supports canceling of the thread, and catches any exceptions that might occur during thread execution (exception details are available as part of the *RunWorkerCompleted* event). The other significant advantage to the BackgroundWorker class is its event-based design, where progress and completion are also raised as events. This design allows the GUI code to handle progress updates and completion notification, updating the UI appropriately. More subtly, an event-based design allows .NET to internally context-switch from the worker thread (raising the event) to the main thread (handling the event), thereby adhering to the rule that only the main thread may touch the UI. This is setup by having the form-based code handle the progress and completion events. Revisit the code for the “Go” button’s Click event (*Form1::GoButton_Click* or *_goButton_Click*):

```
_worker = new BackgroundWorker();
_worker.DoWork += new DoWorkEventHandler( _mandelbrot.Calculate );
_worker.ProgressChanged += new ProgressChangedEventHandler( this.OnProgress );
_worker.RunWorkerCompleted += new RunWorkerCompletedEventHandler( this.OnComplete );
```

Notice the ProgressChanged and RunWorkerCompleted events are handled by code within the form itself, i.e. the methods *OnProgress* and *OnComplete*, respectively.



⁴ <http://msdn.microsoft.com/en-us/library/wewwczdwx.aspx>.

Once started, the worker thread executes the *Calculate* method in the *Mandelbrot* class. This method computes the Mandelbrot image as discussed in [Section 1](#). As each row of the image is generated, it is reported as progress to the main form:

```
//  
// C# version:  
//  
void Calculate(Object sender, DoWorkEventArgs e)  
{  
    _worker = (BackgroundWorker) sender;  
  
    for (int yp = 0; yp < _pixels; yp++)  
    {  
        .  
        . // generate the row YP of the image:  
        .  
  
        Object[] args = new Object[2];  
        args[0] = values; // pixel values:  
        args[1] = AppDomain.GetCurrentThreadId();  
  
        _worker.ReportProgress(yp, args);  
    }  
}
```

In response, the main form draws the given image row on the screen:

```
void OnProgress(Object sender, ProgressChangedEventArgs e)  
{  
    int currLine = e.ProgressPercentage;  
  
    Object[] args = (Object[]) e.UserState;  
    int[] values = (int[]) args[0];  
    int workerID = (int) args[1];  
  
    for (int x = 0; x < values.Length; x++)  
        _image.SetPixel(x, currLine, color(values[x]));  
}
```

For more details, see the code behind Form1, as well as the Mandelbrot class.

3.4 Lab Exercise!

A quick lab exercise to time the sequential version of the app, and to see for yourself the dangers of multi-threading...

1. If you haven't already, open either the VC++ or the C# version of the sequential Mandelbrot application: for VC++ open



[Solutions\Sequential\SeqMandelbrot](#), for C# open [Solutions\Sequential.NET\SeqDotNetMandelbrot](#). Run (F5) and make sure all is well.

2. First, let's record the average execution time of 3 sequential runs on your local workstation. We need this result to accurately compute [speedup](#) values for the upcoming parallel versions. Before you run, make sure you configure for the *Release* version, and the appropriate platform (i.e. Win32 or x64 if available):



Now run without debugging (Ctrl+F5). When you are done, record the average time here:

Sequential_{time} on local workstation for Mandelbrot run (-0.70, 0, 2.5, 600): _____

3. Next, let's experiment with the Mandelbrot image that is generated. Find the "Go" button's Click event handler (*Form1::GoButton_Click* or *_goButton_Click*), and change the parameters for x, y or size — changing x and y repositions the center of the image, and changing size zooms in or out. For example, set x and y to 0, and size to 1.0 (zoom in). Run, and see the effect...

4. Finally, let's look at one of the dangers of multi-threading. In the code for the Mandelbrot class, find the Calculate method. Recall this method is executed by a worker thread, and is responsible for computing the Mandelbrot image, one row at a time. As each row is generated, it is reported as progress to the main form (and thread) for display. Here's the code for generating one row of the image and reporting it:

```
//  
// C# version:  
//  
int[] values = new int[_pixels];  
  
for (int xp = 0; xp < _pixels; ++xp)  
    values[xp] = MandelbrotColor(yp, xp, _y, _x, _size, _pixels);  
  
Object[] args = new Object[2];  
args[0] = values;  
args[1] = AppDomain.GetCurrentThreadId();  
  
_worker.ReportProgress(yp, args);
```

Notice that for each row, we allocate a new integer array to hold the pixel *values*, as well as a new object array to hold the *args* for reporting progress. Why not allocate these arrays just once at the beginning of the method, and reuse them? After all, once we report progress, aren't we done with this row and so it's safe to reuse these arrays for the next row? Try it and find out. What happens? Depending on the type of hardware you have (speed of CPU, number of cores, etc.), you may find that the image is no longer generated / displayed correctly... What's going on?

This is a classic example of a race condition: the worker thread races ahead to generate the next row, while the main thread processes the

current row. Since the arrays are now shared between the threads, if the main thread doesn't finish with the current row before the worker thread starts generating the next row, the data in the array changes while the main thread is still processing it — leading to incorrect results. The most efficient solution (in terms of performance) is to eliminate the data sharing (and thus the race condition) by allocating new arrays each time. If efficiency is being measured in terms of memory usage, then an entirely different solution would be needed, one involving *locks* or *semaphores*⁵.

4. Shared-Memory Parallel Programming — Multi-threading for Performance

To improve performance of the Mandelbrot application, the obvious approach is to take advantage of the embarrassingly-parallel nature of the problem. [As discussed earlier](#), each pixel of the Mandelbrot image can be computed independently of all others, providing a large amount of parallelism. The easiest way to exploit this parallelism is to augment the existing shared-memory design — where the worker and main thread communicate via arrays — with additional threads to perform the computation (see diagram to right). Given N additional threads on a machine with N execution units, we can expect an N-fold increase in performance. In other words, [linear speedup](#).

4.1 Explicit Multi-threading using the .NET Thread Class

Given a .NET application, be it VC++ or C#, the first option is to explicitly create the threads yourself using .NET's *Thread* class⁶. The first step is to formulate your computation as a void method taking a single parameter of type *Object*. In C#:

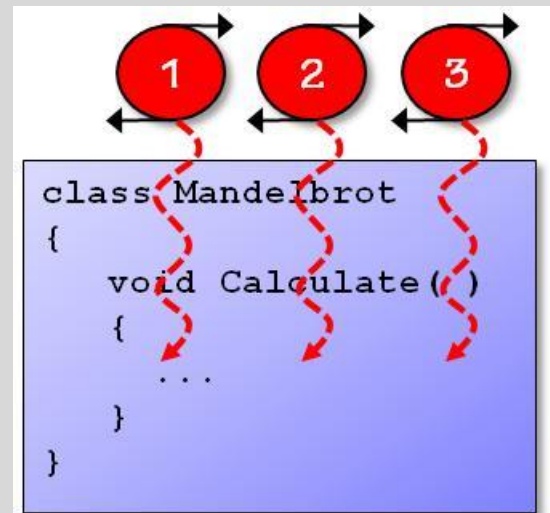
```
void MyComputation(Object arg)
{
    .
    .
    .
}
```

This design allows you to pass 0 or more arguments to the method (to pass multiple values, pass a single array containing the values). Next, we create one or more threads to execute this method. For each thread, instantiate a new *Thread* object, passing a reference to the *MyComputation* method:

```
Object arg = ...;
Thread t    = new Thread( new ParameterizedThreadStart(MyComputation) );
```

When you are ready to start execution, call the thread object's *Start* method:

```
t.Start(arg);    // start thread running!
```



⁵ See most any Operating Systems textbook, and read about solutions to the *Producer-Consumer* problem.

⁶ If you are creating an unmanaged Visual C++ application, the equivalent approach is to use the Windows API *CreateThread* function. For more details, see the online MSDN article “*Multithreading with C and Win32*”, [http://msdn.microsoft.com/en-us/library/y6h8hye8\(VS.80\).aspx](http://msdn.microsoft.com/en-us/library/y6h8hye8(VS.80).aspx).

At this point a new worker thread is executing `MyComputation`. If it's important for the creating thread to wait for the worker thread to finish, it can do so by calling the thread object's `Join` method:

```
t.Join(); // wait here until thread finishes...
```

This style of parallelism is known as *fork-join* parallelism. When parallel execution is desired, one or more worker threads are created (“forked”) off the current thread. Eventually, when parallel execution completes, the worker threads merge back (“join”) with the creating thread, and the application continues as before.

Let's look at the use of explicit multi-threading to parallelize the Mandelbrot application. Start by opening a parallel version of the Mandelbrot app in Visual Studio: for VC++ open the .sln file in [Solutions\Parallel\ParMandelbrot](#), and for C# open the .sln file in [Solutions\Parallel.NET\ParDotNetMandelbrot](#). Select the Release version, build, and run without debugging (Ctrl+F5). If your local workstation contains multiple cores/CPU's (say N), the application should now run N times faster. Record the time:

Thread Parallel_{time} on local workstation for Mandelbrot run (-0.70, 0, 2.5, 600): _____, number of cores = _____

For example, on my dual-core laptop, the sequential version ran in 53 seconds, and the parallel version in under 30 seconds. Another way to visualize is to open the *Task Manager* (Ctrl+Alt+Del), switch to the Performance tab, and watch CPU utilization.

In the sequential version, the main thread handles the UI, and the worker thread handles the computation of the Mandelbrot image. In the parallel version, the worker thread now creates N additional threads, one per core/CPU on the local workstation. Parallelization is thus encapsulated within the *Mandelbrot* class. Open “Mandelbrot.cpp” / “Mandelbrot.cs”, and locate the *Calculate* method. The method now creates N threads, assigning each thread a distinct block of rows to generate for the image:

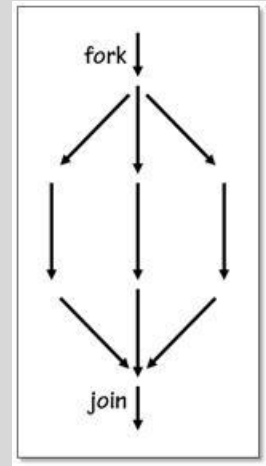
```
int numCores = System.Environment.ProcessorCount; // number N of local cores / CPUs:
int chunkSize = _pixels / numCores;               // number of rows for each thread to compute:
int leftOver = _pixels % numCores;                 // in case things don't divide equally:

Thread[] threads = new Thread[numCores];
Object[] args;

for (int i = 0; i < numCores; i++) // for each core, create & start one thread:
{
    int start = i * chunkSize;
    int end = start + chunkSize;
    int id = i + 1;

    if (leftOver > 0 && i == numCores - 1) // give any extra rows to the last thread:
        end += leftOver;

    args = new Object[3];
    args[0] = start; // startRowInclusive:
    args[1] = end;   // endRowExclusive:
    args[2] = id;    // some sort of thread id, in range 1..N:
```




```

        threads[i] = new Thread( new ParameterizedThreadStart( this.DoCalculate ) );
        threads[i].Start(args);
    }

```

For example, if your workstation contains 2 cores, thread 1 generates rows (0, 299) while thread 2 generates rows (300, 599). Once the threads are started, the worker thread simply waits for the computation threads to finish:

```

//
// now we wait for the threads to finish:
//
for (int i = 0; i < numCores; i++)
    threads[i].Join();

```

Each of the computation threads executes the method *DoCalculate*, parameterized by the row bounds (startRowInclusive, endRowExclusive):

```

void DoCalculate(Object arg)
{
    Object[] args = (Object[])arg;

    int startRowInclusive = System.Convert.ToInt32(args[0]);
    int endRowExclusive   = System.Convert.ToInt32(args[1]);
    int threadID         = System.Convert.ToInt32(args[2]);

    for (int yp = startRowInclusive; yp < endRowExclusive; yp++)
    {
        ...
    }
}

```

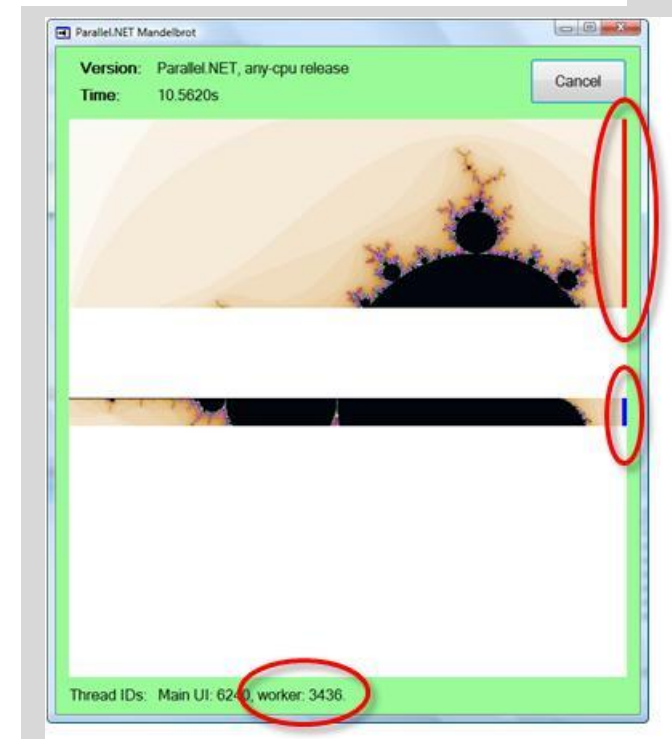
The *threadID* is used to color-code the rows of the image, allowing you to visualize which threads generated which rows. Assuming you have $N > 1$ cores/CPU's, run the app and look more closely at the right-most edge of the image: rows by thread 1 are denoted in red, rows by thread 2 in blue, and so on. Note also the worker thread id shown at the bottom of the form: this will change (flicker?) based on the thread reporting the current row. If your local machine does not contain multiple execution units, have no fear — you can still multi-thread by explicitly setting the number of cores in the Calculate method:

```

//int numCores = System.Environment.ProcessorCount;
int numCores = 7;

```

This is also a great way to test our parallelization, e.g. correct handling of unequal workloads. It also demonstrates the difficulty of the general parallelization problem. If you haven't already, explicitly set numCores to 4, and run. What happens? Threads 1 and 4 finish quickly, since the vast majority of their pixels fall quickly into the Mandelbrot set, while threads 2 and 3 grind out the remainder of the image. Clearly, our *static* division of the workload is less than optimal in this case. The solution



is a more dynamic scheduling of rows to threads, at great complication to the code.

4.2 Multi-threading in VC++ with OpenMP

OpenMP is high-level approach for shared-memory parallel programming. OpenMP, short for *Open Multi-Processing*⁷, is an open standard for platform-neutral parallel programming. Support on the Windows platform first appeared in Visual Studio® 2005 for Visual C++, and continues to enjoy full support in Visual Studio® 2008. The underlying paradigm of OpenMP is a fork-join style of parallelism, using threads.

OpenMP provides excellent support for data and task parallelism, offering higher-level abstractions based on code directives that guide parallelism. The classic example is parallelization of a *for* loop using OpenMP's *parallel for* directive:

```
#pragma omp parallel for
for (int i = 0; i < N; i++)
    PerformSomeComputation(i);
```

In response to the directive, the compiler will automatically generate code to divide up the iteration space and execute the loop in parallel. By default, *static* scheduling is typically used, in which case the iteration space is divided evenly across the threads. However, one can easily specify the use of alternatives such as *dynamic* scheduling, where iterations are assigned to threads one-by-one:

```
#pragma omp parallel for schedule(dynamic)
for (int i = 0; i < N; i++)
    PerformSomeComputation(i);
```

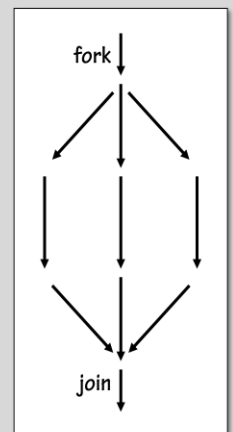
This allows for better load-balancing, at the cost of more scheduling overhead. Note that *parallel for* is really just a special case of OpenMP's more general concept of a *parallel region*, in which threads are forked off to execute code in parallel. We can use parallel regions to assert more control over parallelization. For example, here we explicitly divide the iteration space ourselves:

```
#pragma omp parallel
{
    int setSize = N / omp_get_num_threads(); // divide up the iteration space
    int extra   = N % omp_get_num_threads(); // if not evenly-divisible, note extra iterations

    int thread = omp_get_thread_num(); // which thread am I? 0, 1, 2, ...
    int first  = thread * setSize;      // compute which iteration set I'm processing
    int lastp1 = first + setSize;       // compute end of my iteration set (exclusive)

    if (thread == omp_get_num_threads() - 1) // tack extra iterations to workload of last thread:
        lastp1 += extra;

    for (int i = first; i < lastp1; i++) // process my iteration space:
        PerformSomeComputation(i);
}
```



⁷ <http://www.openmp.org/>. For Microsoft-specific details, lookup “OpenMP” in the MSDN library (F1).

In the case of a parallel region, imagine T threads (one per core/CPU) executing this code in parallel. This implies the above code is identical to that of the original for loop with a parallel for directive, and hence of no real value. It does however demonstrate the use of OpenMP's API for obtaining run-time information, and for modifying some aspects of the run-time environment. Here are the most commonly used functions:

<code>omp_get_num_procs()</code>	<i>returns number of processors currently available at time of call</i>
<code>omp_get_max_threads()</code>	<i>returns maximum number of threads available for execution of a parallel region</i>
<code>omp_get_num_threads()</code>	<i>returns number of threads currently executing in the parallel region</i>
<code>omp_get_thread_num()</code>	<i>returns thread number of calling thread (0 .. <code>omp_get_num_threads()-1</code>)</i>
<code>omp_in_parallel()</code>	<i>returns a nonzero value if called from within a parallel region</i>
<code>omp_set_num_threads(N)</code>	<i>sets the number N of threads to use in subsequent parallel regions</i>

Finally, note that shared variables, and race conditions that may result from parallel access to shared variables, are the responsibility of the **programmer**, not OpenMP. For example, let's revisit our original for loop:

```
#pragma omp parallel for
for (int i = 0; i < N; i++)
    PerformSomeComputation(i);
```

Suppose that *PerformSomeComputation* adds the parameter i to a global variable (this could just as easily be a global data structure):

```
int global = 0;

void PerformSomeComputation(int i)
{
    global += i;
}
```

The sequential version of this program computes $N*(N-1)/2$. The parallel version has a race condition, and computes a value somewhere between 0 and $N*(N-1)/2$, inclusive. The solution is to control access to the shared resource, e.g. with an OpenMP *critical section* directive:

```
void PerformSomeComputation(int i)
{
    #pragma omp critical
    {
        global += i;
    }
}
```

Race conditions are the single, largest problem in parallel applications. Beware!

4.3 Configuring Visual Studio and VC++ to use OpenMP

OpenMP is easily enabled for Visual C++ in both Visual Studio 2005 and 2008: right-click on your project in the Solution Explorer, select Properties, Configuration Properties, C/C++, and Language. Then, for the property “OpenMP Support”, click its field to enable the drop-down, and select Yes.

Second, in any source code file that uses OpenMP, you need to #include the file <omp.h>:

```
//  
// somefile.cpp  
//  
#include <omp.h>
```

That’s it, you are now ready to use OpenMP.

4.4 Lab Exercise!

Okay, let’s explore parallelization of the VC++ version of Mandelbrot application using OpenMP. What you’re going to find is that in the context of C++ and loop-based parallelism, OpenMP is a very effective approach for shared-memory parallelism. Note that a solution to the lab exercise is provided in [Solutions\OpenMP\OpenMPMandelbrot](#).

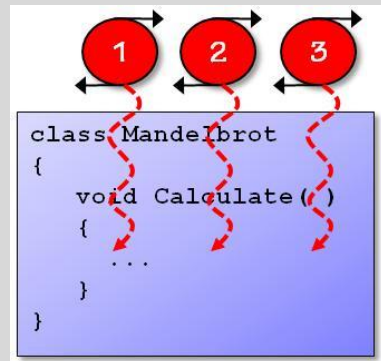
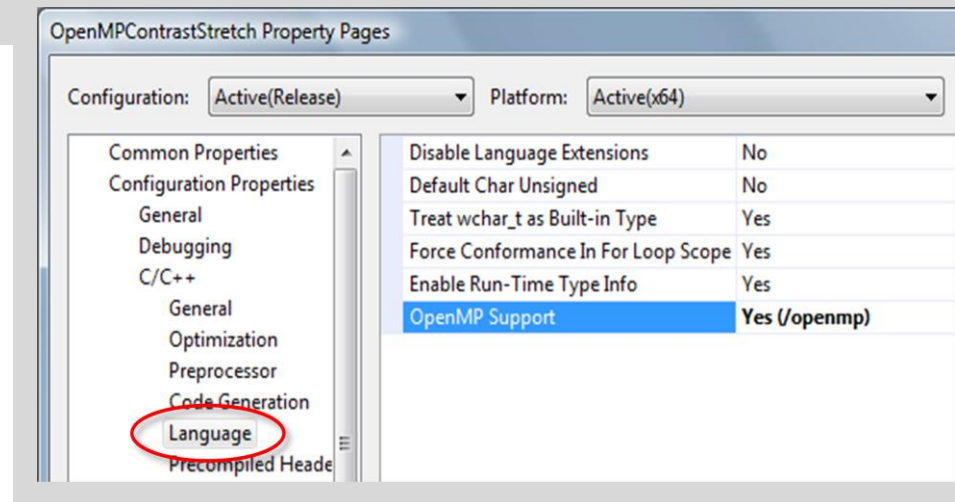
1. Start by opening the sequential version of the application provided in [Exercises\01 OpenMP\OpenMPMandelbrot](#). Switch to the target platform appropriate for your workstation (Win32 or x64):



Enable OpenMP as discussed in the [previous section](#). The idea is to parallelize the Mandelbrot class’s *Calculate* method, using multiple threads as we did [earlier](#) to generate the image rows in parallel. Only this time we’ll let OpenMP do the work of dividing up the iteration space and creating the threads.

2. Since the parallelization will be encapsulated within the Mandelbrot class, open “Mandelbrot.cpp” and #include <omp.h>. OpenMP directives and API functions are now available for use in this source code file. In the Calculate method, parallelize the row processing by adding a *parallel for* directive immediately above the outer for loop:

```
#pragma omp parallel for  
for (int yp=0; yp<_pixels; yp++)  
{  
    .  
    .  
    .  
}
```



That's it, OpenMP will take care of the rest! However, build the application, and you'll encounter a compilation error: one of the restrictions imposed by OpenMP is that you cannot break or return out of a parallel loop. Thus the following code, which stop processing when the user cancels, is illegal inside the body of the loop:

```
if (_worker->CancellationPending)
{
    e->Cancel = true;
    return;           // <-- this is illegal inside a parallel for loop:
}
```

In this case, the simplest solution is to replace the *return* statement with a *continue* statement — causing each thread to skip through the remaining iterations quickly, thereby simulating a break / return. Change the return to a continue, build the app, and run (Ctrl+F5). If your workstation has multiple cores/CPU's, you should see an immediate performance improvement. You should also see a static division of the iteration space: thread 1 generates the first chunk, thread 2 generates the second chunk, and so on. Finally, bring up the Task Manager (Ctrl+Alt+Del), switch to the Performance tab, and confirm 100% utilization of each core/CPU.

3. If your workstation contains only a single core/CPU, have no fear, you can simulate the same result. Immediately above the pragma, call the OpenMP function *omp_set_num_threads(N)* to force the creation of N threads, even though you have only a single CPU. For example, to run with 2 threads, do the following:

```
omp_set_num_threads(2);

#pragma omp parallel for
for (int yp=0; yp<_pixels; yp++)
{ ... }
```

Even though you'll see no performance benefit, you'll still be able to visualize OpenMP's multi-threaded behavior.

4. Let's color-code the rows so we can see which threads are generating which rows. In the Calculate method, immediately *after* the inner for loop that generates a row of pixels, add code to tag the end of the row with the thread's id:

```
for (int xp = 0; xp < _pixels; xp++) // inner loop to generate row of pixels:
    values[xp] = MandelbrotColor(...);

int threadID = omp_get_thread_num() + 1; // 1..N:

for (int xp = _pixels-5; xp < _pixels; xp++)
    values[xp] = -threadID;
```

Build and run — the right-most edge of each row will now display a color denoting the generating thread (thread 1 in red, thread 2 in blue, ...).

5. If you haven't already, explicitly set the number of threads to 4 or more, and run. How well is the workload spread across the available threads? Here's the code:

```

omp_set_num_threads(4);

#pragma omp parallel for
    for (int yp=0; yp<_pixels; yp++)
    { ... }

```

This case reveals the primary weakness of static scheduling: no effort is made to load-balance the computations. The opposite is *dynamic* scheduling of size 1, where each thread processes 1 row at a time. Modify the *parallel for* pragma to contain an additional *schedule* clause, requesting dynamic scheduling:

```

omp_set_num_threads(4);

#pragma omp parallel for schedule(dynamic)
    for (int yp=0; yp<_pixels; yp++)
    { ... }

```

The effect is dramatic. Does performance change as well? With only 2 cores/CPU's, your application may in fact run a bit slower, given the overhead of dynamic scheduling. But with 4 or more cores/CPU's, the benefit of load-balancing should outweigh the additional scheduling cost. Experiment with larger dynamic chunks, e.g. 20 rows at a time:

```

#pragma omp parallel for schedule(dynamic, 20)
    for (int yp=0; yp<_pixels; yp++)
    { ... }

```

Does this improve performance? Read about other scheduling options available in OpenMP, such as *guided* and *runtime* (press F1 to open Visual Studio's MSDN library, and then search for "OpenMP"). Experiment with these as well.

6. Let's collect some timing results... Assuming your local workstation has multiple cores/CPU's, select the scheduling option that best optimizes performance in your case. Make sure you configure for the Release version, and the appropriate platform (i.e. Win32 or x64). Run without debugging (Ctrl+F5), and record the average execution time of 3 runs:

OpenMP Parallel_{time} on local workstation for Mandelbrot run (-0.70, 0, 2.5, 600): _____, number of cores = _____, speedup = _____

What kind of [speedup](#) are you seeing over the sequential VC++ version we timed earlier? How does it compare to the explicitly-threaded version?

4.5 Multi-threading in C# with the TPL

The *Task Parallel Library*, or TPL, is a component of the Microsoft® *Parallel Extensions to the .NET Framework* (PFX). Currently in beta, PFX is expected to release as part of .NET 4.0. It is currently available for download from <http://msdn.microsoft.com/en-us/concurrency/default.aspx>.

The TPL shares some of the same goals as OpenMP: to provide higher levels of abstraction for shared-memory parallel programming. While OpenMP is platform-neutral, the TPL is focused on the .NET platform and its associated languages (C#, F#, VB, managed VC++, etc.). The TPL is a multi-threading library based on fork-join, offering 3 main constructs for parallelism:

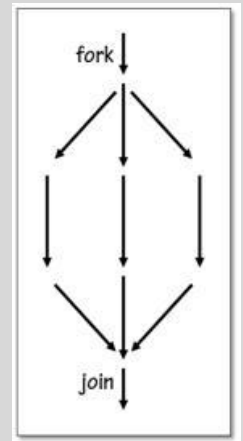
1. `Parallel.For / Do`: *parallel loop execution with support for “work stealing” (i.e. dynamic scheduling)*
2. `Task`: *encapsulates a parallel computation, mapped onto pool of reusable threads for execution*
3. `Future`: *encapsulates a value computed in parallel, any access to value before computation completes will block accessing thread*

There are many advantages to using the TPL over that of the `Thread` class [discussed earlier](#), including load-balancing, thread pooling (which not only reuses threads, but avoids issues related to over-creation), and support for higher-level functionality like canceling, callbacks, and lambda expressions. Here’s a usage summary of the main TPL components:

```
Parallel.For(0, N, loopbody); // fork off threads to execute iterations in parallel:

Task t = Task.Create(code, args); // fork a thread to execute this task:
.
.
.
t.Wait(); // join:

Future<int> f = Future.Create<int>(code); // fork a thread to compute integer result:
.
.
.
Console.WriteLine(f.Value); // output result, if necessary blocking until available:
```



For example, suppose you have 2 independent computations you would like to perform in parallel, *Compute1* and *Compute2*:

```
void Compute1(Object arg)
{ ... }

void Compute2(Object arg)
{ ... }
```

Recall that a single `Object` parameter can be used to pass 0 or more arguments to each method. To execute these computations in parallel, we do the following:

```
Object args1 = ...;
Task t1 = Task.Create(Compute1, args1);

Object args2 = ...;
Task t2 = Task.Create(Compute2, args2);
```

At this point 2 threads from the .NET thread pool are executing `Compute1` and `Compute2`. Once tasks are created, you have the option to cancel, wait for them to complete, or register additional work for them to do (e.g. notification when they complete):

```
t1.Wait(5000); // wait at most 5 seconds...
if (!t1.IsCompleted) // if task didn't complete, cancel it:
    t1.Cancel();
```

```
t2.Wait(); // wait forever for task 2 to complete:
```

The TPL offers the flexibility of specifying computation through *lambda expressions*, i.e. inline code blocks. For example, let's fork off an additional thread to perform yet another computation, this time specifying the work as a lambda expression:

```
Object args3 = ...;
Task t3 = Task.Create(arg =>           // this is a lambda expression with a single parameter arg:
{                                       // start of code block for lambda expression:
    Console.WriteLine("Starting...");
    .
    .
    Console.WriteLine("Done!");
},                                     // end of code block for lambda expression:
args3);                               // actual parameter to lambda expression:
```

While tasks perform computations that do not explicitly return a result (e.g. `Compute1` and `Compute2` are void methods), *futures* are tasks designed to yield a result. For example, we can use a future to perform a long-running computation yielding a double:

```
Future<double> f = Future.Create<double>(() => // a lambda expression taking no arguments:
{                                               // start of code block for lambda expression:
    double result;
    .
    .
    return result;
});                                             // end of code block for lambda expression:
```

This code forks off a thread to execute the computation behind the future. Meanwhile, the creating thread can perform additional work in parallel, eventually harvesting the result when it is needed:

```
.
. // perform other work in parallel with future:
.
Console.WriteLine(f.Value); // harvest result (blocking if not yet available):
```

In the upcoming lab exercise we'll look at a concrete use of the TPL — for parallelizing the C# version of the Mandelbrot application.

4.6 Lab Exercise!

Okay, let's explore parallelization of the C# version of the Mandelbrot application using the Task Parallel Library. In this case, the *Parallel.For* construct in the TPL is going to enable simple and effective parallelization of our app. Note that a solution to the lab exercise is provided in [Solutions\TPL\TPLMandelbrot](#).

1. If you haven't already, start by downloading and installing the most recent release of PFX — the Parallel Extensions to the .NET Framework. This is available from the following site: <http://msdn.microsoft.com/en-us/concurrency/default.aspx>. At the time of this writing, the current release is the June 2008 CTP, which under 32-bit Windows installs into *C:\Program Files\Microsoft Parallel Extensions Jun08 CTP* (and under 64-bit Windows installs into *C:\Program Files (x86)\Microsoft Parallel Extensions Jun08 CTP*).

2. Now open the sequential version of the application provided in [Exercises\02 TPL\TPLMandelbrot](#). Set a reference to the PFX library denoted by the newly-installed assembly file “System.Threading.dll”, as follows: Project menu, Add Reference..., Browse tab, and navigate to the folder where PFX was installed. Select “System.Threading.dll”, and click OK. This assembly should now be listed in the *References* folder in the Solution Explorer window (top-right) of Visual Studio.

3. As we did earlier with [OpenMP](#) and the [Thread class](#), the idea is to parallelize the Mandelbrot class's *Calculate* method by using multiple threads to generate the image rows in parallel. In this exercise we'll use the TPL's *Parallel.For* construct to automatically divide up the iteration space and fork off the threads.

Since parallelization will be encapsulated within the Mandelbrot class, open “Mandelbrot.cs”. Start by adding 2 *using* directives to the top of the source code file, making it easier to use the features offered by the TPL:

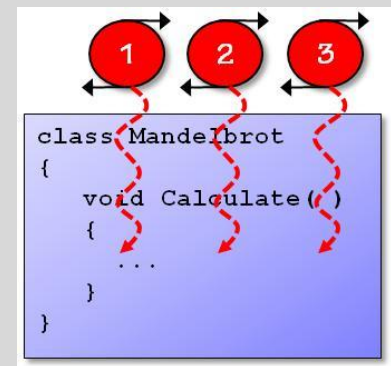
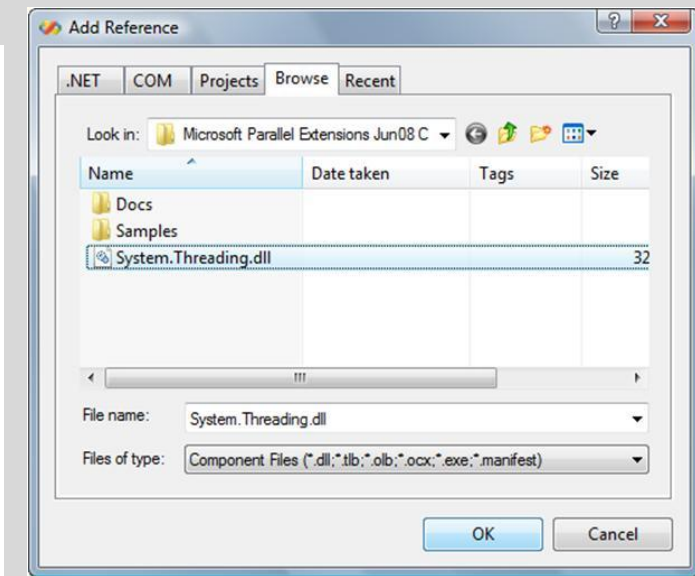
```
using System.Threading;
using System.Threading.Tasks;
```

In the *Calculate* method, locate the outer for loop responsible for row processing, i.e.

```
for (int yp=0; yp<_pixels; yp++)
{
    .
    .
    .
}
```

Let's replace this loop with a parallel version. The easiest approach is to use the TPL's *Parallel.For(start, end, loopbody)*, a static method that forks off a set of threads to execute the *loopbody* in parallel. The loop body can be left as a code block { ... }, and passed to *Parallel.For* as a lambda expression taking one argument, the loop index. Here's the equivalent parallel version:

```
Parallel.For(0, _pixels, yp =>    // a lambda expression taking one argument yp, the loop index:
{
    .
    // the same loop body as before, computing one row of image:
    .
});
```



Notice that the `Parallel.For` statement ends with “);”, which makes sense if you keep in mind that `Parallel.For` is actually a method call, taking 3 parameters: start index inclusive, end index exclusive, and loopbody.

Okay, that’s it! TPL now will take care of dividing up the iteration space and forking off an appropriate number of threads. Run the app (Ctrl+F5), and you should see the image generated by 2 or more threads in parallel. The actual number of threads will depend on the number of cores/CPU’s available on your local workstation.

If your workstation has multiple cores/CPU’s, you should see an immediate performance improvement. Along the bottom of the form, you’ll see the worker id change (flicker?) since the displayed value depends on which thread generated the current row. Finally, bring up the Task Manager (Ctrl+Alt+Del), switch to the Performance tab, and confirm 100% utilization of each core/CPU.

4. Let’s color-code the rows so we can see how many threads are running, and which threads are generating which rows. In the `Calculate` method, immediately *after* the inner loop that generates a row of pixels, add code to tag the end of the row with the thread’s id:

```
for (int xp = 0; xp < _pixels; xp++) // inner loop to generate row:
    values[xp] = MandelbrotColor(...);

int threadID = Thread.CurrentThread.ManagedThreadId;

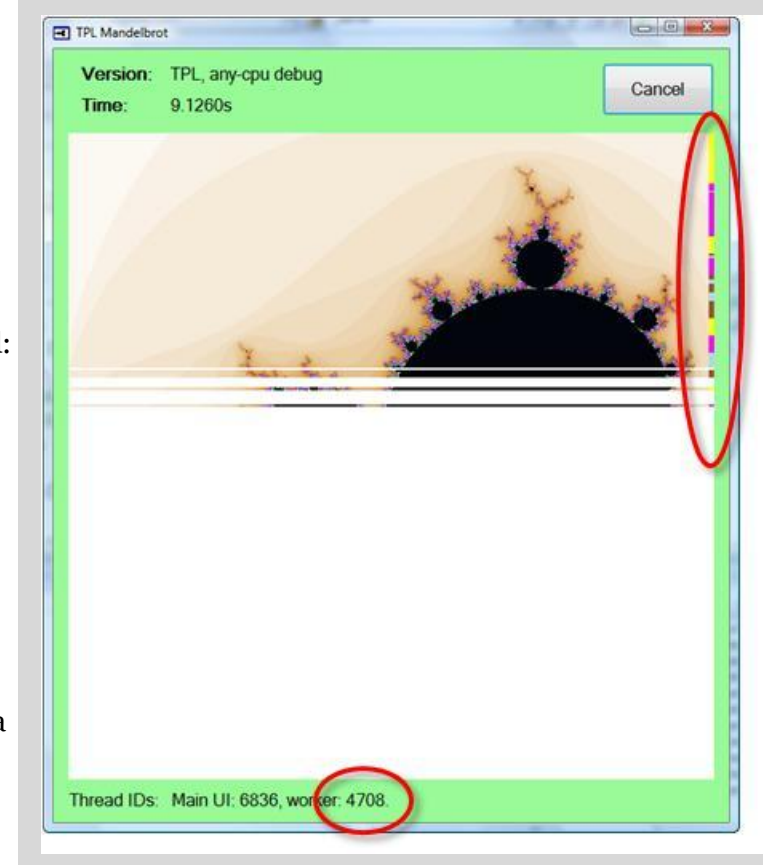
for (int xp = _pixels-5; xp < _pixels; xp++)
    values[xp] = -threadID;
```

Build and run — the right-most edge of each row will now display a color denoting the generating thread. It should be more obvious now that the TPL implements dynamic scheduling, or what the designers call “work-stealing”. Threads perform fewer iterations at a time, and then upon completion, look for additional work. Contrast this with static scheduling, in which threads are assigned a fixed workload before they start executing.

5. If you haven’t already, run the app and press the Cancel button — does the app stop? It does, because each thread returns when cancel is signaled:

```
if (_worker->CancellationTokenPending)
{
    e->Cancel = true;
    return;
}
```

If the threads were instead breaking out of the loop, a better approach would be to use the TPL’s *ParallelState* construct and simply tell the threads to stop. Parallel state is available as a 2nd argument to the lambda expression, like this:



```
Parallel.For(0, _pixels, (yp, state) =>
{
    if (_worker->CancellationTokenPending)
    {
        e->Cancel = true;
        state.Stop();
    }

    .
    .
    .
});
```

Run and test — the app should behave exactly as before.

6. If your workstation contains only a single core/CPU, have no fear, you can create as many threads as you want to visualize the TPL's multi-threaded behavior. For example, here's the code to create 4 threads for each core/CPU in your local workstation:

```
int numProcs    = System.Environment.ProcessorCount;
int idealProcs = System.Environment.ProcessorCount;
int threadsPer = 4;

TaskManagerPolicy tmp = new TaskManagerPolicy(numProcs, idealProcs, threadsPer);
TaskManager        tm  = new TaskManager(tmp);
Func<Object>        setup  = () => { return null; };
Action<Object>      cleanup = arg => { };

Parallel.For<Object>(0, _pixels, 1, setup, (yp, state) =>
{
    .
    .
    .
}, cleanup, tm, TaskCreationOptions.None);
```

Does this improve performance? Experiment with different numbers of threads, what value maximizes performance?

7. Let's collect some timing results... Assuming your local workstation has multiple cores/CPU's, select the # of threads that best optimizes performance in your case. Make sure you configure for the Release version. Run without debugging (Ctrl+F5), and record the average execution time of 3 runs:

TPL Parallel_{time} on local workstation for Mandelbrot run (-0.70, 0, 2.5, 600): _____, number of cores = _____, speedup = _____

What kind of [speedup](#) are you seeing over the sequential C# version we timed earlier? How does it compare to the explicitly-threaded version?

5. A Client-Server Version of the Mandelbrot Application

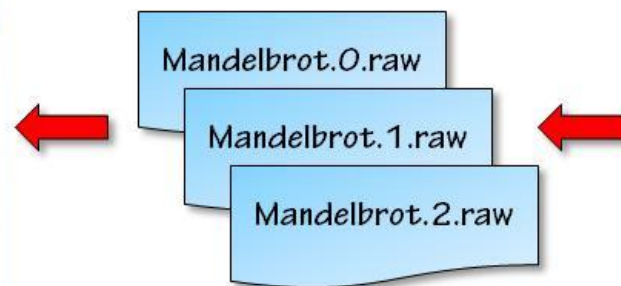
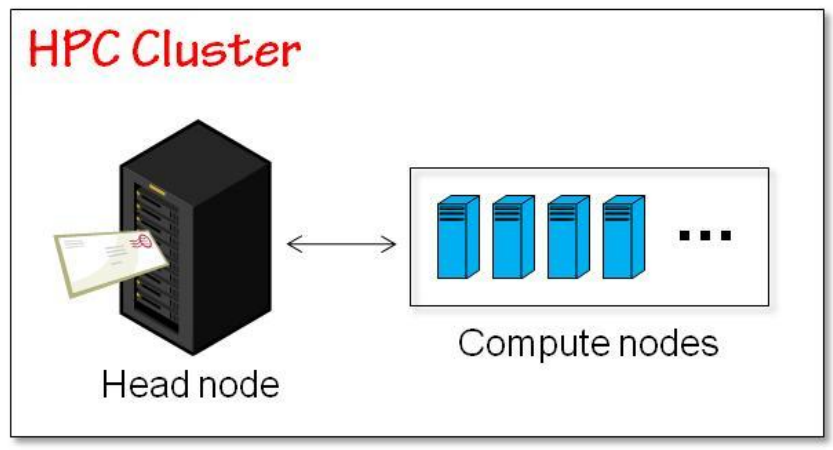
Ultimately, our goal is to maximize performance by running our applications on a Windows-based HPC cluster. Shown on the right, a typical *cluster* is a tightly-coupled set of computers, with one *head* node for submitting workloads, and one or more *compute* nodes for execution. By “tightly-coupled”, this implies the nodes exist as part of the same LAN, and Windows domain.

It’s helpful to think of an HPC cluster as a *time-shared mainframe*, in which jobs are submitted, you wait for execution to complete, and then you fetch the results. If your job is large enough, you might execute across the entire cluster. If your job is smaller, it may well execute alongside other jobs on the cluster. Regardless, one of the key aspects of cluster-based applications is their non-interactive nature — they run on remote hardware with a primitive user interface and minimal user interaction.

In the case of our Mandelbrot application (and most GUI apps), this presents a design problem: users expect a highly-interactive, visual experience, while the cluster offers a high-performance, batch-oriented framework. The implication is that GUI applications must be redesigned in a client-server fashion, with the UI components running on the client’s workstation, the compute-intensive code running on the cluster, and some form of communication between the two.

5.1 Client-Server Mandelbrot

Given its current [design](#) based on the BackgroundWorker class, the Mandelbrot app already adheres to a client-server approach. The main thread acts as the *client*, interacting with the user and updating the UI. The worker thread acts as the *server*, performing the computation row-by-row and communicating the results back to the client. This logical design will make the next step easier, that of physically separating the client and server components into distinct programs that can be run on different machines — the client’s workstation, and the cluster.



```
C:\Windows\system32\cmd.exe
D:\Temp\server>ServerSeqMandelbrot.exe -0.70 0 2.5 600 0 600
** Server-Side Sequential Mandelbrot [64-bit release] **
x:      -0.70
y:       0.00
size:    2.50
pixels:  600
startRow (inclusive): 0
endRow (exclusive):  600
Generated row 0...
Generated row 1...
Generated row 2...
Generated row 3...
Generated row 4...
Generated row 5...
Generated row 6...
```


The critical decision is the method of communication between the two components. For simplicity and flexibility, we'll use files to pass the results from the server back to the client (see diagram, previous page). Each file will contain one row of the Mandelbrot image, allowing one server to create 600 result files, or 600 servers — spread across a cluster — to each create one result file. While files are not necessarily the highest-performing communication mechanism, they are simple to code, and require only minimal network support in the form of a network share. The filenames are of the form “Mandelbrot.R.raw”, where R denotes the row number.

The client-side application will be a WinForms app, and the server-side application will be a console app. In its sequential form, the WinForms app will start a single instance of the console app as a separate Windows process, running on the *same* workstation. The console app is passed command line arguments representing the server-side computation to perform, in particular the characteristics of the Mandelbrot image and the rows to generate. For example, here's the command line to start a single server-side app generating the entire Mandelbrot image:

```
ServerSeqMandelbrot.exe -0.70 0.0 2.5 600 0 600
```

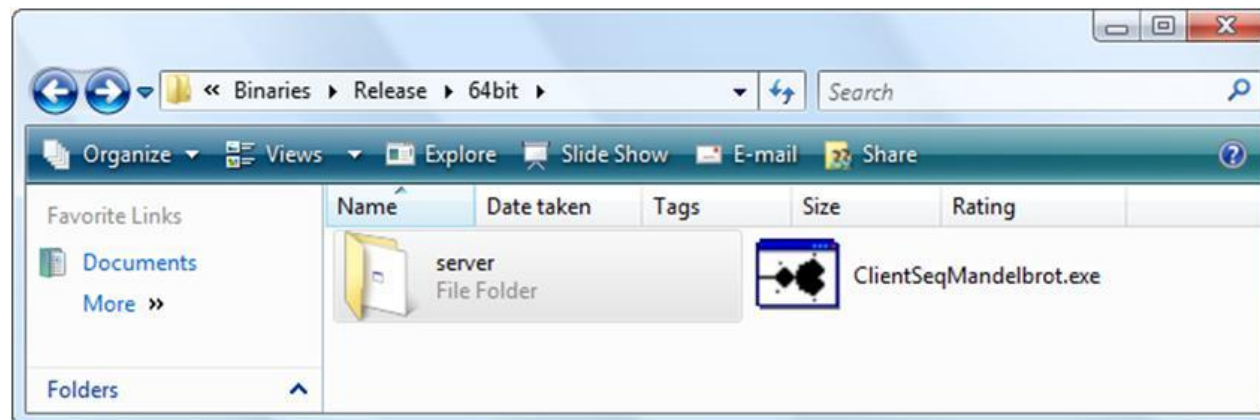
The arguments correspond to x, y, size, pixels, start row inclusive, and end row exclusive. In this case, the server-side app would generate 600 .raw files, “Mandelbrot.0.raw” .. “Mandelbrot.599.raw”.

What if the user presses Cancel? In this case, the client-side app produces a file named “Mandelbrot.cancel”. The server-side app watches for the presence of this file, and if detected, stops processing. Finally, to further simplify the file processing logic, the server-side app actually produces 2 files for every image row: a “.raw” file containing the image data, and a “.ready” file denoting that the data is ready for processing. This helps alleviate the problem of the client trying to read the “.raw” file while the server is still writing it.

Let's look in more detail at the VC++ and C# versions of the client-server Mandelbrot application. Once we have the sequential version up and running on our local workstation, then we'll focus on parallelizing the app across a cluster using Windows HPC Server 2008.

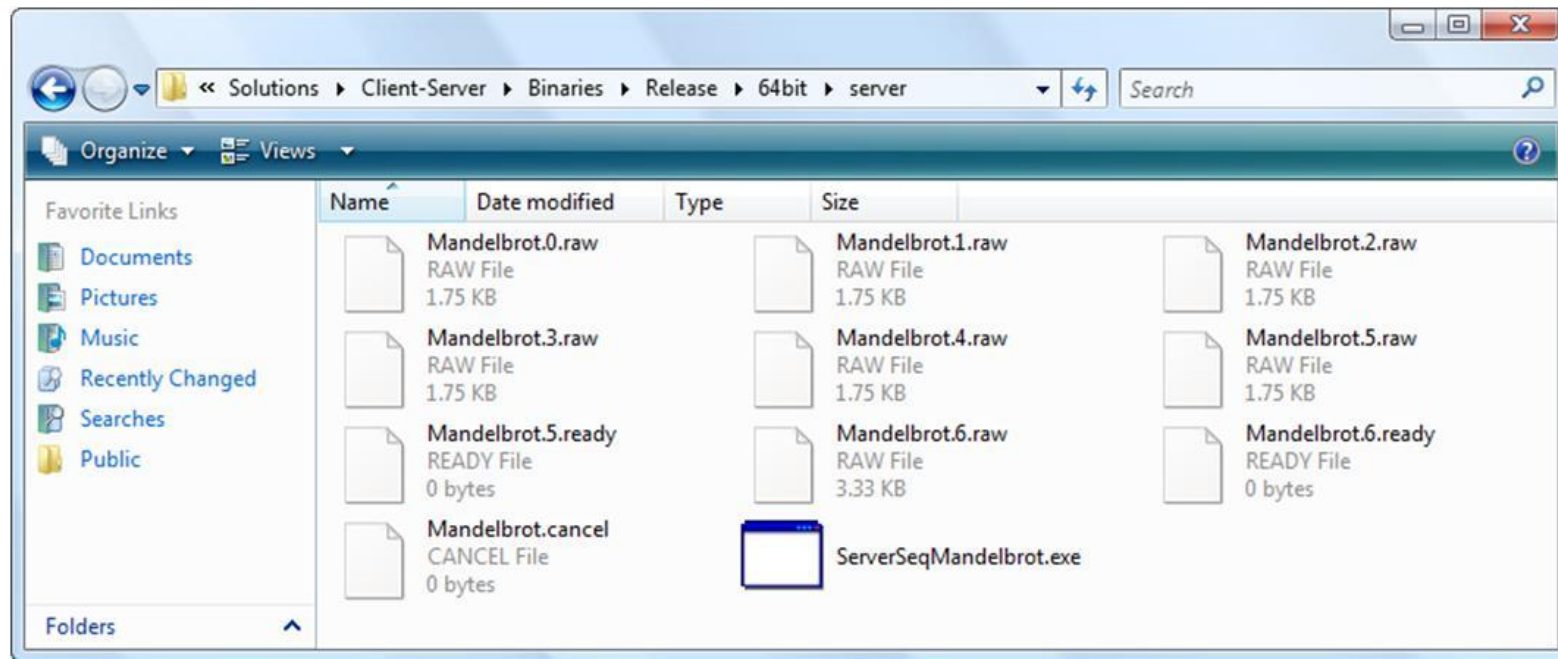
5.2 Running the Sequential Client-Server Mandelbrot App

Start by running the application to see exactly how it behaves. The VC++ app is available in [Solutions\Client-Server\Binaries\Release](#), and the C# app in [Solutions\Client-Server.NET\Binaries\Release](#). Each app consists of two .EXEs, the client-side and the server-side. For example, in the case of VC++, the client-side app is “ClientSeqMandelbrot.exe”, and the server-side app is “server\ ServerSeqMandelbrot.exe”:



Double-click the client-side .exe to start the application running. The app should behave exactly as before, including the cancel functionality. However, each time you click the “Go” button, the client-side launches an instance of the server-side to compute the Mandelbrot image. Click the “Go” button, and immediately bring up the Task Manager (Ctrl+Alt+Del). Click the “Processes” tab, and confirm that both the client-side and server-side .EXEs are listed. Now click the “Cancel” button, and confirm that the server-side .EXE disappears from the list of processes. Finally, click “Go” to start another computation, and notice a new server-side .EXE appears in the Processes list.

Exit the Mandelbrot application, and open the “server” sub-folder. You’ll see the server-side .EXE, along with potentially 1200 other files representing the last Mandelbrot image you generated. For example, here’s what you might see if you canceled the computation immediately after starting it:



In this case the server-side generated 6 rows of the image before detecting the “.cancel” file and stopping.

5.3 The Visual C++ and Visual C# versions of the Sequential Client-Server Mandelbrot App

Okay, let’s take a quick look at the implementation details. Open either the Visual C++ version or the Visual C# version: for VC++, open the solution (.sln) file found in [Solutions\Client-Server\ClientServerSeqMandelbrot](#); for C#, open the solution in [Solutions\Client-Server.NET\ClientServerSeqMandelbrot.NET](#). Each solution contains 2 projects, one for the client-side and one for the server-side. Notice the client-side project appears in **boldface**, which means this is the “Startup” project — when you run (F5), Visual Studio will start this project, i.e. the client-side application. Run the app to make sure all is well; it should build and execute without error.



The client-side [WinForms application](#) follows the same architecture as previous versions, using a main thread to manage the UI and a worker thread to handle the computation. However, instead of performing the computation itself, the worker thread starts a Windows process to run an instance of the server-side application. Expand the client-side project, open the Mandelbrot class (“Mandelbrot.cpp” or “Mandelbrot.cs”), and locate the *Calculate* method. Scroll through the code, and you’ll see the logic to launch the server-side app. Here’s the C# version:

```
//
// C# version:
//
string serverFile = "ServerSeqDotNetMandelbrot.exe";

string serverArgs = String.Format("{0} {1} {2} {3} {4} {5}",
    _x,
    _y,
    _size,
    _pixels,
    startRowInclusive,
    endRowExclusive);

Process server = new Process();
server.StartInfo.FileName = serverFile;
server.StartInfo.Arguments = serverArgs;
server.StartInfo.WindowStyle = ProcessWindowStyle.Hidden;
server.StartInfo.WorkingDirectory = _basedir;
server.Start();
```

At this point the server-side app is running, generating one file per row. Back in the client-side app, the worker thread starts monitoring the “server” sub-folder, watching for the appearance of “.ready” files denoting the availability of data:

```
//
// For each row of the image, read & report for display:
//
int rowCount = 0;

while (rowCount < _pixels)
{
    if (_worker.CancellationPending) // if user wants to cancel, tell the server to stop:
    { ... }

    string[] files = Directory.GetFiles(_basedir, "*.ready"); // any files to process?
    for (int i = 0; i < files.Length; i++, rowCount++)
        this.ProcessRow(files[i]);

    if (files.Length == 0 && server.HasExited) // if server has stopped, we should as well:
        break;
}
```

For each file “.ready” generated by the server-side, the client-side opens the associated “.raw” file, reads the row of data, and reports this row for display in the UI:

```
void ProcessRow(string filename)
{
    string basename = Path.GetFileNameWithoutExtension(filename); // Mandelbrot.YP
    string extension = Path.GetExtension(basename); // .YP
    int yp = Convert.ToInt32(extension.Substring(1)); // this is the row #

    // We have a ".raw" file to process, representing one row, so read it in:
    string rawname = Path.Combine(_basedir, String.Format("Mandelbrot.{0}.raw", yp));
    StreamReader reader;

    while (true) // repeatedly try to open until .RAW file is available:
    {
        try
        {
            reader = new StreamReader(rawname);
            break;
        }
        catch
        { }
    }

    int[] values = new int[_pixels];

    for (int xp = 0; xp < _pixels; xp++)
        values[xp] = Convert.ToInt32(reader.ReadLine());

    reader.Close();

    // Now report this row to UI for display:
    Object[] args = new Object[3];

    args[0] = values;
    args[1] = AppDomain.GetCurrentThreadId();
    args[2] = _serverID;

    _worker.ReportProgress(yp, args);

    // Finally, delete .ready file so we don't process again:
    File.Delete(filename);
}
```

That's it for the client-side.

Now let's focus on the server-side, which is responsible for generating the Mandelbrot image. Recall that the server-side is a [console app](#), since it requires no real user interface — its job is computation only. In Visual Studio, expand the server-side project folder. Start by opening the main method, found in “Main.cpp” or “Program.cs”. In essence, the main method does just two things: (1) processes the command line arguments to determine the characteristics of the computation, and then (2) creates a Mandelbrot object to do the work:

```
//  
// C# version:  
//  
void Main(string[] args)  
{  
    double x      = Convert.ToDouble(args[0]);  
    double y      = Convert.ToDouble(args[1]);  
    double size    = Convert.ToDouble(args[2]);  
  
    int pixels     = Convert.ToInt32(args[3]);  
    int startRowInclusive = Convert.ToInt32(args[4]);  
    int endRowExclusive  = Convert.ToInt32(args[5]);  
  
    Mandelbrot mandelbrot = new Mandelbrot();  
    mandelbrot.Calculate(x, y, size, pixels, startRowInclusive, endRowExclusive);  
}
```

The Mandelbrot class is similar to earlier versions, except that each row of the image is written to a file for processing by the client-side. View the source code for the class (“Mandelbrot.cpp” or “Mandelbrot.cs”), confirm the presence of *MandelbrotColor* that does the bulk of the computation, and then locate the *Calculate* method. As before, the Calculate method generates each row of the Mandelbrot image:

```
int Calculate(double x, double y, double size, int pixels, int startRowInclusive, int endRowExclusive)  
{  
    for (int yp = startRowInclusive; yp < endRowExclusive; yp++)  
    {  
        if (File.Exists("Mandelbrot.cancel")) return -1; // canceled by user:  
  
        // Compute one row of pixels:  
        int[] values = new int[pixels];  
  
        for (int xp = 0; xp < pixels; xp++)  
            values[xp] = MandelbrotColor(yp, xp, y, x, size, pixels);  
  
        // Write values to a file for processing by client-side:  
        string filename = String.Format("Mandelbrot.{0}.raw", yp);  
        StreamWriter writer = new StreamWriter(filename);  
  
        for (int xp = 0; xp < pixels; xp++)  
            writer.WriteLine(values[xp]);  
  
        writer.Close();  
    }  
}
```

```

        // Now let client-side know the file is ready:
        filename = String.Format("Mandelbrot.{0}.ready", yp);
        writer = new StreamWriter(filename);
        writer.Close();
    }

    return 0; // success!
}

```

Again, the only real difference is that for each row of the Mandelbrot image, the server-side now generates two files, “Mandelbrot.YP.raw” and “Mandelbrot.YP.ready” (where YP is the row number). That’s it!

5.4 Lab Exercise!

Very shortly we’re going to parallelize the client-server Mandelbrot application for execution on a Windows-based cluster. In preparation, let’s collect some timing results for the sequential version so we can accurately compute [speedup](#) values for the parallel versions. Your goal here is to record the average execution time of 3 sequential runs of the client-server Mandelbrot application.

1. First, run and time just the server-side component. Open a console window (“black screen”): Start menu, *cmd.exe*. Navigate (cd) to the folder containing either the VC++ or C# binaries: [Solutions\Client-Server\Binaries\Release](#) or [Solutions\Client-Server.NET\Binaries\Release](#). Navigate (cd) to the “server” sub-folder, and run the server-side .EXE: “ServerSeqMandelbrot.exe” or “ServerSeqDotNetMandelbrot.exe”. For example, here’s the command line to run the VC++ version:

```
ServerSeqMandelbrot.exe -0.70 0.0 2.5 600 0 600
```

Use a similar command line for the C# version (only the .EXE name is different). Record the average execution time here:

Sequential Server-side_{time} on local workstation for Mandelbrot run (-0.70, 0, 2.5, 600): _____

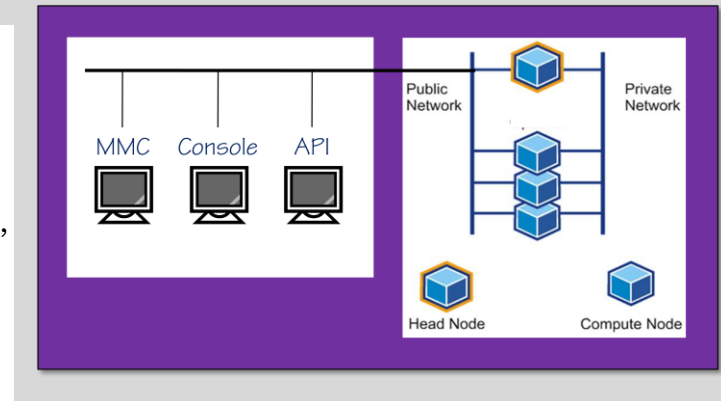
2. Now run and time the client-server application. Simply double-click the client-side .EXE, either “ClientSeqMandelbrot.exe” or “ClientSeqDotNetMandelbrot.exe”. Record the average execution time here:

Sequential Client-Server_{time} on local workstation for Mandelbrot run (-0.70, 0, 2.5, 600): _____



6. Working with Windows HPC Server 2008

Let's assume you have a working Windows HPC Server 2008 cluster at your disposal (if not, you can safely skip this section, or if you need help setting one up, see [Appendix A](#)). Jobs are submitted to the cluster in a variety of ways: through an MMC plug-in such as the *Microsoft® HPC Pack Job Manager*, through Windows PowerShell™ or a console window ("black screen"), or through custom scripts / programs using the cluster's API. We'll focus here on using the MMC plug-ins to submit jobs from your local workstation.



The first step is to install *Microsoft HPC Pack 2008* on your local workstation, which can be running a 32-bit or 64-bit version of Windows® (XP, Windows Vista®, Windows Server®2003/2008). *Microsoft HPC Pack 2008* will install the client-side utilities for interacting with the cluster. The *Microsoft HPC Pack 2008* is available for purchase from Microsoft, may be downloaded from the MSDN Subscriber Download site or a free evaluation version may be downloaded from <http://www.microsoft.com/hpc>.

Cluster configurations vary nearly as much as snowflakes :-). For the purposes of this tutorial, let's assume the following hypothetical cluster:

Name of head node:	<i>headnode</i>
Name of compute nodes:	<i>compute1, compute2, ...</i>
Run-as credentials for job execution:	<i>domain\hpcuser</i>
Network share accessible by all nodes:	<i>\\headnode\Public</i>
Network share on every node (mapped to C:\Apps):	<i>\\headnode\Apps, \\compute1\Apps, \\compute2\Apps, ...</i>

Users of the cluster are assumed to have full R/W access to these network shares, as well as access to the cluster itself.

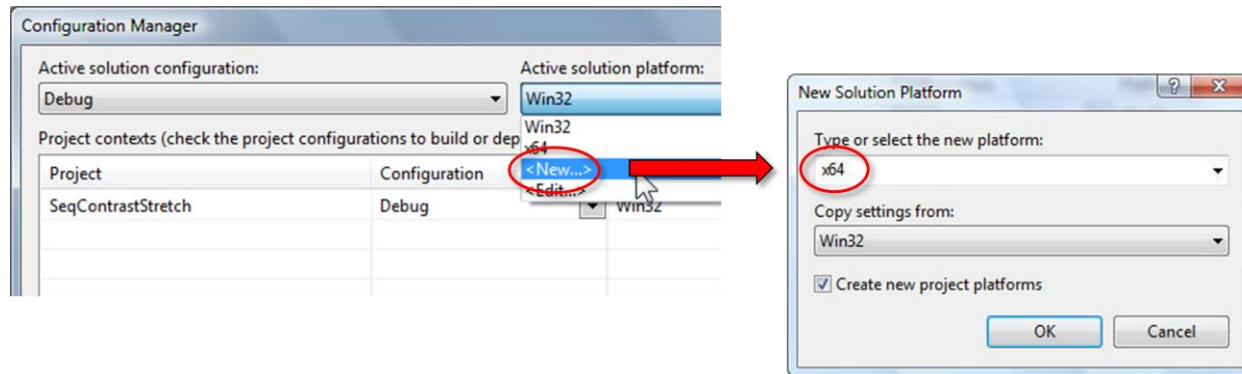
6.1 Submitting a Job to the Cluster

In the [previous section](#) we presented a client-server version of the Mandelbrot application. The sole motivation for this version is to encapsulate the computation into a server-side component that can be executed across a Windows-based cluster. Let's use the server-side component to gain some experience with Windows HPC Server 2008.

1. Open either the VC++ or the C# version of the client-server Mandelbrot application. For VC++ open the solution (.sln) file in [Solutions\Client-Server\ClientServerSeqMandelbrot](#), for C# open the solution in [Solutions\Client-Server.NET\ClientServerSeqMandelbrot.NET](#). Build a Release version of your application for deployment to the cluster (and if appropriate, a 64-bit version). This is selected via Visual Studio's Standard toolbar:



If you are working with VC++ and “x64” doesn’t appear in the toolbar drop-down, add this configuration to your project via the Build menu, *Configuration Manager*. If x64 is not an option in the New Solution Platform dialog (shown below to the right), then you need to exit Visual Studio and install the 64-bit compilers for Visual C++.



2. After the build, your .EXE can be found in [Solutions\Client-Server\Binaries\Release](#) (VC++) or [Solutions\Client-Server.NET\Binaries\Release](#) (C#). Drill down to the “server” sub-folder and locate the server-side application, either “ServerSeqMandelbrot.exe” or “ServerSeqDotNetMandelbrot.exe”. If your cluster is configured to allow .NET applications to run from network shares, simply copy your .EXE to a unique folder on the cluster’s public share, e.g. [\\headnode\Public\DrJoe](#). Note that by default, .NET is configured for security reasons to prevent the execution of non-local code. [*If you are not sure, the simplest way to find out is to deploy your .EXE to a network share, remote desktop into one of the cluster nodes, open a console window, and try running the app by typing [\\headnode\Public\DrJoe\appname.exe](#).*]

If your cluster does not support network execution of .NET apps, deploy the .EXE locally to a unique folder on each node of the cluster, e.g. `C:\Apps\DrJoe`. The folder name **must** be the same on every node. One approach is to copy your .EXE to the cluster’s public share ([\\headnode\Public](#)), remote desktop into each node, and then copy from the public share to the local folder `C:\Apps\DrJoe`. If available, another approach is copying directly to network shares that map to a local folder on each node, e.g. [\\headnode\Apps](#), [\\computer1\Apps](#), etc. Finally, if you have an account with administrative rights on the cluster, you can deploy via Windows HPC Server’s *clusrun* utility. Clusrun executes a DOS command on one or more nodes of the cluster, shown here in a screen snapshot:

```
C:\Windows\system32\cmd.exe
Microsoft Windows [Version 6.0.6001]
Copyright (c) 2006 Microsoft Corporation. All rights reserved.

C:\Users\hummel>set CCP_SCHEDULER=headnode

C:\Users\hummel>clusrun /user:minihipc\hummel mkdir C:\Apps\DrJoe
Enter the password for 'minihipc\hummel' to connect to 'headnode':
Remember this password? (Y/N)N
----- COMPUTE1 returns 0 -----
----- HEADNODE returns 0 -----

----- Summary -----
2 Nodes succeeded
0 Nodes failed

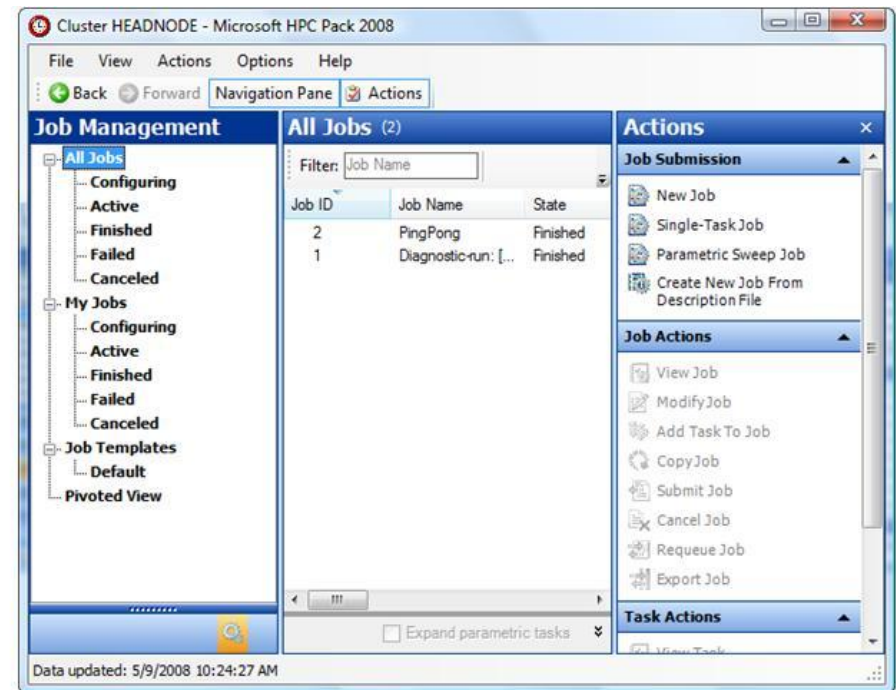
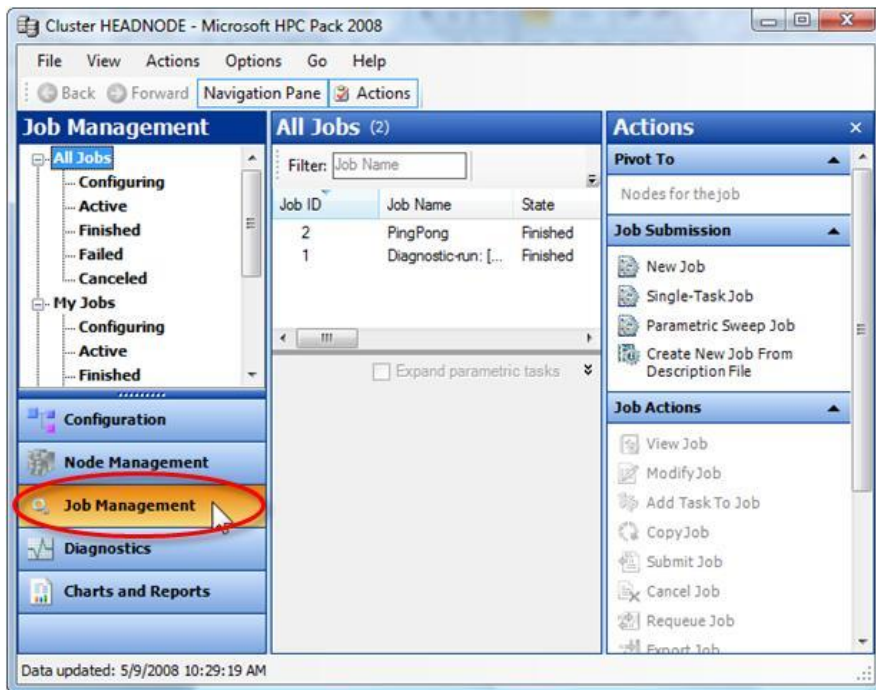
C:\Users\hummel>
```

To use clusrun for deployment, copy the .EXE to a public share, open a console window, and execute the following commands:

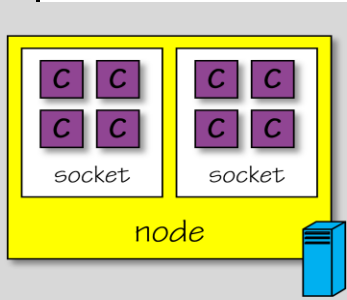
```
set CCP_SCHEDULER=headnode
clusrun /user:domain\username mkdir C:\Apps\DrJoe
clusrun /user:domain\username copy \\headnode\Public\Drjoe\*.exe C:\Apps\DrJoe
```

The first command sets an environment variable denoting the machine name of the cluster's headnode. The second command creates a unique deployment folder on each node. Finally, the third command deploys the .EXE across the cluster. The user account "domain\username" must have administrative rights on the cluster.

3. Now that the application has been deployed, let's submit a job to run the .EXE. If you have administrative rights on the cluster, startup the *Microsoft HPC Pack Cluster Manager* and click the Job Management tab (screenshot below left); this plug-in allows you to submit jobs as well as monitor the cluster. Otherwise, assuming you have user rights on the cluster, startup the *Microsoft HPC Pack Job Manager* (below right). If this is the first time, you'll be asked to supply the machine name of the cluster's head node. Here are screen snapshots of the two MMC plug-ins:



4. Create a new job via Actions menu, Job Submission > Create New Job (dialog is shown to right, click for full-size image). Supply a job name, a project name (optional), and then decide how many execution cores you need to run your job. For example, since this is a sequential run, explicitly set the minimum and maximum to 1. However, if this were an OpenMP or TPL run, you'd want to specify the number of cores on a single node; e.g. if your cluster nodes have 4 cores per node, set the minimum and maximum to 4. And if this were an MPI run, you'd set the minimum and maximum to the range of cores you can effectively use. Note that if you want to run on N cores, don't be afraid to set both the min and max to N.



Note that you can schedule a job by other types of resources, e.g. Node, Socket, or Core. A node refers to an entire compute node. A socket refers to physical CPU chip in a node. A core refers to an execution core in a socket. For example, a *dual quadcore PC* is a single node with 2 sockets and 4 cores per socket, for a total of 8 execution cores.

Finally, if this is a performance/timing run, check the box for “*Use assigned resources exclusively for this job*”. This maximizes your performance, but may waste resources from the perspective of other cluster users. Don't submit just yet...

5. Once the job is configured, you need to add one or more tasks — tasks are the actual executable units of work scheduled on the compute nodes. Let's assume this is a simple job with just one task to execute your application. Click on “Task List” in the new job window, and then click the Add button to add a new task to the job.

Configure the task by assigning a name (optional), and then specifying the *Command line*. The command line is essentially what you would type if you were running the job from your local workstation via a console window (“black screen”). For a sequential, OpenMP or TPL app, this would be something like:

```
app.exe argument1 argument2 ...
```

For an MPI app, you would specify:

```
mpiexec mpiapp.exe argument1 argument2 ...
```

In our case, the command line is either

```
ServerSeqMandelbrot.exe -0.70 0.0 2.5 600 0 600
```

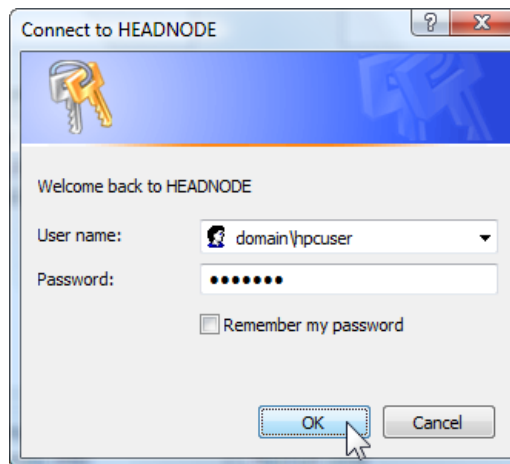
or

```
ServerSeqDotNetMandelbrot.exe -0.70 0.0 2.5 600 0 600
```


Next, set the *Working directory* to the location where you deployed the .EXE, e.g. [\\headnode\Public\DrJoe](#), or C:\Apps\DrJoe if you deployed locally to each node. Redirect Standard output and error to text files; these capture program output and error messages, and will be created in the working directory (these files are very handy when troubleshooting). Finally, select the minimum and maximum number of execution cores to use for executing this task. The range is constrained by the values set for the overall job: use a min/max of 1 for sequential apps, the number of cores on a node for OpenMP/TPL apps, and a range of cores for MPI apps. Click Save to save the configuration.

6. You should now be back at the job creation window, with a job ready for submission. First, let's save the job as an XML-based template so it's easier to resubmit if need be: click the "Save Job as..." button, provide a name for the generated description file, and save.

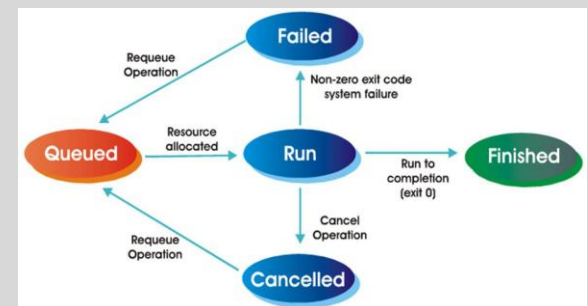
Now submit the job to the cluster by clicking the Submit button. You'll be prompted for the job's "Run-as" credentials, i.e. the username and password for the identity under which this job will execute on the compute nodes. On some clusters, there is a single account under which all jobs are run, e.g. "domain\hpcuser":



On other clusters, you supply your personal username and password. Regardless, this identity is critically important for it determines job rights during execution — such as resource access to machines, network shares, folders, and files. Note that Windows HPC Server 2008 manages these credentials securely across the cluster. [*Note that you also have the option to securely cache these credentials on your local workstation by checking the "Remember my password" option. Later, if you need to clear this cache (e.g. when the password changes), select Options menu, Clear Credential Cache in the HPC Cluster or Job Manager.*]

7. When a job has been submitted, it enters the *queued* state. When the necessary resources become available, it enters the *run* state and starts executing. At that point, it either *finishes*, *fails*, or is *cancelled*. You monitor the job using the HPC Cluster or Job Managers. You can monitor all jobs or just your own, and jobs in various stages of execution: configuring, active, finished, failed, or cancelled.

When a job completes (by finishing or failing), open the working directory you specified in the [task](#) (e.g. [\\headnode\Public\DrJoe](#)) and view the task's redirected Standard output and error files (these were "_OUT.txt" and "_ERR.txt" in the task-based [screen snapshot](#) shown earlier). If the job failed to run, troubleshooting tips are given in [Appendix B](#).



8. If you want to resubmit the job, use the description file we saved: Actions menu, Job Submission > Create New Job from Description File. You can submit the job exactly as before, or adjust the parameters and resubmit. Note that if you change the number of cores allocated to the job, you need to reconfigure the task accordingly (“Task List”, select task, Edit).

Finally, if you want to use Windows PowerShell or a console window to submit your jobs (or likewise automate with a script), here are two examples. First, submitting the sequential version of the VC++ Mandelbrot app via a console window (Start, cmd.exe):

```
> job submit /scheduler:headnode /jobname:MyJob /numprocessors:1-1 /exclusive:true /workdir:\\headnode\Public\DrJoe  
/stdout:_OUT.txt /stderr:_ERR.txt /user:domain\hpcuser SeqMandelbrot.exe
```

If you want to run the C# version, recall the .exe is named “SeqDotNetMandelbrot.exe”. Let’s repeat, this time via Windows PowerShell (Start, Microsoft HPC Pack 2008 > Windows PowerShell™):

```
> $job = new-hpcjob -scheduler "headnode" -name "MyJob" -numprocessors "1-1" -exclusive 1  
> add-hpctask -scheduler "headnode" -job $job -workdir "\\headnode\Public\DrJoe" -stdout "_OUT.txt" -stderr  
"_ERR.txt" -command "SeqMandelbrot.exe"  
> submit-hpcjob -scheduler "headnode" -job $job -credential "domain\hpcuser"
```

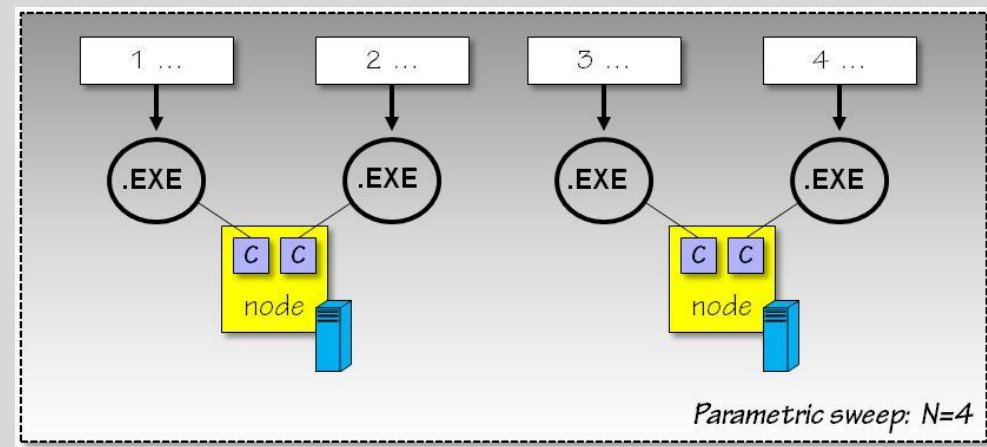
For more info, type “job submit /?” in your console window or “get-help submit-hpcjob” in Windows PowerShell.

6.2 Lab Exercise – Parallelization with Parametric Sweep!

1. First, collect timing results for execution of the server-side component on one node of your cluster. The goal is to record the average execution time of 3 sequential runs of the server-side .EXE (“ServerSeqMandelbrot.exe” or “ServerSeqDotNetMandelbrot.exe”) so we can accurately compute [speedup](#) values for the upcoming parallel versions. Make sure you have deployed the Release version (64-bit if appropriate) on the cluster. After each job submission, the execution time is available in the redirected stdout file. When you are done, record the average time here:

Sequential Server-side_{time} on one node of cluster for Mandelbrot run (-0.70, 0, 2.5, 600): _____

2. Now let’s get a feeling for how effective parallelization is going to be for the Mandelbrot application. In Windows HPC Server 2008, the fastest way to parallelize an application is to use *parametric sweep*. A parametric sweep is when multiple instances of an application are run across the cluster, each instance solving a different aspect of the problem. For example, in step 1 above, we ran a single server-side instance of the Mandelbrot app to compute all 600 rows of the image. In a parametric sweep, we would run N instances of the server-side Mandelbrot app, each computing 600/N rows of the image. If our cluster contains at least N execution cores, the parametric sweep should execute N times faster! [For example: if $N = 4$, then each instance computes 150 rows, and sweep should finish in 1/4 the time.] The trade-off in using a parametric sweep is



the potential difficulty of collecting the individual results back into a coherent answer.

3. The first step in using a parametric sweep is to parameterize the app based on Windows HPC Server's framework. The idea is simple: we configure Windows HPC Server to generate a *sweep* of values V in the range $1..N$, stepping by S . An instance of the application is created for each sweep value V , and V is then passed to the application as a command-line argument. For example, let's suppose our cluster has 4 execution cores available. For best performance, we want to run 4 instances of our application, one per core. This implies we want to divide the problem space into 4 equal chunks.

In the case of our Mandelbrot application with 600 rows, we want instance 1 to generate rows (0, 4, 8, ..., 596), instance 2 to generate rows (1, 5, 9, ..., 597), instance 3 to generate rows (2, 6, 10, ..., 598), and instance 4 to generate rows (3, 7, 11, ... 599). Since some rows take much longer than others, this approach spreads the workload more equally across the instances. To iterate in this way, we'll have to redesign the server-side component to accept a different set of command line arguments. In particular, we'll pass the characteristics of the Mandelbrot image (x , y , size, and pixels), along with the instance number (1..4) and the total number of instances (4). For example, the first instance of the server-side .EXE would be started as follows (VC++ version):

```
ServerSeqMandelbrot.exe -0.70 0.0 2.5 600 1 4
```

Or in the case of the C# version:

```
ServerSeqDotNetMandelbrot.exe -0.70 0.0 2.5 600 1 4
```

Let's modify the server-side component of the Mandelbrot application to accept this redesign. Open the provided VC++ or C# version of the server-side console app, found in [Exercises\03 ParametricSweep\Client-Server\ServerSeqMandelbrot](#) (VC++) or [Exercises\03 ParametricSweep\Client-Server.NET\ServerSeqMandelbrot.NET](#) (C#). [*Solutions to this part of the lab exercise are available in [Solutions\ParametricSweep](#).*] Start by modifying the main method, which processes the command line arguments. First, change the usage output, and the parsing of the arguments:

```
//  
// C# version:  
//  
if (args.Length != 6)  
{  
    Console.WriteLine("Usage: ServerSeqMandelbrot.exe x y size pixels instance numInstances");  
    Console.WriteLine("Exiting...");  
    return;  
}  
  
double x = Convert.ToDouble(args[0]);  
double y = Convert.ToDouble(args[1]);  
double size = Convert.ToDouble(args[2]);  
int pixels = Convert.ToInt32(args[3]);  
int instance = Convert.ToInt32(args[4]);  
int numInstances = Convert.ToInt32(args[5]);
```

Now add code to compute the starting and ending rows for this instance, along with the iteration step value:

```
int startRowInclusive = instance - 1;
int endRowExclusive   = pixels;
int stepBy             = numInstances;
```

Finally, when we perform the calculation, pass *stepBy* as an additional parameter to the Calculate method:

```
mandelbrot.Calculate(x, y, size, pixels, startRowInclusive, endRowExclusive, stepBy);
```

Likewise, modify the Mandelbrot class's Calculate method to accept this additional parameter, and modify the method so the for loop steps the iteration correctly:

```
int Calculate(double x, ..., int stepBy)
{
    for (int yp = startRowInclusive; yp < endRowExclusive; yp += stepBy)
    {
        ...
    }
}
```

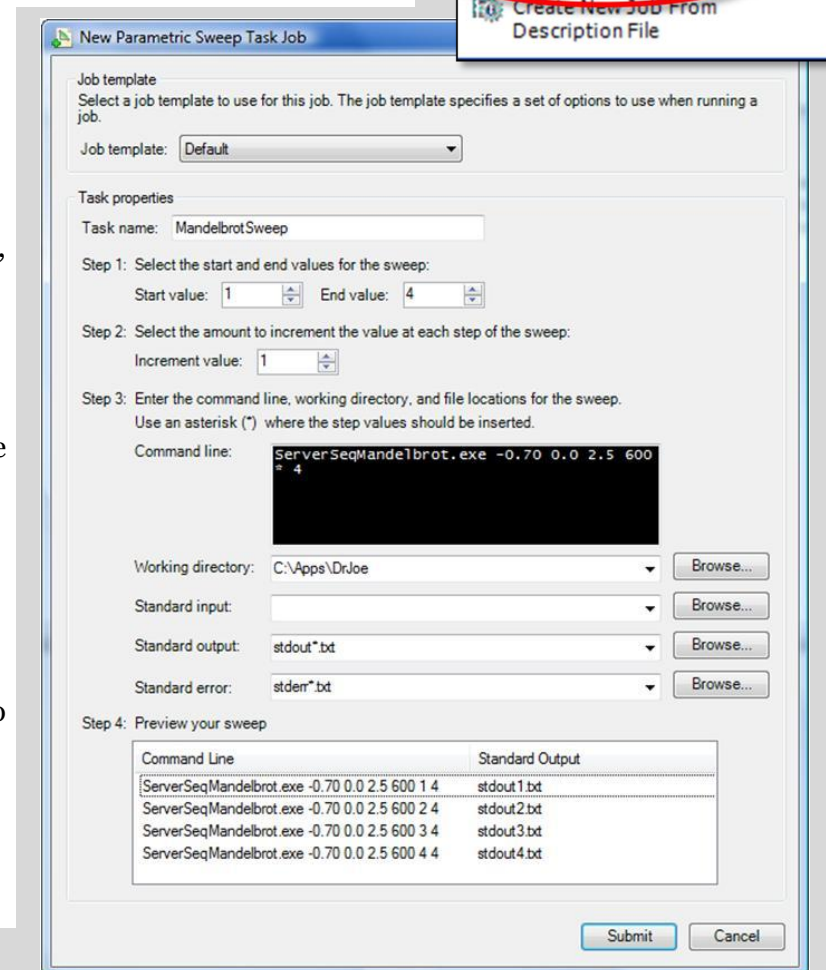
That's it! Build a release version of the app, and deploy the .EXE as you did earlier.

4. Now that the application has been redesigned for a parametric sweep, let's submit a *Parametric Sweep Job* to the cluster. Using the HPC Cluster Manager or Job Manager, click on the Parametric Sweep Job option. This will open a window for configuring the parameters of the job, and for specifying the generic command line from which the instances are started (see screen snapshot to right). Assuming a total of N=4 execution cores, configure the *Start value* as 1 and the *End value* as 4. Leave the *Increment value* as 1, since we want a total of 4 instances numbered 1, 2, 3, 4. Next, enter the *Command line* for execution, using * where the sweep value is inserted. For the VC++ version of the app, the command line is:

```
ServerSeqMandelbrot.exe -0.70 0.0 2.5 600 * 4
```

Notice the preview pane (under Step 4) expands to show you the actual command lines that will be generated when the job runs. Finally, set the working directory based on where you deployed the .EXE (e.g. `\\headnode\Public\DrJoe` or `C:\Apps\DrJoe`), and redirect stdout and stderr — use filenames of the format “stdout*.txt” and “stderr*.txt” so that each instance produces a different set of files. When you are ready, click Submit, supply the necessary run-as credentials, and see how long the sweep takes... Start counting, the job should finish in roughly 1/4 the time of your earlier sequential run!

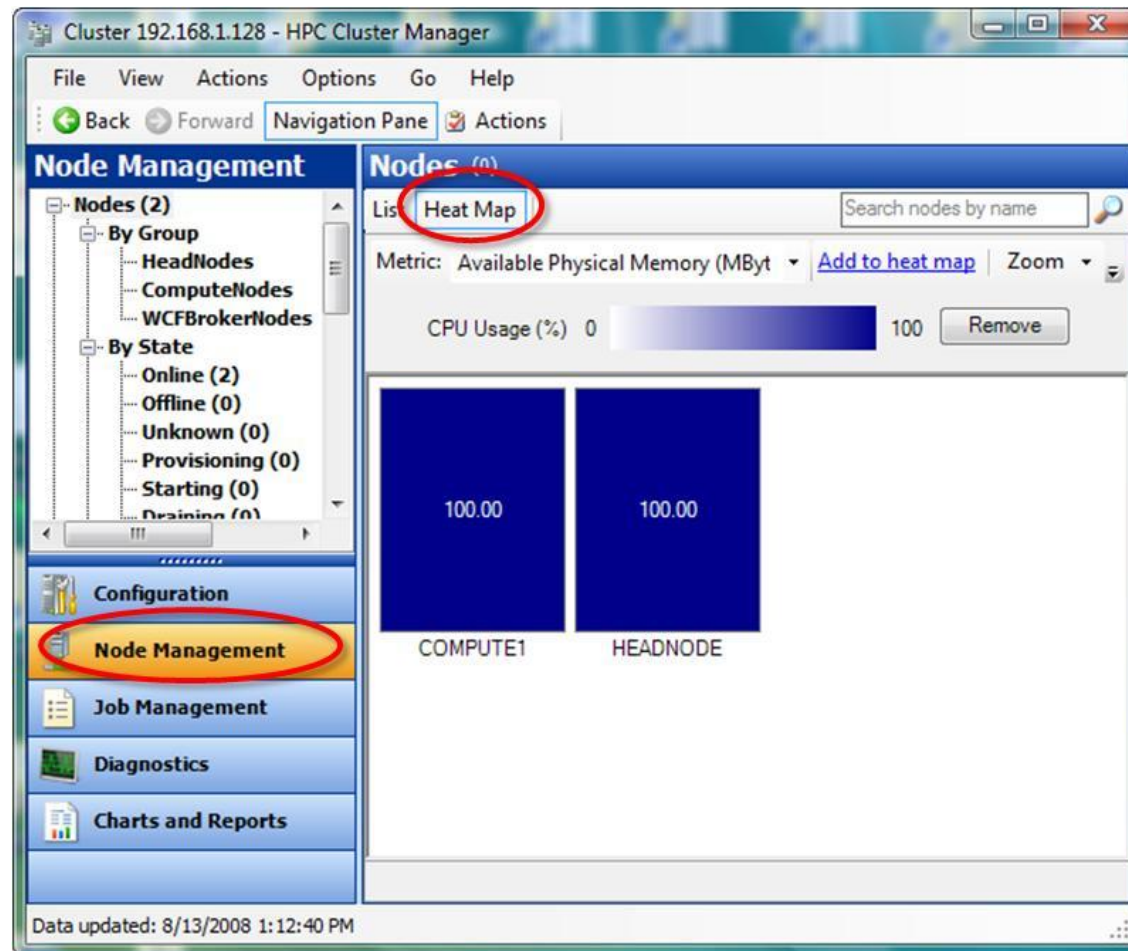
5. If the job finished successfully, open the working directory(ies) and view the stdout*.txt files. If you deployed locally, these files will be spread across the compute



nodes. What's the maximum execution time? Record this time, and compute the [speedup](#) based on the sequential time you recorded in step 1:

Parametric Sweep Server-side_{time} on cluster for Mandelbrot run (-0.70, 0, 2.5, 600): time = _____, number of cores = _____, speedup = _____

6. If you haven't already, bring up Windows HPC Server's *Heat Map* so you can visualize how resources are being used by the cluster during job execution. Submit another parametric sweep, and then switch to the Node Management tab. Click on Heat map, and monitor CPU usage. Here's a screen snapshot of my mini-cluster with 4 cores, running at 100%:



100% utilization is a good thing in this case! Experiment with visualization of other resources, such as available memory and network usage.

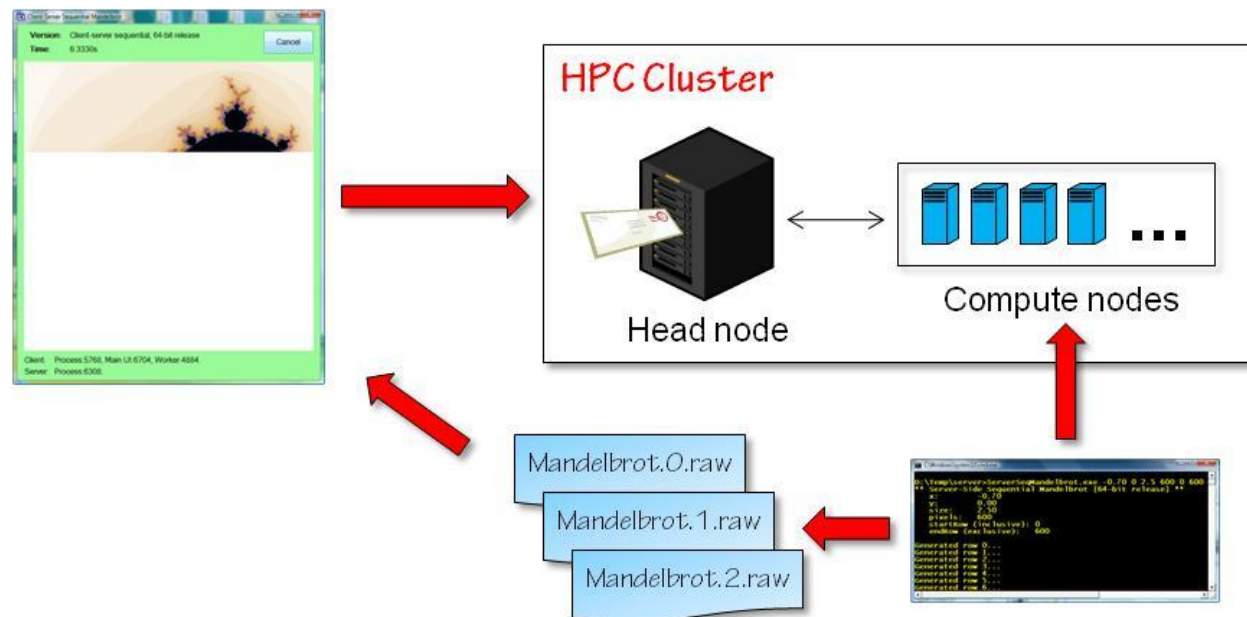
7. Parallelizing the Client-Server Mandelbrot Application using Windows HPC Server 2008

The goal of the client-server version of the Mandelbrot app is to enable parallel, cluster-based execution. We saw the beginnings of this in the [previous lab exercise](#), using Windows HPC Server's *parametric sweep* to easily parallelize the Mandelbrot computation. However, only the server-side component was involved, and this was manually deployed and submitted to the cluster for execution. The next step is to involve the client-side component, automating the deployment and submission process, and displaying the results.

The subtle implication is that the client-side component needs to programmatically interact with Windows HPC Server 2008 in order to hide the details of cluster-based execution — application deployment, job submission, and result aggregation. In the case of the Mandelbrot app, generating an image will involve the following steps:

1. Client-side component copies server-side .EXE to cluster's public share
2. Client-side component communicates with head node to create a new job
3. Client-side component adds job task to copy server-side .EXE to local folder on each node
4. Client-side component adds job task to run server-side .EXE, configuring task's working directory so image files appear in public share
5. Client-side component submits job for execution
6. Server-side component runs on one or more nodes of cluster, generating image files...
7. Client-side component monitors public share for image files, displaying as they appear...

The following graphic depicts the roles played by each component:



7.1 Communicating with Windows HPC Server 2008 – The Job Scheduler API

The main function of Windows HPC Server 2008 is to execute jobs on the underlying cluster. This functionality is performed by the *Job Scheduler*, which is responsible for queuing jobs, allocating resources, launching applications, and monitoring execution. The architecture of Windows HPC Server's Job Scheduler is shown to the right.

The Job Scheduler executes on the head node, offering clients access to the cluster over a secure, remoting-based communication channel. The *Job Scheduler API* is defined as a set of .NET interfaces, with an interop library providing COM support. The MMC plug-ins presented in [Section 6](#) use this API, along with Windows PowerShell and the console-based utilities.

The API is available as part of the *SDK for Microsoft HPC Pack 2008*. The SDK is installed on your local development workstation, which can be running a 32-bit or 64-bit version of Windows (XP, Windows Vista, Windows Server 2003/2008). Download the SDK from <http://go.microsoft.com/fwlink/?linkID=127031>. By default the SDK's installation folder is *C:\Program Files\Microsoft HPC Pack 2008 SDK*.

The API consists of a number of interfaces for interacting with the cluster. The most commonly-used interfaces include

```
IRemoteCommand
IScheduler
ISchedulerCollection
ISchedulerJob
ISchedulerNode
ISchedulerTask
```

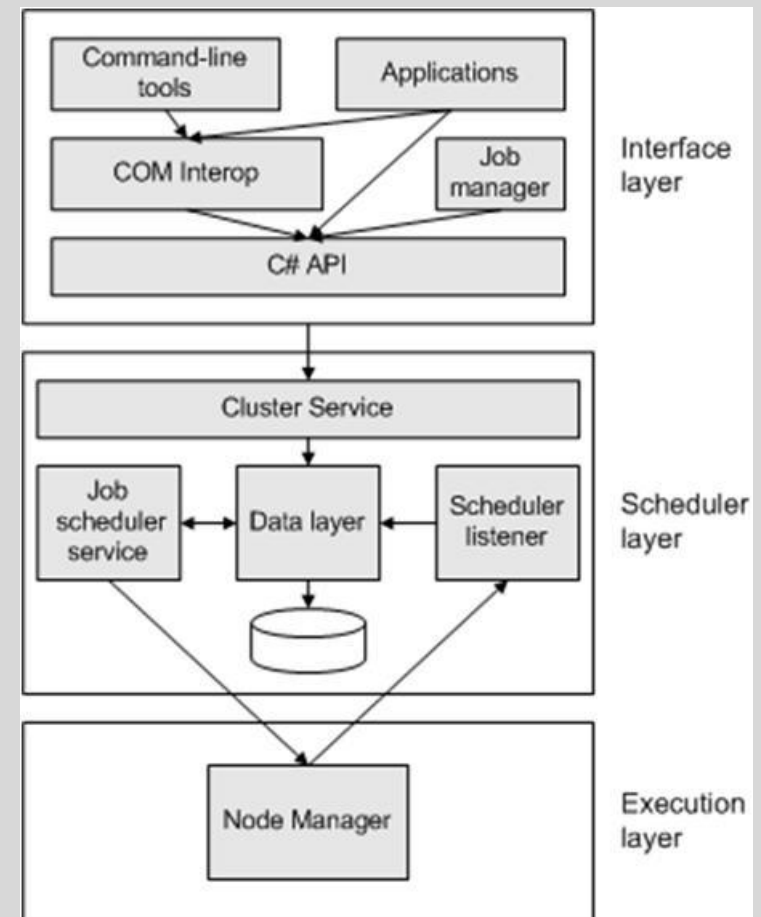
For example, use *IScheduler* to connect to a cluster, *ISchedulerCollection* to browse information about cluster resources (jobs, nodes, etc.), and *ISchedulerJob* to create and configure jobs for cluster execution. The API is implemented across 3 .NET DLLs, which are found in *C:\Program Files\Microsoft HPC Pack 2008 SDK\Bin*:

```
Microsoft.Hpc.Scheduler.dll
Microsoft.Hpc.Scheduler.Properties.dll
Microsoft.Hpc.Scheduler.Session.dll
```

When programming against the API, reference each of these .DLLs from your Visual Studio project (Project menu, Add Reference..., Browse tab).

7.2 Executing a Job on the Cluster

Executing a job on the cluster consists of 3 steps: connecting, creating, and submitting. To connect to the cluster, you create an object of type *Scheduler*, which implements the



```
interface IScheduler
{
    void Connect(string cluster);

    ISchedulerJob CreateJob();
    ISchedulerCollection GetJobList(...);
    ISchedulerCollection GetNodeList(...);

    void SetInterfaceMode(...);
    void SubmitJob(...);
    :
    :
}
```


IScheduler interface. You then call the object's *Connect* method, specifying the name or IP address of the headnode:

```
//  
// C#:  
//  
using Microsoft.Hpc.Scheduler;  
  
IScheduler scheduler = new Scheduler();  
scheduler.Connect("headnode"); // headnode's machine name or IP address:
```

Windows checks to make sure your user account — i.e. the credentials under which you are currently logged in — has rights to access the cluster. If so, you are successfully connected (otherwise a security exception is thrown). The next step is to create a new job, and then a new task. These objects implement the *ISchedulerJob* and *ISchedulerTask* interfaces, allowing you to configure the job much like we did [earlier](#) using the MMC plug-ins. For example, let's configure the task to run *hostname*, which outputs the name of the machine on which it is run:

```
ISchedulerJob job = scheduler.CreateJob();  
ISchedulerTask task = job.CreateTask();  
  
task.CommandLine = "hostname"; // output name of machine on which we run:  
task.WorkDirectory = @"\\headnode\Public"; // output written to cluster's public share:  
task.StdOutFilePath = "hostname.out.txt";  
job.AddTask(task);
```

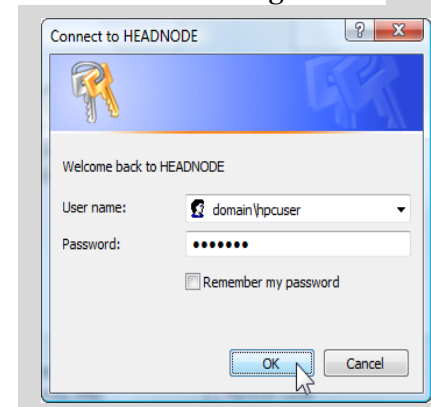
The task's *CommandLine* property is the most important, specifying the program to run on the cluster. Other properties include the working directory, where to redirect stdout/stderr, the number of requested resources (cores/sockets/nodes), and so on. Once configured, the task is then added to the job. The last step is to submit the job, supplying the user credentials under which the job will run. If the password is not given, the user is prompted to enter one, in this case using a forms-based (i.e. GUI) interface:

```
string runasUser = @"domain\hpcuser";  
  
scheduler.SetInterfaceMode(false /*GUI, not console*/, (IntPtr) null /*no parent window*/);  
scheduler.SubmitJob(job, runasUser, null /*prompt for pwd*/);  
  
MessageBox.Show("Job " + job.Id + " successfully submitted for execution.");
```

At this point the job has been submitted to the cluster, validated by Windows HPC Server 2008, and ready for execution. Note that the result is identical to submitting the job by hand using *HPC Pack Cluster* or *Job Manager*; if you open either of these MMC plug-ins, you would see the job listed in the queue (click “All Jobs”).

7.3 Configuring the Client-Server Mandelbrot Application for Cluster Execution

In its current state, the client-side component of the Mandelbrot app [launches a Windows process](#) to execute the server-side .EXE. The problem is that the .EXE runs on the client's workstation, and so the computation is not off-loaded. Let's modify the client-side component to use the Job



Scheduler API and launch the server-side .EXE on the cluster. In particular, our first approach is to mimic what we did earlier by hand — a [parametric sweep](#). The end result will be a parallel, cluster-based solution to the Mandelbrot set.

Before we start, note that HPC applications often contain cluster-specific details, such as the headnode's machine name or IP address. A smart approach is to store such details in a .NET *configuration* file instead of compiling them into the application binary. This makes it easier to retarget the application for different clusters. The cluster-based client-server Mandelbrot app has 6 configuration parameters:

ServerSideEXEName	name of server-side .EXE file
ClusterHeadnode	machine name or IP address of headnode
PublicNetworkShare	path to cluster's public network share (and typically a unique folder on this share)
LocalWorkDirectory	path to a local directory on each node (and typically a unique folder in this directory)
RunAsUserName	run-as credentials for running programs on cluster — username
RunAsPassword	run-as credentials for running programs on cluster — password

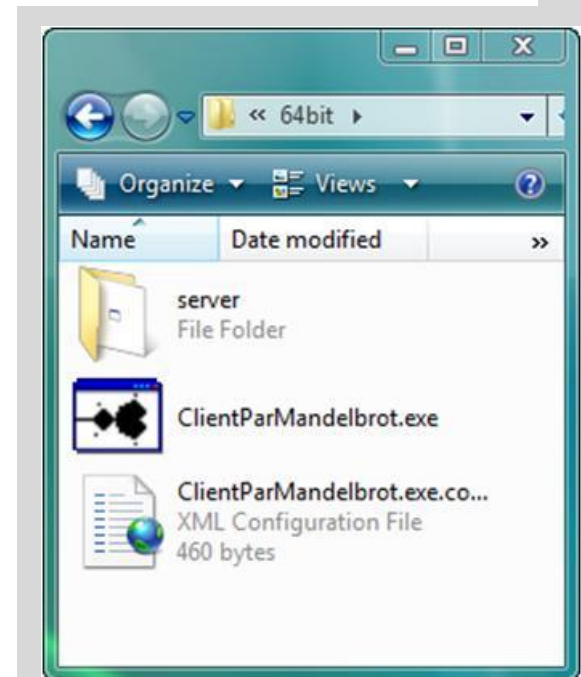
For example, here's the .config file when running the cluster-based Mandelbrot app on my mini-cluster. The settings should not contain spaces:

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <appSettings>
    <add key="ServerSideEXEName" value="ServerParMandelbrot.exe" />
    <add key="ClusterHeadnode" value="headnode" />
    <add key="PublicNetworkShare" value="\\headnode\Public\DrJoe" />
    <add key="LocalWorkDirectory" value="C:\Apps\DrJoe" />
    <add key="RunAsUserName" value="MINIHPC\hummel" />
    <add key="RunAsPassword" value="" />
  </appSettings>
</configuration>
```

Leaving *RunAsPassword* blank causes the application to prompt for the password.

The first step is to get the application up and running on your cluster. The .EXEs are available in [Solutions\ParallelClient-Server\SweepByCore\Binaries\Release](#) (VC++) or [Solutions\ParallelClient-Server.NET\SweepByCore\Binaries\Release](#) (C#). As shown in the screen snapshot (bottom-right), you'll find the client-side .EXE along with its associated .config file. Open an instance of Notepad, and drag-drop the .config file into the empty Notepad window. Modify the .config file based on your cluster's configuration. Here are the steps performed by the client-side component, and how the configuration settings come into play:

1. Makes directory <PublicNetworkShare>, then copies <ServerSideEXEName> to <PublicNetworkShare>
2. Connects to the Windows HPC Server 2008 cluster <ClusterHeadnode>
3. Creates a new job on the cluster
4. For each node in the cluster, adds a task that makes directory <LocalWorkDirectory>, then copies <ServerSideEXEName> from <PublicNetworkShare> to <LocalWorkDirectory>
5. Adds a task that performs a parametric sweep 1..N of <LocalWorkDirectory>\<ServerSideEXEName>,



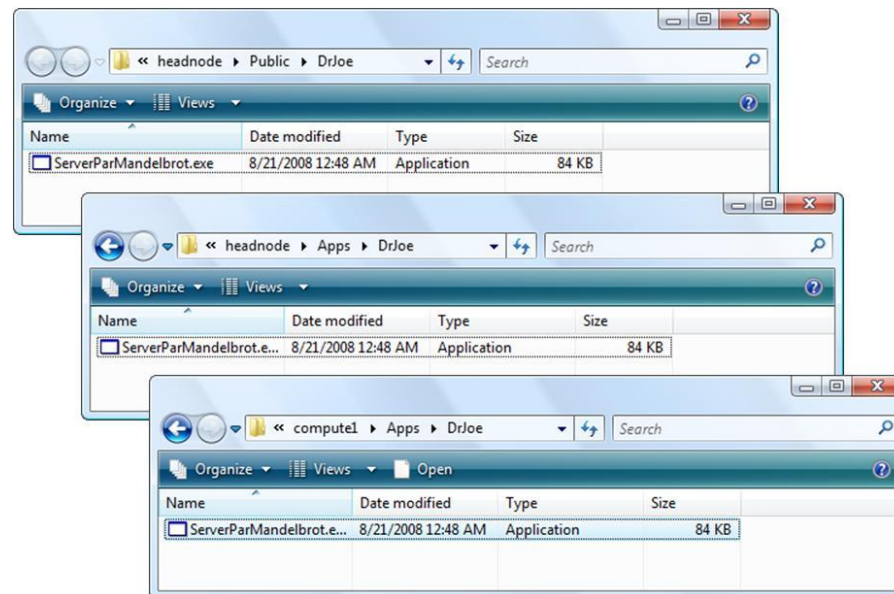
- where N = total number of cores in the cluster; the working directory is <PublicNetworkShare>
6. Submits job for execution with run-as credentials <RunAsUserName>, <RunAsPassword>
 7. Monitors <PublicNetworkShare> for generated image files...

In step 4, notice that tasks are used to deploy the server-side .EXE across the cluster. This is a common technique in HPC applications — to use the cluster not only for computation, but also for chores like cleanup, deployment, and collecting results.

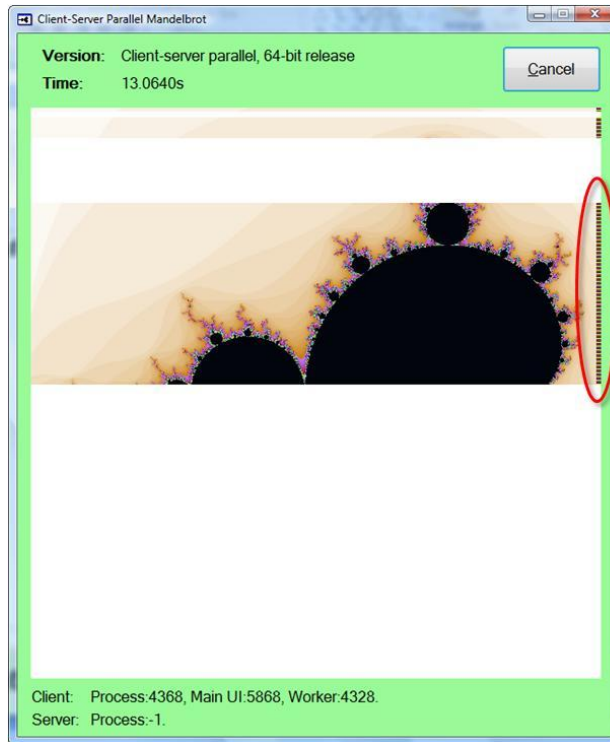
Run the client-side .EXE to see if the application works correctly with your configuration settings. If so, make note of these settings for later. If the app does not run correctly, here are some debugging tips:

- Can you manually connect to the cluster? Check using one of the MMC plug-ins.
- Can you manually create and submit a job to the cluster? Test the run-as credentials <RunAsUserName>, <RunAsPassword>.
- If any of the settings contains spaces, eliminate them (e.g. directory paths should not contain spaces).
- Did <ServerSideEXEName> deploy to <PublicNetworkShare>? If not, check spellings, then check security permissions by copying yourself.
- Did <ServerSideEXEName> deploy to the local folder <LocalWorkDirectory> on each compute node? Does <LocalWorkDirectory> exist? If not, check spellings, and make sure <RunAsUserName> has permission on each node to create this directory and copy into it.
- Does <PublicNetworkShare> contain result files (*.raw, *.ready, *.txt)? If all is well, the folder should contain 600 .raw files and 600 .ready files, as well as N stdout .txt files and N stderr .txt files. Check the stderr & stdout .txt files for error messages.
- Open the HPC Pack Cluster or Job Manager plug-in, and check the job queue. Did the job fail? What was the error message? Open the job's task list, which task(s) failed? Why?

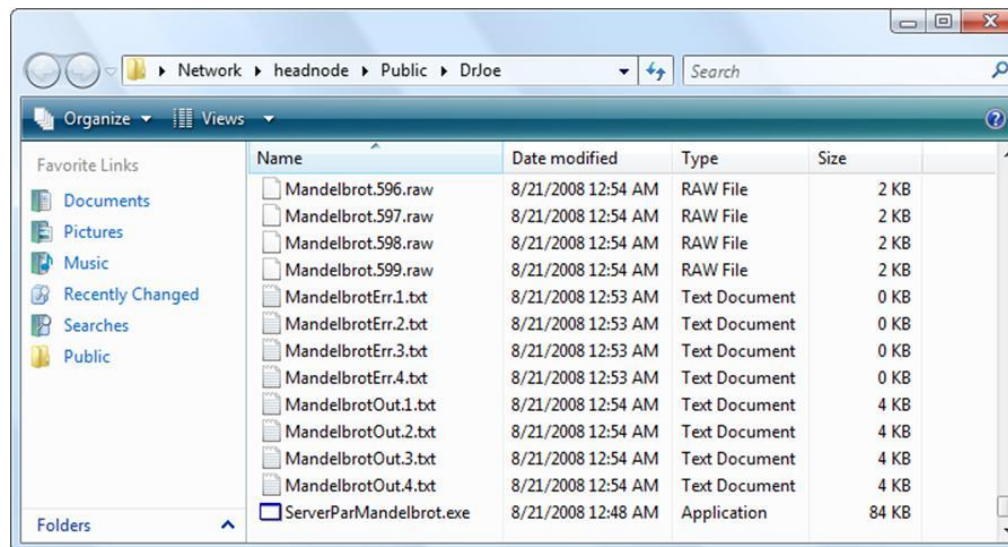
Here are some screen snapshots of what you should be seeing. First, a successful deployment across my 2-node mini-cluster:



Next, the beginnings of a successful execution (note the color coding along the right-edge, one per instance up to 8 colors):

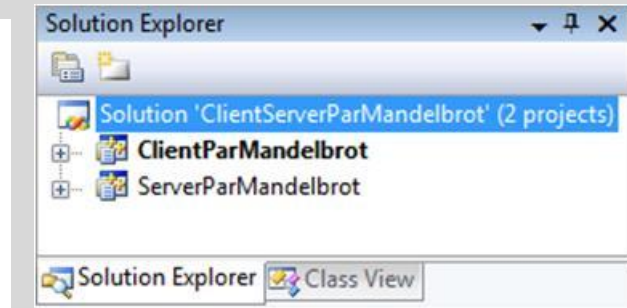


Afterwards, the contents of <PublicNetworkShare>:



7.4 Implementing the Cluster-based Client-Server Mandelbrot Application

Let's look at the implementation of the cluster-based version. Open either the VC++ version found in [Solutions\ParallelClient-Server\SweepByCore\ClientServerParMandelbrot](#), or the C# version in [Solutions\ParallelClient-Server.NET\SweepByCore\ClientServerParMandelbrot.NET](#). As [before](#), this single solution consists of 2 projects, the client-side GUI project and the server-side console project (see Solution Explorer to right).



Expand the client project, open the “*app.config*” file, and change the configuration settings to match the correct values for your cluster. Run (F5), and the app should execute successfully on the cluster. Next, confirm that the client project references the underlying .DLLs for the Job Scheduler API: “*Microsoft.Hpc.Scheduler.dll*”, “*Microsoft.Hpc.Scheduler.Properties.dll*”, “*Microsoft.Hpc.Scheduler.Session.dll*”. In the VC++ version: Project menu, References..., References pane. In the C# version, expand the References folder in the Solution Explorer window. Finally, notice the importing of the API namespaces *Microsoft.Hpc.Scheduler* and *Microsoft.Hpc.Scheduler.Properties* (see “Mandelbrot.h” or “Mandelbrot.cs”).

Keep in mind that the only difference between this cluster-based version and the earlier [sequential](#) client-server version is where the server-side .EXE runs — the cluster, or the client’s workstation. This is controlled by the client-side component, and entirely encapsulated by the *Mandelbrot* class. In the client project, open the source code for the Mandelbrot class (“Mandelbrot.cpp” or “Mandelbrot.cs”). Locate the constructor, and notice that the private class member `_basedir` is set to `<PublicNetworkShare>`:

```
_basedir = ConfigurationSettings.AppSettings["PublicNetworkShare"]; // C#:
```

The working directory of the server-side component will be set to `_basedir`, causing the output files to be written to `<PublicNetworkShare>`. This allows the client-side component to monitor `<PublicNetworkShare>` and harvest the results.

The majority of work occurs in the *Calculate* method. Skim through its implementation. You’ll see that the method communicates with the cluster using the Job Scheduler’s API, performing the 7 steps discussed in the previous section:

1. Makes directory `<PublicNetworkShare>`, then copies...
2. Connects to the cluster...
3. Creates a new job on the cluster
4. For each node in the cluster, adds a task that makes directory `<LocalWorkDirectory>`, then copies...
5. Adds a task that performs a parametric sweep...
6. Submits job for execution...
7. Monitors `<PublicNetworkShare>`...

In step 1, notice that we use .NET to create the public network share (*Directory.CreateDirectory*) and deploy the server-side .EXE (*File.Copy*):

```
void Calculate(Object sender, DoWorkEventArgs e) // C#:  
{  
    .  
    .  
    .  
}
```

```
// (1) deploy server-side .EXE to public network share:
String serverEXEName = ConfigurationSettings.AppSettings["ServerSideEXEName"];

if (!Directory.Exists(_basedir)) // create if it doesn't exist already:
    Directory.CreateDirectory(_basedir);

String localexe = System.AppDomain.CurrentDomain.BaseDirectory;
localexe = Path.Combine(localexe, "server");
localexe = Path.Combine(localexe, serverEXEName);

String destexe = Path.Combine(_basedir, serverEXEName);

File.Copy(localexe, destexe, true /*overwrite*/);
```

The client has to deploy the .EXE since the cluster does not have access to the client workstation. However, once the .EXE is on the cluster's public network share, we use the cluster to do the rest of the work. In preparation, steps 2 and 3 connect to the cluster and create a new job. For maximum performance, the job is configured to request exclusive access to as many cores as possible. Notice the resource *UnitType* is set to *Core* (other options include *Socket* or *Node*), and we iterate across the nodes of the cluster to compute the total number of available cores:

```
// (2) connect to cluster:
IScheduler scheduler = new Scheduler();

scheduler.Connect(ConfigurationSettings.AppSettings["ClusterHeadnode"]);

// (3) create & configure new job:
ISchedulerJob job = scheduler.CreateJob();

job.Name = ConfigurationSettings.AppSettings["RunAsUserName"] + " Mandelbrot";
job.IsExclusive = true; // we want resources to ourselves for better perf
job.RunUntilCanceled = false; // stop as soon as tasks finish/fail

job.AutoCalculateMin = false; // disable so we can manually set number of resources:
job.AutoCalculateMax = false;
job.UnitType = JobUnitType.Core;

// we want to use all the cores in the cluster, so let's count how many there are:
ISchedulerCollection nodes = scheduler.GetNodeList(null, null); // all nodes:

int numCores = 0;
foreach (ISchedulerNode n in nodes) // iterate across all nodes in cluster:
    numCores += n.NumberOfCores;

job.MinimumNumberOfCores = 1; // request 1..ALL cores in the cluster:
job.MaximumNumberOfCores = numCores;
```

```
interface IScheduler
{
    void Connect(string cluster);
    ISchedulerJob CreateJob();
    ISchedulerCollection GetNodeList(...);
    .
    .
    .
}
```

Once the job is created and configured, tasks are added to perform the actual work. The first set of tasks are responsible for deploying the server-side .EXE across the cluster. For each node in the cluster, we create a task that runs on that particular node, executing two DOS commands — one to create the local folder (*mkdir*), and a second to copy the .EXE (*copy /Y*, which overwrites if necessary):

```
// (4) deploy server-side .EXE across the cluster:
localexe = Path.Combine(ConfigurationSettings.AppSettings["LocalWorkDirectory"], serverEXEName);

ISchedulerTask task;

foreach (ISchedulerNode n in nodes) // for each node in cluster:
{
    task = job.CreateTask();
    task.CommandLine = String.Format("mkdir {0} & copy /Y {1} {2}",
        ConfigurationSettings.AppSettings["LocalWorkDirectory"],
        destexe, // defined in step 1:
        localexe);

    // run copy command on this node, and only this node:
    task.RequiredNodes = new StringCollection();
    task.RequiredNodes.Add(n.Name);

    // run this task just once on this node:
    task.MinimumNumberOfCores = 1;
    task.MaximumNumberOfCores = 1;

    // name task in a unique way so we can refer to it later:
    task.Name = "Deploy-to-" + n.Name;

    // finally, add to job for eventual execution:
    job.AddTask(task);
}
```

```
interface ISchedulerJob
{
    void AddTask(ISchedulerTask task);
    ISchedulerTask CreateTask();
    .
    .
    .
}

interface ISchedulerTask
{
    .
    .
    .
}
```

The deployment task(s) are followed by the computation task, i.e. the [parametric sweep](#) of the server-side component that performs the Mandelbrot image computation. We configure this task exactly as we did [before](#), creating N instances of the server-side .EXE, one per core:

```
// (5) perform parametric sweep to parallelize Mandelbrot image generation:
task = job.CreateTask();

task.Name = "Mandelbrot Parametric Sweep";
task.IsParametric = true;
task.StartValue = 1; // parametric sweep 1..N:
task.EndValue = numCores;

String theArgs = String.Format("{0} {1} {2} {3} * {4}",
    _x,
```



```

    _y,
    _size,
    _pixels,
    numCores);

task.CommandLine      = String.Format("{0} {1}", localexe, theArgs);
task.WorkDirectory    = _basedir;
task.StdoutFilePath   = "MandelbrotOut.*.txt";
task.StderrFilePath   = "MandelbrotErr.*.txt";

// this task must wait for all deployment tasks to finish before starting:
task.DependsOn = new StringCollection();
foreach (ISchedulerNode n in nodes)
    task.DependsOn.Add("Deploy-to-" + n.Name);

// finally, add to job for eventual execution:
job.AddTask(task);

```

Note that the computation task “*depends on*” the deployment task(s) — the computation cannot proceed until deployment is complete. Windows HPC Server allows for the specification of dependencies between tasks based on task names. In this case, the computation task “*Mandelbrot Parametric Sweep*” depends on each deployment task “*Deploy-to-**”.

At this point the job is ready for submission to the Job Scheduler. We set the interface mode based on whether this is a console app (*true*) or a GUI app (*false*), fetch the run-as credentials from the configuration file, and submit the job for execution:

```

// (6) job is ready, submit for execution:
scheduler.SetInterfaceMode(false /*GUI*/, (IntPtr) null /*no parent*/);

String runAsUser = ConfigurationSettings.AppSettings["RunAsUserName"];
String runAsPwd  = ConfigurationSettings.AppSettings["RunAsPassword"];
if (runAsPwd == "") // force prompt if there's no password:
    runAsPwd = null;

scheduler.SubmitJob(job, runAsUser, runAsPwd);

```

```

interface IScheduler
{
    void          Connect(string cluster);
    ISchedulerJob CreateJob();
    ISchedulerCollection GetNodeList(...);

    void SetInterfaceMode(...);
    void SubmitJob(...);
    void CancelJob(...);
    .
    .
    .
}

```

Upon submission, the job is validated by Windows HPC Server, and if successful, moved to the queued (“ready”) state for execution. As soon as the requested resources become available, the job starts running on the cluster.

Barring a job failure, the client-side component knows the server-side component will eventually start generating Mandelbrot image files. So the last step is for the client-side component to start monitoring the output folder for results. This is no different than [before](#), except that the client now monitors <PublicNetworkShare> (represented by the private class member *_basedir*):

```

// (7) that's it, starting monitoring output folder for results:
_startTime = System.Environment.TickCount;

```

```
int rowCount = 0;

while (rowCount < _pixels)
{
    if (_worker.CancellationPending)
    { ... }

    // process any files that have appeared:
    string[] files = Directory.GetFiles(_basedir, "*.ready");

    for (int i = 0; i < files.Length; i++, rowCount++)
        this.ProcessRow(files[i]);

    // if no new files were generated *and* the job has finished, then exit loop:
    if (files.Length == 0)
    {
        job.Refresh(); // get current state of job on cluster:

        if (job.State == JobState.Canceled || job.State == JobState.Failed || job.State == JobState.Finished)
            break;
    }
} //while
```

To prevent an infinite loop, the client checks whether any files were generated by the server, and if not, then checks to see if the job has finished, failed, or canceled. A subtle issue is the call to *job.Refresh()*, which is required to retrieve the job's current state. Why? In the design of the Job Scheduler API, the various client-side objects — job, task, node, etc. — are copies of the actual objects residing on the headnode. As time progresses, these client-side copies become out-of-date. Thus, a *Refresh()* is a remote call to the headnode for a new copy.

8. Shared-memory Parallelization using Parametric Sweep, Thread Class, OpenMP, and the TPL

Now that we have a working cluster-based application, let's experiment with various parallelization strategies. In particular, let's time the parametric sweep, and then compare it against various [multi-threaded techniques](#), namely the *Thread* class, *OpenMP*, and *TPL*.

8.1 Lab Exercise – Shared-memory Parallelization on the Cluster!

1. Fetch some of the execution times for the Mandelbrot app recorded in earlier lab exercises. First, from [section 3.4](#), the time for a sequential run on your local workstation (this is the original Mandelbrot app, not the client-server version):

Sequential_{time} on local workstation for Mandelbrot run (-0.70, 0, 2.5, 600): _____

Next, the parallel times on your local workstation for [explicit multi-threading](#) (i.e. Thread class), [OpenMP](#), and the [TPL](#):

Thread Parallel_{time} on local workstation for Mandelbrot run (-0.70, 0, 2.5, 600): _____, number of cores = _____, speedup = _____

OpenMP Parallel_{time} on local workstation for Mandelbrot run (-0.70, 0, 2.5, 600): _____, number of cores = _____, speedup = _____

TPL Parallel_{time} on local workstation for Mandelbrot run (-0.70, 0, 2.5, 600): _____, number of cores = _____, speedup = _____

2. Next, let's get a feeling for the overhead incurred by the client-server design, i.e. the file-based communication mechanism. From [section 5.4](#), the sequential time for the client-server version:

Sequential Client-Server_{time} on local workstation for Mandelbrot run (-0.70, 0, 2.5, 600): _____

This gives you a rough measure of the client-server overhead.

3. Open the VC++ or C# version of the provided VS solution: [Exercises\04 ClusterBased\ParallelClient-Server\ClientServerParMandelbrot](#) or [Exercises\04 ClusterBased\ParallelClient-Server.NET\ClientServerParMandelbrot.NET](#). The solution consists of 2 projects, the client-side GUI project and the server-side console project, and performs a parametric sweep as discussed in the previous section. Expand the client project, open "app.config", and change the configuration settings to match the correct values for your cluster. Run (F5) to confirm that all is well.

4. Switch to the release version, and if applicable, the target platform appropriate for your workstation (Win32 or x64):



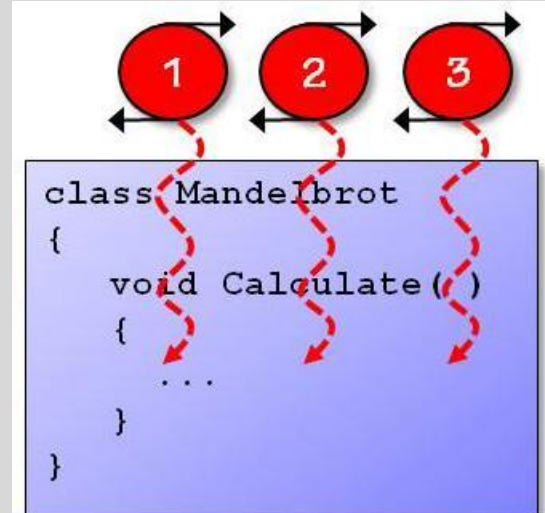
As provided, the app performs a parametric sweep across the cluster, creating N instances, one per core. Record the average execution time of 3 parallel runs:

Parametric Sweep (by core)_{time} on cluster for Mandelbrot run (-0.70, 0, 2.5, 600): _____, number of cores = _____, speedup = _____

To be fair, compute the speedup against the sequential time recorded in step 1 — i.e. against the original sequential Mandelbrot application, not the client-server version. The client-server design is an artifact of cluster-based parallelization, and so its overhead should be included in the cost of parallel execution. This is one reason speedups are often sub-linear in practice.

5. Now let's parallelize the application in other ways, and compare the results. Which technique will yield the fastest execution? For example, the current parametric sweep creates N server-side processes, one per core. Perhaps it is more efficient to create M processes, **one per node**, and then multi-thread each process to parallelize across the cores? Furthermore, is it more efficient to multi-thread using the Thread class, OpenMP, or TPL?

Or, if your compute nodes contain enough cores (4? 8? 16?), another approach is to forgo parametric sweep and use only multi-threading, running one server-side process across the cores of a **single node**.



The idea is that in some cases — e.g. short-lived parallel computations — it may be more efficient to run one multi-threaded process than M or N processes across the cluster (which incurs more startup overhead, network contention, etc.).

How do we determine which approach is best? More often than not, it is impossible to predict which parallelization strategy will be most effective, since it depends on a variety of factors, from hardware capabilities (memory, caching, clock rates) to problem characteristics (data set size, units of computation, and communication). In these cases we turn to brute-force experimentation.

6. As our first experiment, let's perform a parametric sweep across the cluster **by *node*** instead of by core. Modify the client-side Mandelbrot class ("Mandelbrot.cpp" or "Mandelbrot.cs") so that the job's *UnitType* is in terms of *Node* instead of *Core*. Likewise, the rest of the API-related code should work in terms of nodes, not cores. For example, job configuration:

```
job.UnitType = JobUnitType.Node;
.
.
.
int numNodes = nodes.Count;

job.MinimumNumberOfNodes = 1; // request 1..ALL nodes in the cluster:
job.MaximumNumberOfNodes = numNodes;
```

Task configurations should be updated similarly. [*Solutions are available in [Solutions\ParallelClient-Server\SweepByNode](#) and [Solutions\ParallelClient-Server.NET\SweepByNode](#).*] Run and time; does this version perform better or worse than the core-wise parametric sweep?

Parametric Sweep (by node)_{time} on cluster for Mandelbrot run (-0.70, 0, 2.5, 600): _____, number of nodes = _____, speedup = _____

In comparison to the core-wise sweep, the result should be slower (longer execution time and lower speedup) since we are running the app on fewer hardware resources. Use the Heat Map to confirm the app is running on just one core per node.

7. Next, experiment with either OpenMP (in the VC++ version) or the TPL (in the C# version). The idea is to multi-thread the server-side component, and then launch M instances of the server-side .EXE, one per node (which the client-side component currently does). Follow the same logic we did earlier in sections 4.4 (OpenMP) and 4.6 (TPL) when we parallelized the sequential app for execution on the client's workstation. In short, close all the editor windows in Visual Studio, expand the server-side project, and open the Mandelbrot class ("Mandelbrot.cpp" or "Mandelbrot.cs"). Locate the *Calculate* method, which executes sequentially. Now parallelize the outer for loop. [*Solutions are available in [Solutions\ParallelClient-Server\SweepWithOpenMP](#) and [Solutions\ParallelClient-Server.NET\SweepWithTPL](#).*] In the OpenMP version, think about whether to use static scheduling or dynamic scheduling... To visually see the impact of your decision (i.e. which threads generate which rows), modify the *Calculate* method to set the last 5 pixels of each row as follows:

```
int threadID = ((instanceNum - 1) * omp_get_num_threads()) + omp_get_thread_num() + 1;
for (int xp = pixels-5; xp < pixels; xp++)
    values[xp] = -threadID;
```

For the TPL version, recall that the TPL uses work-stealing (i.e. dynamic scheduling) to load-balance the iteration space. To visually see the impact of this approach (i.e. which threads generate which rows), modify the *Calculate* method to set the last 5 pixels of each row as follows:

```
int threadID = ((instanceNum - 1) * Environment.ProcessorCount) + Thread.CurrentThread.ManagedThreadId;
for (int xp = pixels - 5; xp < pixels; xp++)
    values[xp] = -threadID;
```

Run and time; how do these versions compare?

Parametric Sweep (OpenMP)_{time} on cluster for Mandelbrot run (-0.70, 0, 2.5, 600): _____, number of cores = _____, speedup = _____

Parametric Sweep (TPL)_{time} on cluster for Mandelbrot run (-0.70, 0, 2.5, 600): _____, number of cores = _____, speedup = _____

Since we are back to using all the hardware resources of our cluster (confirm via Heat Map?), these versions should run faster than the parametric sweep by node. The more interesting question is how these versions compare to the earlier parametric sweep by core?

8. Is OpenMP or the TPL adding significant overhead? Recall we started our discussion of parallel execution by [explicitly multi-threading](#) the app ourselves using the Thread class. Revisit this earlier discussion, and parallelize the server-side Calculate method explicitly. [*Solutions are available in [Solutions/ParallelClient-Server/SweepWithThread](#) and [Solutions/ParallelClient-Server.NET/SweepWithThread](#).*] Be careful, parallelizing the iteration space is more subtle in this version... Run and time, confirming via the Heat Map that you are using the entire cluster:

Parametric Sweep (Thread)_{time} on cluster for Mandelbrot run (-0.70, 0, 2.5, 600): _____, number of cores = _____, speedup = _____

This approach trades programmer productivity for hardware efficiency. So how does performance compare to your sweep by core? To your sweep by OpenMP / TPL?

9. Finally, let's experiment with explicit multi-threading across just one node of the cluster. This will eliminate any overhead associated with parametric sweep. Close all the editor windows in Visual Studio, and expand the client-side project. Open the client-side Mandelbrot class ("Mandelbrot.cpp" or "Mandelbrot.cs"), locate the Calculate method, and change the computation task from a parametric sweep to an ordinary task:

```
task.Name = "Mandelbrot Multi-threaded";
task.IsParametric = false;
task.MinimumNumberOfNodes = 1;
task.MaximumNumberOfNodes = 1;

String theArgs = String.Format("{0} {1} {2} {3} 1 1",
    _x,
    _y,
    _size,
    _pixels);

task.CommandLine = String.Format("{0} {1}", localexe, theArgs);
task.WorkDirectory = _basedir;
task.StdOutFilePath = "MandelbrotOut.1.txt";
task.StdErrFilePath = "MandelbrotErr.1.txt";
```

Run and time, confirming via the Heat Map that you are fully utilizing one node of the cluster:

Multi-threaded (Thread)_{time} on cluster for Mandelbrot run (-0.70, 0, 2.5, 600): _____, number of cores = _____, speedup = _____

Unless you have a large number of cores per node (8? 16? 32?), this approach should be slower than the others.

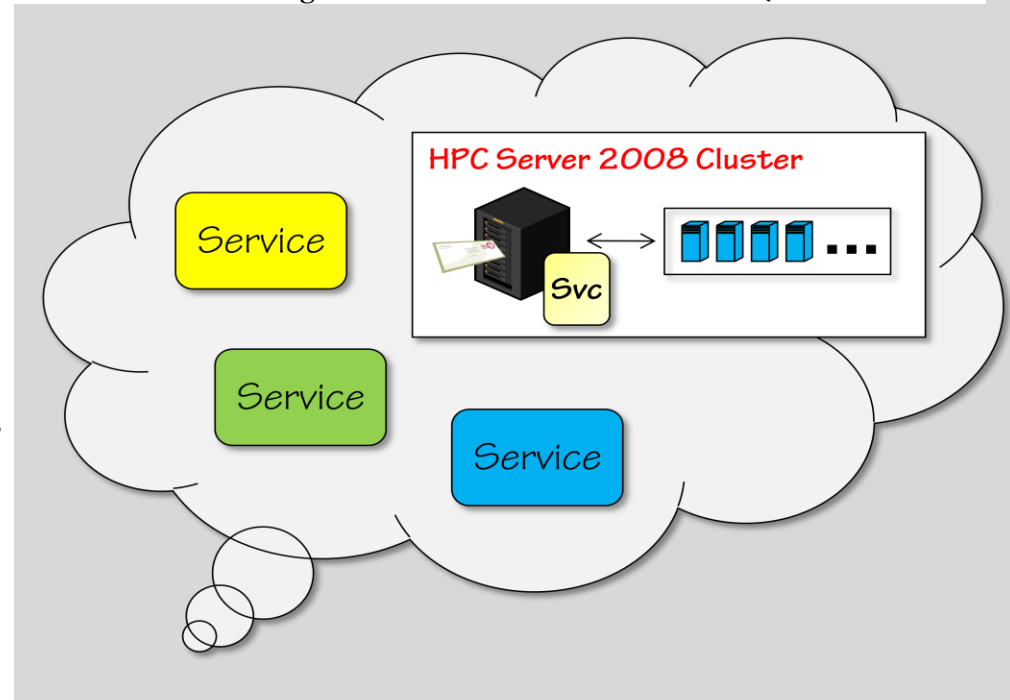
10. Okay, so which approach is best on your particular cluster? Why?

9. SOA-based Parallelization using WCF

Another perspective of cluster-based computing is that of high-performance *services* running in the *cloud*. In the world of SOA (*Service-Oriented Architectures*), Windows HPC Server 2008 can provide access to services running on your cluster. The obvious advantage is making high-performance computing **accessible** to a wider-range of your developers — HPC is now only a web service call away.

The more subtle advantages are two-fold. First, this provides an easier path for building **interactive** HPC applications, since the cluster behaves more like a web service than a batch system. Second, we have another technique for cluster-wide **parallelization**, since Windows HPC Server handles the details of running the service code across the cluster.

In short, we encapsulate our computation as a service-oriented .DLL, where each call to the service performs one unit of computation. The .DLL is installed on the compute nodes, and the service registered on a head-like node called a *broker* node. The client-side portion of the app calls the service using WCF (*Windows Communication Foundation*), achieving parallelism through multiple calls. The broker node is responsible for scheduling the calls across the cluster for execution.



9.1 Creating a WCF Service

As an example, let's create a simple WCF-based service for adding 2 numbers together. Startup Visual Studio 2008, and create a new project: Visual C#®, WCF, and then select the *WCF Service Library* template. Name the project "*ComputationServices*". Visual Studio creates a default service for you, with an interface ("*IService1.cs*") and an implementation ("*Service1.cs*"). Rename these files to "*IMathService.cs*" and "*MathService.cs*", respectively (answer "Yes" when asked if you want to update all project references).

Open the interface file "*IMathService.cs*", and delete the auto-generated contents of the namespace. Now define the contract for our service, which provides a single method for adding two integers and returning their sum:


```
[ServiceContract]
public interface IMathService
{
    [OperationContract]
    int Add(int a, int b);
}
```

Next, open the source code file “MathService.cs”, delete the auto-generated contents of the namespace, and implement the service. Define a class named *MathService* which implements *IMathService*, providing an implementation for the *Add* method:

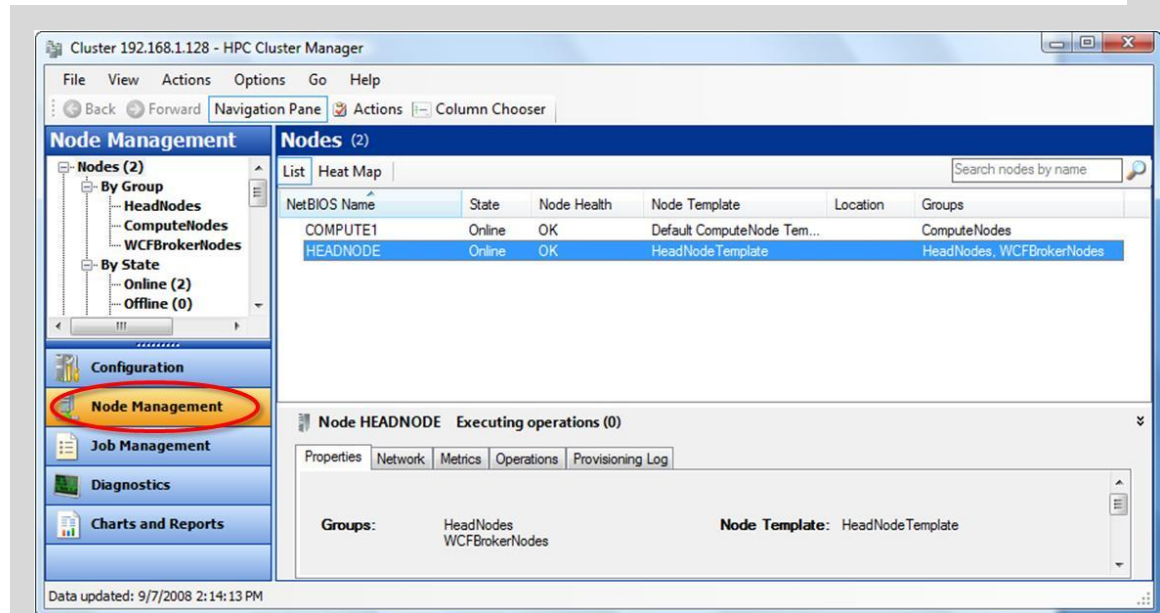
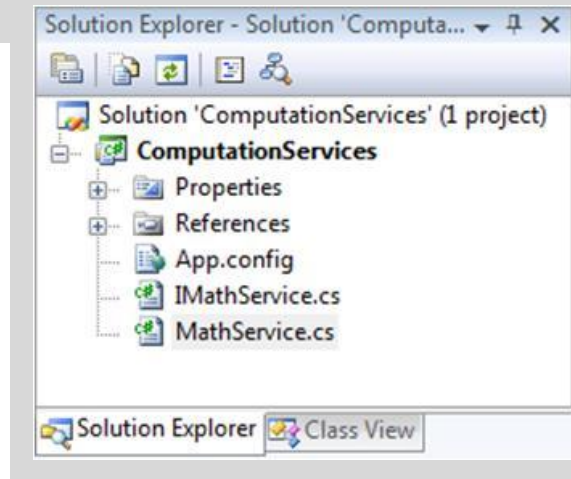
```
[ServiceBehavior(IncludeExceptionDetailInFaults = true)]
class MathService : IMathService
{
    public int Add(int a, int b)
    {
        return a + b;
    }
}
```

At this point the service is complete. Build a Release version, and locate the .DLL in *ComputationServices\ComputationServices\bin\Release*. Copy the .DLL to a folder that is easy to find, say C:\Temp.

9.2 Configuring the Broker Node

At least one of the nodes in the cluster must be set aside as a *WCF broker* node. Any node can serve as a broker node, as long as it's accessible on the public network. It is common for the head node to also serve as a broker node, or you can turn any compute node into a broker node. In Windows HPC Server 2008, the only restriction is that a node cannot serve as both compute and broker node, it's an either-or proposition.

To create a broker node in your cluster, you change a node's *role*. Startup the *HPC Pack Cluster Manager* utility, switch to the Node Management tab, right-click on the particular node, and “Take Offline”. Right-click again, and select “Change Role...”. In the window that appears, select the option for “WCF broker node”, and click OK. Finally, right-click again, and “Bring Online”. Once the node is online, it is now a broker node, ready to accept WCF-based service calls. To the right is a snapshot of my mini-cluster, with the head node also serving as a broker node.



9.3 Installing a WCF Service under Windows HPC Server 2008

Every service has 3 components: an interface, an implementation, and a configuration. We have already written and compiled the former, so let's look at the latter. A configuration file is needed to inform Windows HPC Server 2008 about the service, in particular specifying three things:

- the name and location of the service .DLL,
- the type name of the service interface in the format "Namespace.InterfaceName", and
- the type name of the service implementation in the format "Namespace.ClassName".

Here's the necessary configuration file for our example — the lines in **boldface** specify the service-specific information needed by Windows HPC Server:

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <configSections>
    <sectionGroup name="microsoft.Hpc.Session.ServiceRegistration"
      type="Microsoft.Hpc.Scheduler.Session.Configuration.ServiceRegistration,
        Microsoft.Hpc.Scheduler.Session,
        Version=2.0.0.0,
        Culture=neutral,
        PublicKeyToken=31bf3856ad364e35">
      <section name="service"
        type="Microsoft.Hpc.Scheduler.Session.Configuration.ServiceConfiguration,
          Microsoft.Hpc.Scheduler.Session,
          Version=2.0.0.0,
          Culture=neutral,
          PublicKeyToken=31bf3856ad364e35"
        allowDefinition="Everywhere"
        allowExeDefinition="MachineToApplication" />
      </sectionGroup>
    </configSections>
    <microsoft.Hpc.Session.ServiceRegistration>
      <service assembly="C:\Services\ComputationServices.dll"
        contract="ComputationServices.IMathService"
        type="ComputationServices.MathService" />
    </microsoft.Hpc.Session.ServiceRegistration>
  </configuration>
```

Copy the above text to file named "ComputationServices.config", and save in C:\Temp along with the .DLL.

To install the service on the cluster, first copy *ComputationServices.DLL* to the local folder *C:\Services* on every node⁸. Notice this matches the name and location given in the config file. Now copy *ComputationServices.config* to the local folder *C:\Program Files\Microsoft HPC Pack\ServiceRegistration* on every node. The service is now deployed and ready for use.

⁸ Recall that cluster-wide operations like file copying can be done from a console window using the HPC Server *clusrun* utility (see Section 6.1).

9.4 Calling the Service

In order to call the service, the client-side portion of the HPC application needs a *client-side proxy* — an object that understands the low-level details of communicating with the service. Once the proxy is in place, calling the service is as simple as instantiating the proxy and calling the method:

```
ComputationServicesProxy proxy = new ComputationServicesProxy();  
int sum = proxy.Add(1, 2);
```

The proxy can be automatically generated by .NET from the service .DLL. Start by opening a Visual Studio 2008 console window: Start, All Programs, Visual Studio 2008, Visual Studio Tools, *Visual Studio 2008 Command Prompt*. Navigate (cd) to C:\Temp (i.e. the folder containing ComputationServices.DLL), and run the following commands:

```
> svcutil ComputationServices.dll  
  
> svcutil *.wsdl *.xsd /async /language:C# /out:ComputationServicesProxy.cs
```

The end result is a source code file “ComputationServicesProxy.cs” containing a client-side proxy named “*MathServiceClient*”.

Now let’s call the service from a simple client-side GUI app. Create another project in Visual Studio 2008, this time a Windows Forms Application. Name the project “ComputationClient”. Create a simple UI with 3 textboxes and a command button. Add to the project the source code file “ComputationServicesProxy.cs” that we generated earlier (it should be located in C:\Temp). Now add the following 4 project references:

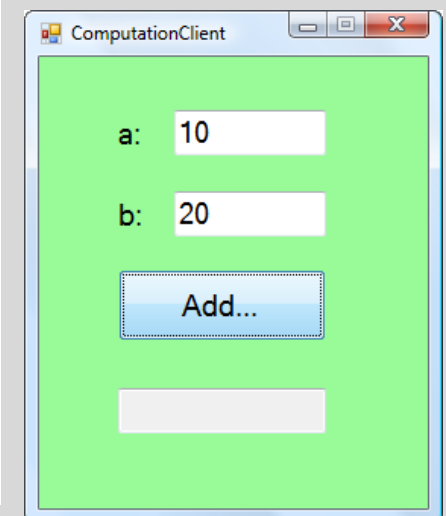
- System.ServiceModel.dll
- Microsoft.Hpc.Scheduler.dll
- Microsoft.Hpc.Scheduler.Properties.dll
- Microsoft.Hpc.Scheduler.Session.dll

The latter 3 are shipped as part of the Windows HPC Server 2008 SDK, and can be found in C:\Program Files\Microsoft HPC Pack 2008 SDK\bin. [SDK not installed? See [Section 7.1](#).]

The client is now ready to call the service. View the form in Design mode, and double-click the button to generate a Click event handler. Scroll to the top of the source code file, and import the following namespaces:

```
using System.ServiceModel;  
using Microsoft.Hpc.Scheduler;  
using Microsoft.Hpc.Scheduler.Properties;  
using Microsoft.Hpc.Scheduler.Session;
```

Back in the Click event handler, start by defining the appropriate parameters for your cluster:



```

private void button1_Click(object sender, EventArgs e)
{
    //
    // parameters for our cluster:
    //
    string broker = "headnode";
    string service = "ComputationServices";

    SessionStartInfo info = new SessionStartInfo(broker, service);
    info.Username = @"minihpc\hummel";
    info.Password = null; // prompt for pwd:

    info.ResourceUnitType = JobUnitType.Core;
    info.MinimumUnits = 1;
    info.MaximumUnits = 4;

    Session.SetInterfaceMode(false /*GUI*/, (IntPtr)null /*no parent*/);

    Session session = Session.CreateSession(info); // connect to broker node, start service running:

    MathServiceClient proxy = new MathServiceClient( new NetTcpBinding(SecurityMode.Transport, false),
        session.EndpointReference);

    int a = Convert.ToInt32(textBox1.Text);
    int b = Convert.ToInt32(textBox2.Text);

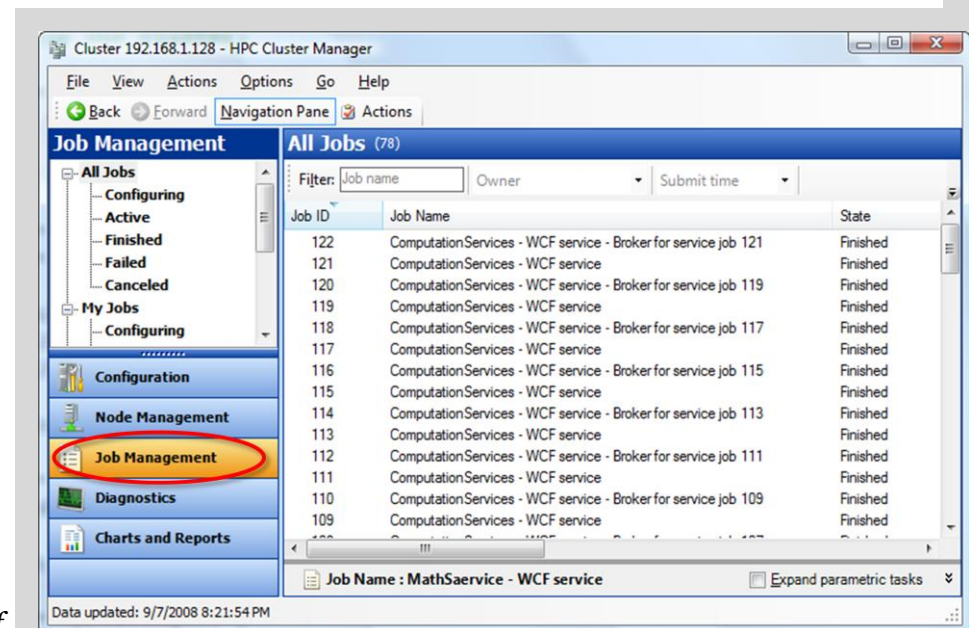
    int sum = proxy.Add(a, b); // call service:

    textBox3.Text = sum.ToString();

    proxy.Close();
    session.Dispose();
}

```

That's it! Build the application, and run... When you click the *Add* button, be prepared to wait 5-10 seconds — this is the overhead required by Windows HPC Server to acquire the necessary resources, load the service, and start it running (more on this in a moment). If the application fails, note that the service may still be running in Windows HPC Server, causing problems if you try to run the client again. The solution is to startup HPC Cluster or Job Manager, view the list of running jobs, and cancel any “ComputationServices” jobs that are running (right-click, “Cancel Job”). [If



problems persist, compare your work to the solution available in [Solutions\SOA](#). For example, [Solutions\SOA\Binaries](#) contains the correct .config file for installation on the cluster.]

9.5 Behind the Scenes

Each time you start a WCF-based session (*Session.CreateSession*), Windows HPC Server creates and schedules a number of jobs on the cluster. These jobs reserve the necessary resources for service execution, allowing calls to run quickly. If you haven't already, open HPC Cluster or Job Manager, and view the list of finished jobs — you should see a number of jobs containing the service name "ComputationServices".

To identify exactly where time is being spent in the code, modify the Click handler to display a message such as "starting", "creating", or "calling" in the result textbox:

```
textBox3.Text = "starting...";
textBox3.Refresh();
Session session = Session.CreateSession(info);

textBox3.Text = "creating...";
textBox3.Refresh();
MathServiceClient proxy = new MathServiceClient( new NetTcpBinding(SecurityMode.Transport, false),
    session.EndpointReference);

int a = Convert.ToInt32(textBox1.Text);
int b = Convert.ToInt32(textBox2.Text);

textBox3.Text = "calling...";
textBox3.Refresh();
int sum = proxy.Add(a, b);
```

Run the app, and watch the textbox closely. What did you discover? Given that the computation itself — adding 2 numbers — is so simple, the vast majority of time is devoted to starting the service via *Session.CreateSession*.

Is this overhead a problem? Generally, no. In most realistic HPC applications, any startup cost is amortized across hundreds and thousands of calls, calls that routinely take minutes, hours or even days to complete. Startup costs becomes negligible in these scenarios.

9.6 Amortizing Overhead and Executing in Parallel

In the previous example, notice that we start a new session every time the user clicks the Add button. An obvious way to amortize the startup cost is to create the session and proxy exactly once, e.g. during client startup. Likewise, we close the proxy and session during client shutdown. This is easily done during the *Load* and *FormClosed* events of a Windows Form application. Open the Visual Studio solution provided in [Solutions\SOA\ComputationParClient](#), view the code for Form1, and notice how the logic is spread across the *Load*, *Click*, and *FormClosed* event handlers.

More interestingly, let's revisit the code that calls the service — `proxy.Add(a, b)`. Here's the exact line from the Click handler:

```
int sum = proxy.Add(a, b);
```

What if the client had to perform 4 additions, and wanted to perform them in parallel? You might be tempted to try the following:

```
int sum1 = proxy.Add(a, b);
int sum2 = proxy.Add(c, d);
int sum3 = proxy.Add(e, f);
int sum4 = proxy.Add(g, h);
```

Would the calls execute in parallel, or sequentially one after another? The answer depends on how the client-side proxy is implemented... By default, calls to the proxy behave like a traditional method call, meaning the caller waits until a value is returned. Thus the meaning of

```
int sum1 = proxy.Add(a, b);
```

is predictable — the caller waits until the sum is returned. This implies that a series of such calls is sequential. How then do we take advantage of the cluster for parallel execution?

To execute a computation in parallel, the client makes a call to *start* the execution, but does not wait for it to finish. Instead, the client sets up a mechanism to be notified when the computation completes, and performs other work in the meantime. For example, to start an addition, we call the proxy's *BeginAdd* method, providing a *callback* method for notification:

```
proxy.BeginAdd(a, b, this.CompletionCallback, null /*no additional state necessary*/);
.
. // perform other work IN PARALLEL with service call:
.
```

The callback method is invoked automatically when execute completes. It is passed a parameter from which the result can be retrieved via the proxy's *EndAdd* method:

```
private void CompletionCallback(IAsyncResult result)
{
    int sum = proxy.EndAdd(result); // harvest result:
    Console.WriteLine(sum);
}
```

This approach forms the basis for parallel execution of WCF calls across the cluster.

9.7 Design Considerations

Each call to the service runs on one core/socket/node of the cluster, so a call is your unit of work — and your unit of parallelism. Too small, and the overhead of the call may dominate the computation, limiting your performance gains. Too large, and you have fewer calls executing in parallel, hindering both application performance and its interactive feel. The best approach is to parameterize your service as much as possible, and then experiment to identify the proper *granularity* on your cluster.

Each call will execute in parallel using the proxy's *Begin...* and *End...* methods. Think carefully about how you will aggregate the results harvested in the callback method. For example, as discussed in [Section 3.3](#), the only thread that may touch the UI is the main thread. Yet callbacks occur on worker threads which may not touch the UI. How then to display the results? [*One solution is to define events on the UI for display*

purposes. The worker threads raise these events, and .NET context switches to the main thread to handle the events. This is the approach taken by the [BackgroundWorker](#) class, which is used quite effectively in the Mandelbrot application.] Or perhaps you plan to collect the results in a shared data structure, where the worker threads simply add to the data structure on each callback. Be sure to control access to the data structure, otherwise you'll have a race condition, and ultimately, incorrect results. [One solution is to use C#'s lock construct.]

9.8 Lab Exercise!

As our final lab exercise, let's parallelize the C# version of the Mandelbrot application using Windows HPC Server's WCF-based capabilities. A solution to this exercise is available in [Solutions\WCF\WCFMandelbrot](#).

1. We'll start from a sequential, WCF-based version of the Mandelbrot app. The compiled version is available in [Solutions\WCF\Binaries](#). Open the Release sub-folder, and you'll find 4 files:

- WCFMandelbrotClient.exe: client-side application
- WCFMandelbrotClient.exe.config: configuration file for client-side app
- WCFMandelbrotService.dll: service .DLL
- WCFMandelbrotService.config: Windows HPC Server configuration file for service

As discussed [earlier](#), deploy the service .DLL and .config files to the cluster. Review the cluster parameters in "WCFMandelbrotClient.exe.config", and adjust based on your cluster:

ServiceName	name of service (WCFMandelbrotService)
ClusterBrokerNode	machine name or IP address of broker node
RunAsUserName	run-as credentials for running programs on cluster — username
RunAsPassword	run-as credentials for running programs on cluster — password

If your cluster does not have a broker node, you'll need to configure one as discussed in [Section 9.2](#).

2. Run the client-side app "WCFMandelbrot.exe", and generate a Mandelbrot image. Performance should be slow, since the WCF-based calls to the cluster are executing sequentially. Record the average execution time:

Sequential WCF-based_{time} on cluster for Mandelbrot run (-0.70, 0, 2.5, 600): _____

Confirm the sequential execution by watching the Heat Map in Cluster Manager — exactly one core should be executing.

3. Open the sequential implementation provided in [Exercises\05 WCF\WCFMandelbrot](#). First, review the service code. Expand the WCFMandelbrotService project, and open the service interface in "IMandelbrotService.cs". The service provides one method that generates the Mandelbrot image for exactly one row *yp*:

```
[ServiceContract]
public interface IMandelbrotService
{
    [OperationContract]
    int[] GenerateMandelbrotRow(double yp, double y, double x, double size, int pixels);
}
```

```
}
```

Notice the pixel values (an integer array) are returned as the result of the call. The implementation (see “MandelbrotService.cs”) is identical to previous versions of the application. Generating one row per call strikes a reasonable balance of simplicity vs. granularity — generating a single pixel is too small, while generating multiple rows offers more flexibility but complicates parameter passing.

4. Now let’s focus on the client-side application. Close any editor windows, and expand the “WCFMandelbrotClient” project. Open “app.config” and configure to match your cluster’s configuration. Notice the presence of the client-side proxy “MandelbrotServiceProxy.cs”, which was auto-generated as discussed [earlier](#). Now open the source code for the Mandelbrot class, “Mandelbrot.cs”.

The main form creates one instance of the Mandelbrot class the first time the “Go” button is clicked. This instance is used over and over again until the form is closed, in which case the Mandelbrot’s *Close()* method is called. Locate the constructor, and notice it starts the service on the cluster and creates the proxy:

```
try
{
    string broker = ConfigurationSettings.AppSettings["ClusterBrokernode"];
    string service = ConfigurationSettings.AppSettings["ServiceName"];
    .
    .
    .

    _session = Session.CreateSession(...);

    _proxy = new MandelbrotServiceClient(...);
}
```

This design amortizes the startup cost across the lifetime of the application. The *Close()* method shuts down the proxy and service:

```
try
{
    _proxy.Close();
    _session.Dispose();
}
```

Go ahead and run the application (F5). Click the “Go” button, and notice you are prompted to login to the cluster. There’s the usual 5-10 second delay, and then the image will start appearing. When it finishes, click the “Go” button again, and now the image appears almost immediately since the service is already up and running.

5. Locate the Mandelbrot’s *Calculate* method, and confirm that the service calls are being made sequentially:

```
int[] values = _proxy.GenerateMandelbrotRow(yp, _y, _x, _size, _pixels);
```

For parallel execution, we need to use the proxy’s *Begin...* and *End...* methods. So replace the sequential call with the following:

```
_proxy.BeginGenerateMandelbrotRow(yp, _y, _x, _size, _pixels, this.MandelbrotRowCallback, yp /*state*/);
```

In this case we supply both a callback method and some state — the row number — to accompany call. This way we’ll know which row was generated when the call completes. Now add the following callback method to the Mandelbrot class:

```
private void MandelbrotRowCallback(IAsyncResult result)
{
    //
    // did user close the form and shutdown the app prematurely? In that case,
    // no more results are coming through:
    //
    if (_proxy.State == CommunicationState.Closed)
        return;

    //
    // else proxy is still up and running, and so results should be available:
    //
    int yp = (int) result.AsyncState; // retrieve which row this represents:

    int[] values = _proxy.EndGenerateMandelbrotRow(result); // now harvest results:

    //
    // we've generated a row, report this as progress for display:
    //
    Object[] args = new Object[2];

    args[0] = values;
    args[1] = AppDomain.GetCurrentThreadId();

    _worker.ReportProgress(yp, args);
}
```

The callback retrieves the row number from the associated state, and then retrieves the generated pixels from the service. These values are reported to the UI as before.

6. Okay, let’s give it a try! Run the app and see what happens... Oops. “*Unable to proceed because something is closed...*” What happened? The worker thread is executing the Calculate method, which *starts* 600 or so service calls on the cluster. However, we don’t *wait* for these calls to complete, which means the Calculate method completes and returns. This has the effect of raising the worker’s *RunWorkCompleted* event, and then closing the worker object. Meanwhile, as the service calls complete and trigger callbacks, the last line of *MandelbrotRowCallback* fails:

```
_worker.ReportProgress(yp, args);
```

We need to design a solution where the Calculate method waits until all service calls complete. The simplest approach is a shared variable that is incremented each time a service call is started, and decremented each time a service call completes. After all calls are started, if we then wait until

the count reaches 0, we'll have waited until all calls have completed.

At the top of the Mandelbrot class, add a long integer field named `_asyncCallsOutstanding`:

```
private long _asyncCallsOutstanding = 0;
```

Now, each time a service call is started, increment this variable:

```
System.Threading.Interlocked.Increment(ref _asyncCallsOutstanding);  
_proxy.BeginGenerateMandelbrotRow(yp, _y, _x, _size, _pixels, this.MandelbrotRowCallback, yp /*state*/);
```

Notice that we use the Threading library's *interlocked increment* function, which performs an atomic update — in essence the variable is locked, updated, and then unlocked. This eliminates race conditions due to concurrent access... So who else is accessing this variable? The callback method, each time a row is returned! Modify the callback to decrement the count after processing:

```
_worker.ReportProgress(yp, args);  
System.Threading.Interlocked.Decrement(ref _asyncCallsOutstanding);
```

The last step is to wait until all the service calls have completed. The simplest approach is a loop that waits until the count reaches 0:

```
for (int yp = 0; yp < _pixels; yp++)  
{  
    .  
    .  
    .  
}  
  
while (System.Threading.Interlocked.Read(ref _asyncCallsOutstanding) > 0)  
    System.Threading.Thread.Sleep(200);
```

Run and test — the application should execute correctly, and with much better performance. Interestingly, note the service has not changed at all in our move from sequential to parallel execution. Given a properly-designed service, parallelization is a client-side decision.

7. How much faster does the parallel version run? How efficient is WCF-based parallelization? Record the average execution time:

Parallel WCF-based_{time} on cluster for Mandelbrot run (-0.70, 0, 2.5, 600): _____, number of cores = _____, speedup = _____

That's it, excellent work!

10. Conclusions

Windows HPC Server 2008 provides a powerful framework for developing and executing high-performance applications. This tutorial presented a common HPC development scenario: the sequential C++ or C# developer looking for performance gains through parallel processing. We presented a number of techniques, including explicit parallelization using the .NET Thread class, as well as implicit parallelization using OpenMP and the TPL. We also demonstrated how to communicate with Windows HPC Server 2008, enabling cluster-wide parallelization via both parametric sweep and WCF.

Other tutorials are available on Windows HPC Server 2008, including scenarios for the [classic HPC developer](#), the [enterprise developer](#), and those [new to HPC](#). The following are also good references and resources on HPC. Cheers!

10.1 References

Windows HPC Server 2008 homepage: <http://www.microsoft.com/HPC/>

Windows HPC Server 2008 resources, blogs, forums: <http://windowshpc.net/Pages/Default.aspx>

OpenMP: <http://openmp.org/>

TPL and PFX: <http://msdn.microsoft.com/en-us/concurrency/default.aspx>

General HPC news: <http://www.hpcwire.com/>

10.2 Resources

“Multi-Core Programming”, by S. Akhter and J. Roberts (Intel Press)

“Patterns for Parallel Programming”, by T. Mattson, B. Sanders and B. Massingill

“Parallel Programming in C with MPI and OpenMP”, by M. Quinn

“Parallel Performance: Optimize Managed Code for Multi-Core Machines”, by D. Leijen and J. Hall, MSDN Magazine, Oct 2007. Online at <http://msdn.microsoft.com/en-us/magazine/cc163340.aspx>

Appendix A: Summary of Cluster and Developer Setup for Windows HPC Server 2008

This appendix serves as a brief summary of how to setup an Windows HPC Server 2008 cluster. It outlines the main software components you need, and the major steps to perform. I use this summary when setting up small personal clusters; this summary is not appropriate for setting up large, production-ready clusters.

Hardware requirements

One or more 64-bit capable machines; network interconnect.

Software requirements

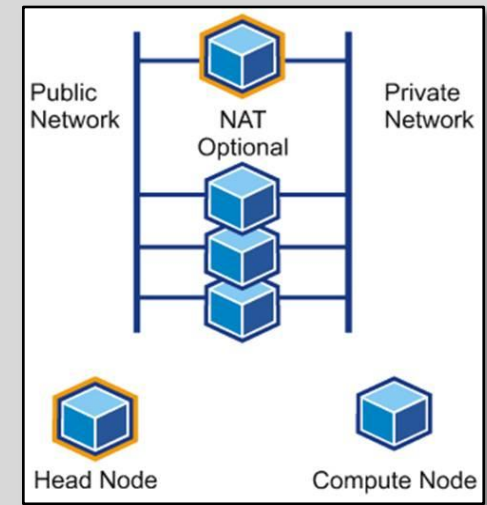
Windows Server® 2008 64-bit, standard or enterprise edition. Microsoft HPC Pack 2008. Developers will need Visual Studio 2008, SDK for Microsoft HPC Pack 2008, and Windows PowerShell. Download the SDK from <http://go.microsoft.com/fwlink/?linkID=127031>.

Developer Workstation Setup

Install Visual Studio 2008 (full install), Windows PowerShell, Microsoft HPC Pack 2008 (client-side utilities), and SDK for Microsoft HPC Pack 2008. Note that unlike cluster nodes, a developer machine can be running 32-bit or 64-bit Windows. In the 32-bit case, you build and test locally in 32-bit, then rebuild as 64-bit and deploy to cluster.

Cluster Setup

1. Install Windows Server 2008 64-bit on each machine. Assign admin pwd, enable remote desktop. Activate Windows if necessary via Control Panel System applet. Add roles: Web Server, App Server (in particular ASP.NET). Windows update.
2. Install latest release of .NET framework (<http://msdn.microsoft.com/en-us/netframework/aa569263.aspx>), and Visual C++ runtimes on each machine. At the very least, you want to install 64-bit release runtimes for Visual Studio 2008. Download here: <http://www.microsoft.com/downloads/details.aspx?familyid=bd2a6171-e2d6-4230-b809-9a8d7548c1b6&displaylang=en>. Now ask yourself, do you plan to run debug versions on the cluster (e.g. to do remote debugging)? 32-bit release / debug versions? Apps built with Visual Studio 2005? If you answered yes to any of these, you have more runtimes to install. I've written a more detailed set of instructions on what you need, where to get it, and how to install it. See [Misc\VC++ Runtimes\Readme.docx](#). The release runtimes are easy; the debug runtimes are not.
3. Decide which machine is going to be the head node, when in doubt pick the one with the largest capacity hard disk. The head node can also act as a compute node, which is typical for small clusters. Have this machine join an existing domain, or if not, add role: Active Directory® Domain Services. Create the following groups in the domain: HPCUsers, HPCAdmins. Create the following users in the domain: hpcuser, hpcadmin. Add both users to HPCUsers group, add hpcadmin to HPCAdmins group. Add yourself and others to these groups as appropriate.
4. On head node, create directories C:\Apps and C:\Public. Give everyone read access, give HPCUsers and HPCAdmins full access. Now share both C:\Apps and C:\Public, giving everyone read access, HPCUsers and HPCAdmins full access.
5. On remaining nodes of cluster, join domain. Create directory C:\Apps. Give everyone read access, give HPCUsers and HPCAdmins full access. Now share C:\Apps, giving everyone read access, HPCUsers and HPCAdmins full access.



6. Back on head node, install Microsoft HPC Pack 2008 to setup a new cluster. Follow step-by-step “To do” list to configure network, firewalls, etc. When it comes to “node template”, create an empty template since we will configure the compute nodes manually. Add HPCUsers as a cluster user, add HPCAdmins as a cluster admin. If necessary, change head node role to include acting as a compute node. Bring head node online via Node Management tab, right-click. When you are done, make sure firewall is off (if you have exactly one NIC then it has to be off, if you have multiple NICs then make sure it’s off for the private network MPI will use).
7. On remaining nodes, install Microsoft HPC Pack 2008 to join an existing cluster. When you are done, make sure firewall is off (if you have exactly one NIC then it has to be off, if you have multiple NICs then make sure it’s off for the private network MPI will use).
8. Back on head node, startup the Microsoft HPC Pack Cluster Manager MMC plug-in, and select Node Management tab. For each of the remaining nodes, right-click and apply empty node template. Then right-click and bring online. When you’re done, the cluster is ready for testing! Run the pingpong diagnostic from the HPC Cluster Manager, which sends messages between every pair of execution cores in the cluster. This may take a few minutes (to a few hours based on the size and complexity of the cluster).
9. If you plan to develop software on any of the nodes (e.g. sometimes you might want to recompile apps on the head node), install Visual Studio 2008 and the SDK for Microsoft HPC Pack 2008.
10. If you want to enable remote desktop connections for non-administrative users, add *HPCUsers* and *HPCAdmins* as members of the “Remote Desktop Users” group. Also, make sure that “Remote Desktop Users” have been granted access through Terminal Services: Security Settings, Local Policies, User Rights Assignment, Allow log on through Terminal Services. Repeat on each node you want to support remote access.
11. If you want to enable ETW tracing for non-administrative users, add *HPCUsers* and *HPCAdmins* as members of the “Performance Log Users” group. Repeat on each node you want to support tracing.
12. If you want to enable remote debugging on the cluster, install Visual Studio’s Remote Debugger on each compute node. Locate your Visual Studio 2008 install disk, and copy *Remote Debugger\amd64\rdbgsetup.exe* to the cluster’s Public share. For each compute node, including the head node if it also serves as a compute node, do the following: (1) run *rdbgsetup.exe*, and (2) copy the contents of *C:\Program Files\Microsoft Visual Studio 9.0\Common7\UDE\Remote Debugging\amd64* to *C:\RDB*. Note that you can use a different target directory for step 2, but make sure the path does not contain spaces, and use the same path on each node.
13. If you plan to support MPI.NET on the cluster, install the MPI.NET runtime on each compute node: download from <http://osl.iu.edu/research/mpi.net/software/>.

Shutdown

When shutting down the cluster, first shutdown the compute nodes via the remote desktop and then shutdown the head node.

Startup

When booting the cluster, boot the head node first, give it 5+ minutes (especially if it’s a domain controller), then boot the compute nodes. Check (via HPC Cluster Manager) to make sure each machine is online. Finally, check the head node and one of the compute nodes (via remote desktop) to make sure the firewall is turned off (if you have exactly one NIC then it has to be off, if you have multiple NICs then make sure it’s off for the private network MPI will use).

Appendix B: Troubleshooting Windows HPC Server 2008 Job Execution

Are you submitting a job and getting nothing in return? Here's what I do to troubleshoot execution failures on the cluster. The symptoms range from submitting a job that never runs (e.g. remains queued forever) to error messages such as "execution failed".

Step 1: It's a Local Problem

The best way to troubleshoot a failure of any kind is to deploy the app as you normally would, and then remote desktop into one of the compute nodes, open a console window (Start, cmd.exe), and run the .EXE directly. If you deployed locally, navigate to the working directory you specified in the job configuration, and type "appname.exe" to run it. If you deployed to a network share and set the network share as your working directory, then run by concatenating the network share and the .EXE name, e.g. "\\headnode\public\appname.exe". What happens? If the application failed to start, the most common problem with VC++ is that you deployed a debug version of the app, and the debug runtimes are not installed (or you deployed a 32-bit app and the 32-bit runtimes are not installed, etc.). The simplest fix in this case is to build a 64-bit release version of your application, redeploy, and test this version of the .EXE. The other fix is to deploy the necessary runtimes to the cluster; see Appendix A on [cluster setup](#) (step 2). For .NET applications, make sure the latest release of .NET is installed; e.g. MPI.NET requires .NET 3.5 (see <http://msdn.microsoft.com/en-us/netframework/aa569263.aspx>). Also, are you trying to run a .NET app from a network share or other remote location? Note that by default, .NET is configured for security reasons to prevent the execution of non-local code; if you get a security exception when you run the .EXE, this is most likely the problem. The solution is to deploy the .EXE locally to each node, or configure .NET on each node to make the public network share a trusted location.

Once the .EXE at least loads and starts, type the command-line as given in the task that failed (e.g. "mpiexec MPIApp.exe ..."). Now what happens? The most common problems include forgetting to specify mpiexec at the start of an MPI run, misspelling the share or application name, omitting required command-line arguments to the app, or missing input files. Also, make sure you are specifying a working directory in your task configuration — this might explain why the .EXE and input files cannot be found. Once the application runs locally on a compute node, resubmit the job and see what happens.

Are you trying to run an MPI application, and mpiexec just hangs? In other words, you correctly type "mpiexec MPIApp.exe ...", but execution just hangs? I've seen this problem if both Microsoft® Compute Cluster Pack and Microsoft HPC Pack are installed. In particular, echo the PATH environment variable from a console window ("echo %path%") and see if it contains "C:\Program Files\Microsoft Compute Cluster Pack\...". If so, you need to delete any references to WCCS from the PATH variable (Control Panel, System, Advanced Settings, Environment Variables). Close the console window, reopen, and try running again.

Step 2: It's a Global Problem

Assuming the app runs locally on a compute node, the next most common problem is a global one, typically firewall or security. Security problems appear as jobs that fail immediately, or jobs that queue but never finish. Firewall problems typically result in MPI failures along the lines of "unable to open connection".

In terms of security, keep in mind there are two distinct logins needed: one to connect to the head node for submitting a job, and one for job execution ("run-as"). When you connect to the head node to submit a job, you are using the credentials of your personal login on your local workstation; if you were able to open the HPC Job Manager and connect to the head node, you have the proper permissions to use the cluster. The more common problem are the "run-as" credentials. You are prompted for these credentials when you submit a job. Does this login have permission to access the .EXE? Access the input files? Execute the .EXE? Write the output file(s)? The simplest way to find out is to use a known administrator account for the run-as credentials. Start by clearing the credential cache on the machine you have been submitting jobs from: open

the HPC Job Manager, Options menu, and select *Clear Credential Cache*. This will ensure that you are prompted for run-as credentials the next time you submit a job. Now submit your job once again, entering a known administrator login and password for the cluster when prompted. If the job runs, you know it's a security problem related to the login you were trying to use.

To collect more data about what the security problem might be, remote desktop into one of the compute nodes and login with the credentials you are specifying for the run-as credentials. Can you login? Try to run the application. Do you have permission? If all this works, the last step is to repeat this exercise on the head node. If the head node is a domain controller, note that logins need explicit permission to login, and this is most likely the source of the problem.

If security does not appear to be the cause of the problem, are you trying to execute MPI programs? In this case, make sure the firewall is off on all the nodes: if nodes have exactly one NIC then it has to be off, if they have multiple NICs then make sure it's off for the private network MPI will use. A quick way to determine firewall status on the cluster is to open the Microsoft HPC Pack Cluster Manager MMC plug-in and run the diagnostic *FirewallConfigurationReport*. This will take a few seconds, then you can view the completed test results to see the status of the firewall on every node. If you are not sure whether this is the problem, turn the firewall completely off on 2 nodes, and resubmit the job configured to run on just those nodes. If the job now runs, it's a firewall problem; fix the other machines to match the firewall configuration of the working machines.

If you are running MPI programs and absolutely require compute nodes to have their firewalls on, then the next best solution is to limit MPI to a range of ports, and configure your firewall software to open these ports. You specify the range of MPI ports via the environment variable *MPICH_PORT_RANGE* in the format min,max (e.g. 9500,9999). You then configure the firewall to open this range of ports. Repeat on each compute node. The alternative is to grant each individual MPI program full access to the network on each compute node.

Step 3: Job Submission Tweaks

Make sure you are specifying a working directory in your task configuration, otherwise the .EXE and input files may not be found. Likewise, be sure to redirect stdout and stderr to capture error messages you might be missing. Are you resubmitting jobs from a template? Note that if you change the number of processors for the job (such as reducing the maximum number of processes requested), you also need to adjust the number of processors requested for the individual tasks — changing job configuration parameters does not necessarily update the underlying task configurations. For example, you might change a job to request a max of 4 processors, but one of the tasks is still requesting 5..8 processors, so the job never runs. When in doubt, submit a new, simpler job from scratch.

Step 4: Okay, I'm Really Stuck

If you've come this far, then you're really stuck, and the problem is not one of the more common ones. So now what? First, abandon the program you are trying to run, and run a much simpler application — try to submit and run a trivial, sequential “Hello world” type of application. If you need such an app, use *Misc\VC++ Runtimes\VS2008\Hello-64r.exe*. Start the troubleshooting process over with this trivial application. Second, try running one of the diagnostics in the HPC Cluster Manager, such as pingpong. This can identify nodes that are not working properly, at least with respect to the network infrastructure. Third, try Microsoft's HPC Community forums, you might find a solution listed there. Or worst-case, post your problem as a new thread and see what unfolds. Start here: <http://www.windowshpc.net/Pages/Default.aspx> . The forums can be found here: <http://forums.microsoft.com/WindowsHPC/default.aspx> .

Good luck!

Appendix C: Screen Snapshots

HPC Cluster Manager / HPC Job Manager : Create New Job dialog

Create New Job

Job Details

Job name:

Job template:

Project:

Priority:

Job run options

☐ Do not run this job for more than:

Days: Hours: Minutes:

☐ Run job until cancelled or run time expires

☐ Fail the job if any task in the job fails

Job resources

Select the type of resource to request for this job:

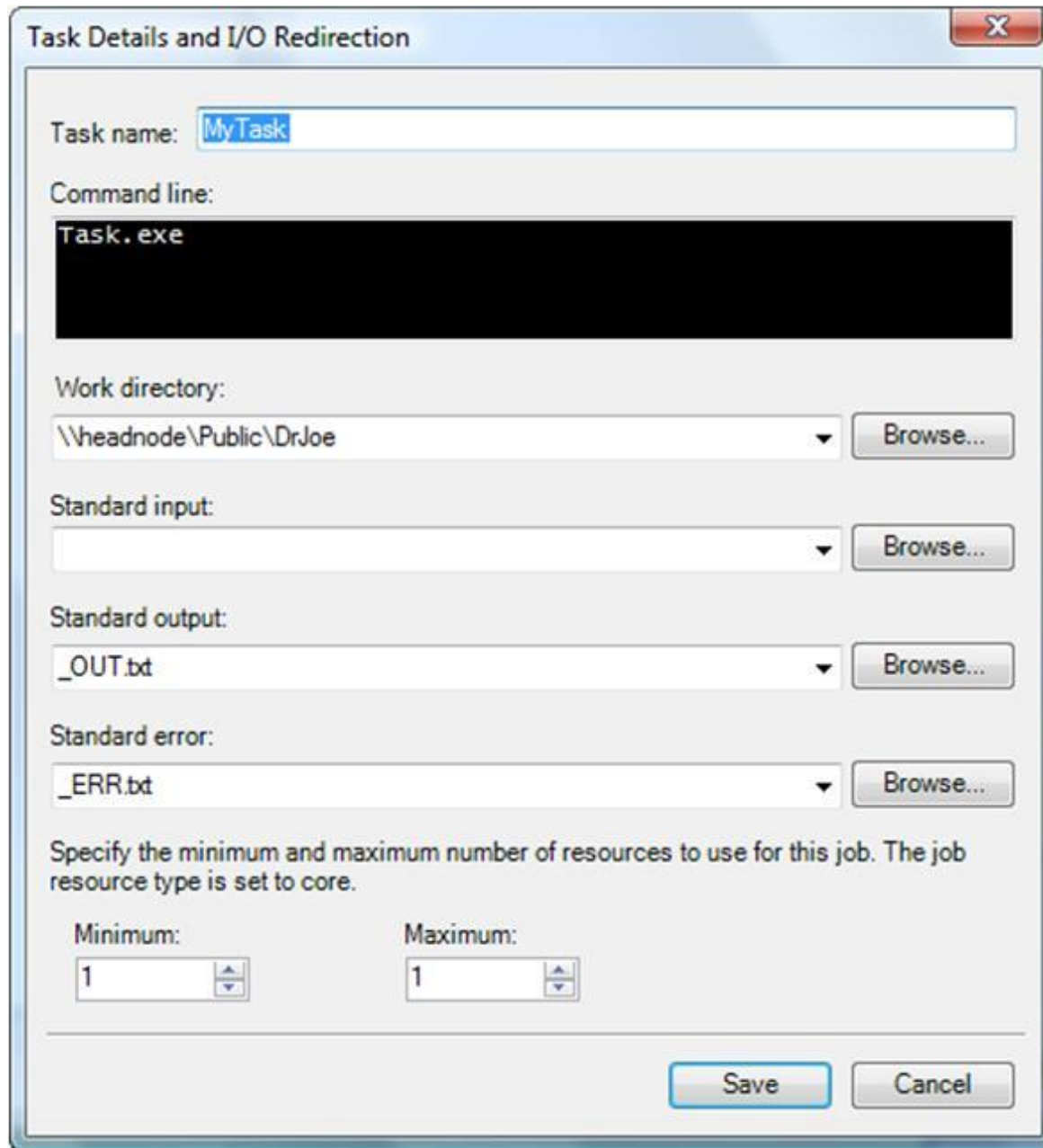
Enter the minimum and/or maximum of the selected resource type that this job is allowed to use:

Minimum: ☒ Auto calculate ☐

Maximum: ☒ Auto calculate ☐

☐ Use assigned resources exclusively for this job

No other jobs will be allowed to run on the selected nodes while the job is running.



The dialog box is titled "Task Details and I/O Redirection" and contains the following fields and controls:

- Task name:** A text box containing "MyTask".
- Command line:** A text area containing "Task.exe".
- Work directory:** A dropdown menu showing "\\headnode\\Public\\DrJoe" and a "Browse..." button.
- Standard input:** A dropdown menu (empty) and a "Browse..." button.
- Standard output:** A dropdown menu showing "_OUT.txt" and a "Browse..." button.
- Standard error:** A dropdown menu showing "_ERR.txt" and a "Browse..." button.
- Resource specification:** A section with the text "Specify the minimum and maximum number of resources to use for this job. The job resource type is set to core." followed by "Minimum:" and "Maximum:" labels. Each label is next to a spinner box currently set to "1".
- Buttons:** "Save" and "Cancel" buttons at the bottom right.

Feedback

Did you find problems with this tutorial? Do you have suggestions that would improve it? Send us your feedback or report a bug on the Microsoft HPC developer [forum](#).

More Information and Downloads

Allinea DDTLite	http://www.allinea.com/
Intel Cluster Debugger	http://www.intel.com/cd/software/products/asmo-na/eng/index.htm
Portland Group Cluster Kit	http://www.pggroup.com/products/cdkindex.htm
MPI.NET	http://osl.iu.edu/research/mpi.net/software/
OpenMP	http://openmp.org/
MPI	http://www.mpi-forum.org/ , http://www.mcs.anl.gov/mpi/
Windows HPC Server 2008 homepage	http://www.microsoft.com/HPC/
Windows HPC Server 2008 developer resource page	http://www.microsoft.com/hpc/dev

This document was developed prior to the product's release to manufacturing, and as such, we cannot guarantee that all details included herein will be exactly as what is found in the shipping product.

The information contained in this document represents the current view of Microsoft Corporation on the issues discussed as of the date of publication. Because Microsoft must respond to changing market conditions, it should not be interpreted to be a commitment on the part of Microsoft, and Microsoft cannot guarantee the accuracy of any information presented after the date of publication.

This document is for informational purposes only. MICROSOFT MAKES NO WARRANTIES, EXPRESS, IMPLIED, OR STATUTORY, AS TO THE INFORMATION IN THIS DOCUMENT.

Complying with all applicable copyright laws is the responsibility of the user. Without limiting the rights under copyright, no part of this document may be reproduced, stored in or introduced into a retrieval system, or transmitted in any form or by any means (electronic, mechanical, photocopying, recording, or otherwise), or for any purpose, without the express written permission of Microsoft Corporation.

Microsoft may have patents, patent applications, trademarks, copyrights, or other intellectual property rights covering subject matter in this document. Except as expressly provided in any written license agreement from Microsoft, the furnishing of this document does not give you any license to these patents, trademarks, copyrights, or other intellectual property.

© 2008 Microsoft Corporation. All rights reserved.

Microsoft, Active Directory, Visual C++, Visual C#, Visual Studio, Windows, the Windows logo, Windows PowerShell, Windows Server, and Windows Vista are trademarks of the Microsoft group of companies.

All other trademarks are property of their respective owners.