

SET10108 - Concurrent and Parallel Systems

Dr Kevin Chalmers

Contents

	Page
1 Multi-threading	1
1.1 Starting a Thread	1
1.1.1 Waiting for a Thread to Complete	1
1.1.2 A First Multi-Threaded Example	1
1.2 Multiple Tasks	2
1.3 Passing Parameters to Threads	4
1.3.1 Random Numbers in C++11	4
1.3.2 Ranged For Loops in C++11	5
1.3.3 Test Application	5
1.4 Using Lambda (λ) Expressions	6
1.4.1 What is a λ Expression?	6
1.4.2 λ Expressions in C++11	7
1.4.3 Example Application	7
1.5 λ Expressions and Threads	10
1.6 Gathering Data	11
1.6.1 Work Function	11
1.6.2 Creating a File	11
1.6.3 Capturing Times	12
1.6.4 Complete Application	12
1.6.5 Getting the Data	13
1.7 Monte Carlo π	13
1.7.1 Theory	13
1.7.2 Distributions for Random Numbers	14
1.7.3 Monte Carlo π Algorithm	15
1.7.4 Main Application	15
1.7.5 Results	17
1.8 Exercises	17
1.8.1 Class Challenge - Monte Carlo π Calculation	18
1.9 Reading	19
2 Controlling Multi-threaded Applications	21
2.1 Shared Memory Problems	21
2.1.1 Shared Pointers in C++11	21
2.1.2 Application	22
2.2 Mutex	23
2.3 Lock Guards	24
2.4 Condition Variables	25
2.4.1 Using Condition Variables	26
2.4.2 Application	26

2.5	Guarded Objects	28
2.6	Thread Safe Data Structures	30
2.6.1	Overview	30
2.6.2	Push	30
2.6.3	Pop	31
2.6.4	Empty	31
2.6.5	Tasks	31
2.6.6	Main Application	32
2.7	Exercises Part 1	33
2.8	Atomics	34
2.8.1	Sample Atomic Application	34
2.8.2	<code>atomic_flag</code>	36
2.9	Futures	37
2.9.1	Example Future Application	38
2.10	Fractals	40
2.10.1	Mandelbrot	40
2.10.2	Mandelbrot Algorithm	40
2.11	Exercises Part 2	43
2.11.1	Challenge - A Synchronous Channel	44
2.12	Reading	44
3	OpenMP	45
3.1	First OpenMP Application	45
3.2	<code>parallel for</code>	46
3.2.1	Calculating π (not using Monte Carlo Simulation)	46
3.2.2	Parallel For Implementation of π Approximation	47
3.3	Bubble Sort	48
3.3.1	<code>generate_values</code>	48
3.3.2	<code>bubble_sort</code>	49
3.3.3	Main Application	49
3.4	Parallel Sort	50
3.5	The Trapezoidal Rule	52
3.5.1	Trapezoidal Function	53
3.5.2	Testing the Trapezoidal Algorithm	54
3.6	Scheduling	57
3.6.1	Test Function	58
3.6.2	Main Application	58
3.7	Concurrency Visualizer	59
3.8	Exercises	60
3.9	Reading	60
4	CPU Instructions	63
4.1	Memory Alignment	63
4.2	SIMD Operations	64
4.3	Normalizing a Vector	65
4.4	Exercises	68

5	Distributed Parallelism with MPI	69
5.1	Installing MPI	69
5.2	First MPI Application	69
5.3	Running an MPI Application	70
5.4	Using a Remote Host	70
5.5	Sending and Receiving	71
5.6	Map-Reduce	74
5.7	Scatter-Gather	74
5.8	Broadcast	77
5.9	Exercises	78
6	More MPI	79
6.1	Mandelbrot	79
6.2	Parallel Sort	80
6.3	Trapezoidal Rule	83
6.4	Performance Evaluation of MPI	84
6.4.1	Measuring Latency	85
6.4.2	Measuring Bandwidth	86
6.5	Exercises	86
6.6	Reading	86
7	GPU Programming with OpenCL	87
7.1	Getting Started with OpenCL	87
7.2	Getting OpenCL Info	89
7.3	Loading an OpenCL Kernel	91
7.4	Passing Data to OpenCL Kernels	93
7.5	Running and Getting Results from OpenCL Kernels	94
7.6	Matrix Multiplication	96
8	GPU Programming with CUDA	99
8.1	Getting Started with CUDA	99
8.2	Getting CUDA Info	99
8.3	CUDA Kernels	101
8.4	Passing Data to CUDA	101
8.5	Running and Getting Results from CUDA	102
8.6	Freeing Resources	103
8.7	Matrix Multiplication	103
9	GPU Programming Examples	105
9.1	Monte Carlo π	105
9.1.1	OpenCL Monte Carlo π	105
9.1.2	CUDA Monte Carlo π	106
9.1.3	Exercises	109
9.2	Mandelbrot Fractal	109
9.2.1	OpenCL Mandelbrot	109
9.2.2	Exercise	111
9.3	Trapezoidal Rule	111
9.3.1	Exercise	112
9.4	Image Rotation	112
9.4.1	Exercises	115
9.5	Profiling and Debugging	115

10 C++ AMP	117
10.1 Getting C++ AMP Information	117
10.2 Vector Addition Using C++ AMP	118
10.2.1 Declaring Host Memory	118
10.2.2 Creating Device Memory	118
10.2.3 Vector Addition Function in C++ AMP	118
10.2.4 Running the <code>vecadd</code> Function	119
10.2.5 Complete Main for C++ AMP <code>vecadd</code> Application	119
10.3 Matrix Multiplication Using C++ AMP	120
10.4 Monte Carlo π Using C++ AMP	121
10.5 Mandelbrot Using C++ AMP	121
10.6 Trapezoidal Rule Using C++ AMP	122
10.7 Image Rotation Using C++ AMP	123
10.8 Profiling and Debugging	123
10.9 Exercises	123

List of Figures

1.1	Output from Hello World Threading Application	2
1.2	Output from Multiple Tasks Application	4
1.3	Output from Random Numbers Application	6
1.4	Output from λ Expression Application	10
1.5	A Circle of Radius R in a Square of Side $2R$	14
1.6	Monte Carlo π Results	18
2.1	Shared Memory Increment Output	23
2.2	Interleaving Threads	23
2.3	Output from <code>mutex</code> Protected Increment	25
2.4	Output from <code>condition_variable</code> Application	28
2.5	<code>threadsafe_stack</code> Output	33
2.6	<code>atomic_flag</code> Output	37
2.7	Task Manager Output from <code>atomic_flag</code> Application	38
2.8	Mandelbrot Fractal	41
2.9	Split Mandelbrot Fractal	42
3.1	Output from Hello OpenMP Application	46
3.2	Output from <code>parallel for</code> π Calculation	48
3.3	Bubble Sort Performance (no Log Scale)	50
3.4	Bubble Sort Performance (y-axis \log_2 Scale)	51
3.5	Parallel Sort Performance (no Log Scale)	52
3.6	Parallel Sort Performance (y-axis \log_2 Scale)	53
3.7	Trapezoidal Rule	54
3.8	Cosine Function	55
3.9	Sine Function	55
3.10	Output from Trapezoidal Rule Application Using Cosine Function . . .	56
3.11	Output from Trapezoidal Rule Application Using Sine Function . . .	57
3.12	Concurrency Visualizer Overview	59
3.13	Concurrency Visualizer Threads View	60
3.14	Concurrency Visualizer Cores View	61
4.1	Output form SIMD Add Application	66
4.2	Output from SIMD Normalisation Application	68
5.1	Output from Initial MPI Application	71
5.2	Output from <code>ipconfig</code>	72
5.3	Output from MPI Send-Receive Application	73
5.4	Output from Monte-Carlo π MPI Map-Reduce	75
5.5	Output from MPI Scatter-Gather Application	77
6.1	Output from MPI Parallel Sort Algorithm	83

6.2	Output from MPI Trapezoidal Rule Application	84
7.1	GPU Properties from OpenCL	90
7.2	Output from OpenCL Application	96
8.1	CUDA Device Information	100
9.1	Rotating an Image	113

List of Algorithms

1	Monte Carlo π	14
2	Bubble Sort Algorithm	49
3	SIMD Vector Normalization	67

Listings

1.1	Including the <code>thread</code> Header	1
1.2	Thread Creation	1
1.3	Joining a Thread	1
1.4	Hello World Threading	2
1.5	Sleeping For a Number of Seconds	3
1.6	Including the <code>chrono</code> Header	3
1.7	Multiple Tasks	3
1.8	Including the <code>random</code> Header	4
1.9	Creating Default Random Engine	5
1.10	Getting a Random Number	5
1.11	Ranged For Loop	5
1.12	Random Number Generation Application	5
1.13	Add λ Expression in C++	7
1.14	Using the Add Function	7
1.15	Creating and Using a Simple Add Expression	7
1.16	Including the <code>functional</code> Header	8
1.17	Creating a Function Object	8
1.18	Fixed Values in λ Expressions	8
1.19	Reference Values in λ Expressions	9
1.20	Complete λ Expression Example	9
1.21	Hello World Thread Using λ Expressions	10
1.22	Work Function	11
1.23	Creating a File	11
1.24	Gathering Timings	12
1.25	Gathering Data Application	12
1.26	Using Distributions for Randoms	15
1.27	Monte Carlo π in C++	15
1.28	Monte Carlo π Main Application	16
2.1	Creating an <code>int shared_ptr</code>	21
2.2	Including the <code>memory</code> Header	22
2.3	Increment Function	22
2.4	Shared Memory Problem Application	22
2.5	Including the <code>mutex</code> Header	24
2.6	Using a <code>mutex</code>	24
2.7	Using a <code>lock_guard</code>	24
2.8	Including the <code>condition_variable</code> Header	26
2.9	Waiting on a <code>condition_variable</code>	26
2.10	Notifying a <code>condition_variable</code>	26
2.11	Waiting for a Set Time	26
2.12	Task 1	26
2.13	Task 2	27

2.14	<code>condition_variable</code> Main Application	27
2.15	Guarded Object Header	28
2.16	Guarded Object Definition	29
2.17	Guarded Object Main Application	29
2.18	<code>threadsafe_stack</code> Constructors	30
2.19	<code>threadsafe_stack</code> <code>push</code>	30
2.20	<code>threadsafe_stack</code> <code>pop</code>	31
2.21	<code>threadsafe_stack</code> <code>empty</code>	31
2.22	<code>pusher</code> Task	32
2.23	<code>popper</code> Task	32
2.24	<code>threadsafe_stack</code> Main Application	32
2.25	Including the <code>atomic</code> Header	34
2.26	Atomic Increment Function	35
2.27	Atomic Increment Main Application	35
2.28	Using <code>atomic_flag</code>	36
2.29	Including the <code>future</code> Header	37
2.30	Using <code>async</code> to Create a Future	37
2.31	Getting the Result from a Future	38
2.32	<code>find_max</code> Function	39
2.33	Futures Main Application	39
2.34	Constants for Mandelbrot Application	40
2.35	Mandelbrot Algorithm	41
2.36	Mandelbrot Main Application	43
3.1	Hello OpenMP	45
3.2	Calculating π With <code>parallel for</code>	47
3.3	<code>generate_values</code> Function	48
3.4	Bubble Sort Main Application	49
3.5	<code>parallel_sort</code> Function	50
3.6	<code>trap</code> Function for Trapezoidal Rule	53
3.7	Trapezoidal Rule Main Function	55
3.8	Test Function for <code>schedule</code>	58
3.9	Testing <code>schedule</code>	58
4.1	Declaring Aligned Memory in Microsoft Visual C++	63
4.2	Adding Using SIMD Operations	64
4.3	Generate Data for SIMD Normalisation	66
4.4	Normalise Vector Using SIMD	66
4.5	Checking Normalisation	67
5.1	Hello MPI	69
5.2	Using <code>MPI_Send</code> and <code>MPI_Recv</code>	71
5.3	MPI Mad-Reduce Monte Carlo π	74
5.4	Helper Methods for MPI Scatter-Gather Vector Normalisation	74
5.5	Using MPI Scatter-Gather to Normalise a Vector	75
5.6	MPI Broadcast Example	77
6.1	MPI Mandelbrot	79
6.2	Merge Algorithms for MPI Parallel Sort	80
6.3	Iteration Merging in MPI Parallel Sort	81
6.4	Odd-Even Parallel Sort using MPI	82
6.5	Main MPI Application for Odd-Even Parallel Sort	82
6.6	Trapezoidal Rule using MPI	84
6.7	MPI Latency Measurement Application	85

7.1	Initialising OpenCL	87
7.2	Using OpenCL Initialise	88
7.3	Printing OpenCL Information	89
7.4	First OpenCL Kernel - Vector Addition	91
7.5	Loading an OpenCL Kernel	91
7.6	OpenCL Main with Kernel Loading	92
7.7	Creating Host Memory for OpenCL Application	93
7.8	Creating Device Memory for OpenCL Application	93
7.9	Copying Host Memory to Device Memory	93
7.10	Setting OpenCL Kernel Arguments	94
7.11	Freeing OpenCL Memory Buffers	94
7.12	Defining OpenCL Work Dimensions	94
7.13	Enqueuing a OpenCL Kernel for Execution	95
7.14	Reading Results Back from GPU Memory in OpenCL	95
7.15	Verifying OpenCL Results	95
7.16	OpenCL Matrix Multiplication Kernel	97
8.1	Initialising a CUDA Application	99
8.2	Getting CUDA Information	100
8.3	CUDA Vector Addition Kernel	101
8.4	Creating Host Memory for CUDA	101
8.5	Allocating Device Memory with CUDA	102
8.6	Copying Host Memory to the Device in CUDA	102
8.7	Running the CUDA Vector Addition Kernel	102
8.8	Copying Device Memory to the Host in CUDA	102
8.9	Freeing CUDA Memory	103
8.10	Matrix Multiply in CUDA	103
9.1	OpenCL Version of Monte Carlo π	106
9.2	CUDA Version of Monte Carlo π	106
9.3	CUDA Version of Monte Carlo π Using a for Loop	107
9.4	Generating 2-dimensional Random Numbers Using CUDA	107
9.5	CUDA Version of Monte Carlo π With Summing on GPU	108
9.6	OpenCL Mandelbrot Kernel	109
9.7	Running the Mandelbrot Kernel	110
9.8	Saving Mandelbrot Data Using FreeImage	111
9.9	Trapezoidal Rule using CUDA	111
9.10	Image Rotation OpenCL Kernel	113
9.11	Loading and Image with FreeImage	113
9.12	Executing the Image Rotation Kernel	114
10.1	Displaying C++ AMP Information	117
10.2	Declaring Host Memory Using std::array	118
10.3	Creating an array_view of Host Memory	118
10.4	vecadd Function in C++ AMP	118
10.5	Running a C++ AMP Function	119
10.6	Complete C++ AMP vecadd Example	119

Unit 1

Multi-threading

In this first tutorial we will look at how we can start up threads in C++. The tutorial uses the new C++11 standard throughout, and therefore you will need a compiler for C++11 (Visual Studio 2012 and above for example). The multi-threaded features of C++ only came in for the C++11 standard.

1.1 Starting a Thread

Creating a thread in C++ is simple. We need to include the `thread` header file as shown in Listing 1.1.

```
1 #include <thread>
```

Listing 1.1: Including the `thread` Header

To create a thread, we then only need create a new `thread` object in our application, passing in the name of the function that we want the thread to run as shown in Listing 1.2.

```
1 thread t(hello_world);
```

Listing 1.2: Thread Creation

This will create a new thread that will run the function `hello_world`. The thread will start executing the function while the main application continues operating on the rest of the code. Essentially the main application is a thread that is created and executed as soon as an application is launched.

1.1.1 Waiting for a Thread to Complete

Generally we will want to wait for a thread to complete its operation. We do this by using the `join` method on the thread as shown in Listing 1.3.

```
1 t.join();
```

Listing 1.3: Joining a Thread

This will mean the currently executing code (whether it be the main application thread or another thread) will wait for the thread it joins to complete its operation.

1.1.2 A First Multi-Threaded Example

With just these two lines of code, we can create a simple Hello World application for C++ multi-threading. The complete code for this example is given in Listing 1.4.

```
1 #include <thread>
2 #include <iostream>
3
4 using namespace std;
5
6 /*
7  This is the function called by the thread
8  */
9 void hello_world()
10 {
11     cout << "Hello from thread " << this_thread::get_id() << endl;
12 }
13
14 int main()
15 {
16     // Create a new thread
17     thread t(hello_world);
18     // Wait for thread to finish (join it)
19     t.join();
20     // Return 0 (OK)
21     return 0;
22 }
```

Listing 1.4: Hello World Threading

Notice on line 11 we use the following instruction:

```
1 this_thread::get_id()
```

This operation will get the (operating system) assigned ID of the thread running. Each time you run the application, you should get a different value. Running this application will give you a simple output as shown in Figure 1.1.

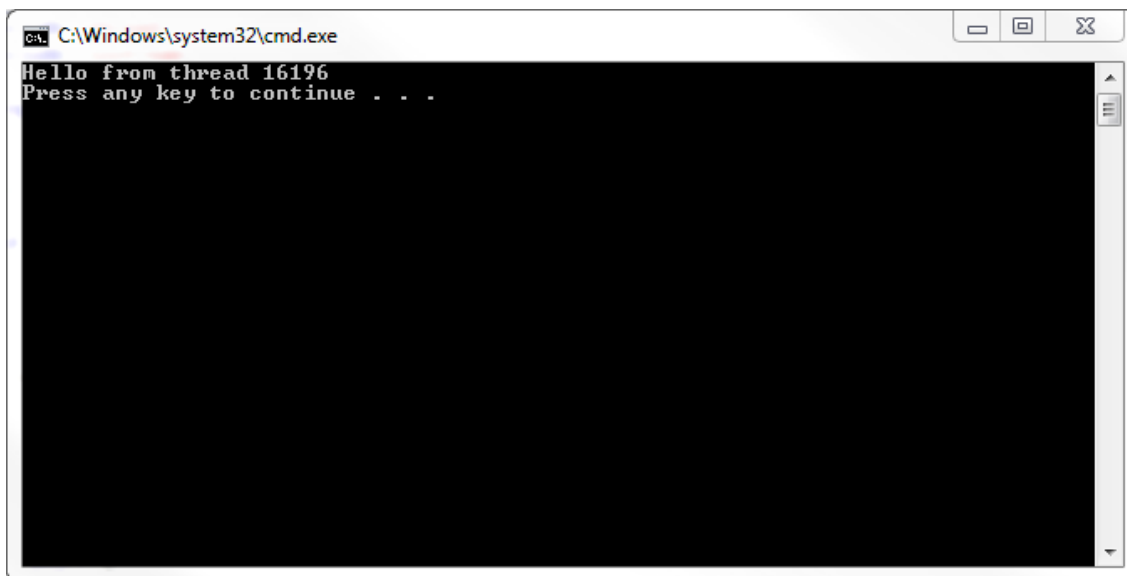


Figure 1.1: Output from Hello World Threading Application

1.2 Multiple Tasks

Starting one thread is all well and good, but we really want to execute multiple tasks. Creating multiple threads is easy we just create multiple thread objects. We

can use the same function if we want (useful for parallelising a task), or we can run different functions (useful for tasks which block computation).

We will use a new operation during this next example `sleep_for`. This operation will allow us to put a thread to sleep for an amount of time. For example, we can use the code in Listing 1.5 to put a thread to sleep for 10 seconds.

```
1 sleep_for(seconds(10));
```

Listing 1.5: Sleeping For a Number of Seconds

To allow us access to the duration constructs (which contains seconds, etc.) we use the `chrono` header as shown in Listing 1.6.

```
1 #include <chrono>
```

Listing 1.6: Including the `chrono` Header

Our application to test the multiple task feature is shown in Listing 1.7.

```
1 #include <thread>
2 #include <chrono>
3 #include <iostream>
4
5 using namespace std;
6 using namespace std::chrono;
7 using namespace std::this_thread;
8
9 void task_one()
10 {
11     cout << "Task one starting" << endl;
12     cout << "Task one sleeping for 3 seconds" << endl;
13     sleep_for(seconds(3));
14     cout << "Task one awake again" << endl;
15     cout << "Task one sleeping for 5 seconds" << endl;
16     sleep_for(milliseconds(5000));
17     cout << "Task one awake again" << endl;
18     cout << "Task one ending" << endl;
19 }
20
21 void task_two()
22 {
23     cout << "Task two starting" << endl;
24     cout << "Task two sleeping for 2 seconds" << endl;
25     sleep_for(microseconds(2000000));
26     cout << "Task two awake again" << endl;
27     cout << "Task two sleeping for 10 seconds" << endl;
28     sleep_for(seconds(10));
29     cout << "Task two awake again" << endl;
30     cout << "Task two ending" << endl;
31 }
32
33 int main()
34 {
35     cout << "Starting task one" << endl;
36     thread t1(task_one);
37     cout << "Starting task two" << endl;
38     thread t2(task_two);
39     cout << "Joining task one" << endl;
40     t1.join();
41     cout << "Task one joined" << endl;
42     cout << "Joining task two" << endl;
43     t2.join();
```

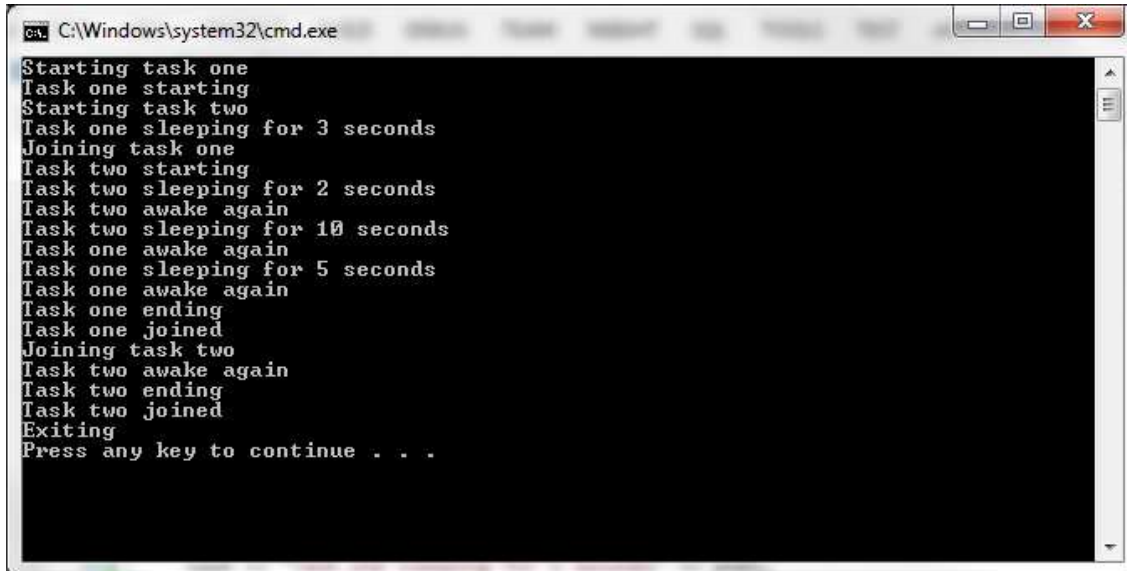
```

44     cout << "Task two joined" << endl;
45     cout << "Exiting" << endl;
46     return 0;
47 }

```

Listing 1.7: Multiple Tasks

This application has been designed to show the different time constructs, as well as showing the interleaving that occurs with multiple tasks. Running this application will give you the output shown in Figure 1.2.



```

C:\Windows\system32\cmd.exe
Starting task one
Task one starting
Starting task two
Task one sleeping for 3 seconds
Joining task one
Task two starting
Task two sleeping for 2 seconds
Task two awake again
Task two sleeping for 10 seconds
Task one awake again
Task one sleeping for 5 seconds
Task one awake again
Task one ending
Task one joined
Joining task two
Task two awake again
Task two ending
Task two joined
Exiting
Press any key to continue . . .

```

Figure 1.2: Output from Multiple Tasks Application

1.3 Passing Parameters to Threads

We have now seen how to create threads in C++11 and how to put them to sleep. The next piece of functionality we are going to use is how we pass parameters to a thread. This is actually fairly easy, and just requires us adding the passed parameters to the thread creation call. For example, given a function with the declaration:

```
1 void task(int n, int val)
```

We can create a thread and pass in parameters to `n` and `val` as follows:

```
1 thread t(task, 1, 20);
```

`n` will be assigned the value 1 and `val` will be assigned the value 10. Our test application this time will use random number generation, which has also changed in C++11.

1.3.1 Random Numbers in C++11

C++11 has added an entirely new random number generation mechanism which is very different, as well as having more functionality. To use random numbers we need to include the `random` header as shown in Listing 1.8.

```
1 #include <random>
```

Listing 1.8: Including the `random` Header

We then need to create a random number generation engine. There are a number of generation engines, but we will use the default one in C++11. We create this as shown in Listing 1.9.

```
1 default_random_engine e(seed);
```

Listing 1.9: Creating Default Random Engine

`seed` is a value used to seed the random number engine (you should hopefully know by now that we cannot create truly random numbers so a seed defines the sequence the same seed will produce the same sequence of random numbers). We will use the system clock to get the current time in milliseconds to use as a seed. To get a random number from the engine we simply use the code shown in Listing 1.10

```
1 auto num = e();
```

Listing 1.10: Getting a Random Number

1.3.2 Ranged For Loops in C++11

The other new functionality we will use in this example is a ranged for loop. You will probably be familiar with the `foreach` loop in C#. The ranged for loop in C++11 has the same functionality. It looks as shown in Listing 1.11.

```
1 for (auto &t : threads)
```

Listing 1.11: Ranged For Loop

We can then use `t` as an object reference in our loop. The `threads` variable is a collection of some sort (here a vector, but we can use arrays, maps, etc.). This is much easier than the iterator approach in pre-C++11 code.

1.3.3 Test Application

Our test application will create 100 threads which will print out an index and the random number generated. The threads will be stored in a vector so that we can join them all (ensuring that they are finished). The application is as follows:

```
1 #include <thread>
2 #include <iostream>
3 #include <vector>
4 #include <random>
5 #include <chrono>
6
7 using namespace std;
8 using namespace std::chrono;
9
10 const int num_threads = 100;
11
12 void task(int n, int val)
13 {
14     cout << "Thread: " << n << " Random Value: " << val << endl;
15 }
16
17 int main()
18 {
19     // C++11 style of creating a random with the current time in
20         milliseconds
```

```

20 |     auto millis = duration_cast<milliseconds>(system_clock::now().
    |         time_since_epoch());
21 |     default_random_engine e(static_cast<unsigned int>(millis.count
    |         ()));
22 |
23 |     // Create 100 threads in a vector
24 |     vector<thread> threads;
25 |     for (int i = 0; i < num_threads; ++i)
26 |         threads.push_back(thread(task, i, e()));
27 |
28 |     // Use C++11 ranged for loop to join the threads
29 |     // Same as foreach in C#
30 |     for (auto &t : threads)
31 |         t.join();
32 |
33 |     return 0;
34 | }

```

Listing 1.12: Random Number Generation Application

Running the application gives the output shown in Figure 1.3.

```

C:\Windows\system32\cmd.exe
Thread: 12 Random Value: -995674595
Thread: 41 Random Value: 272015038
Thread: 54 Random Value: -371926525
25 Random Value: -1140376126
Thread: 77 Random Value: 1571578716
Thread: 90 Random Value: -1868171819
Thread: 35 Random Value: 318966490
Thread: 50 Random Value: 488528884
Thread: 68 Random Value: -337795689
Thread: 10 Random Value: -1856096744
Thread: 13 Random Value: -1884492086
Thread: 7 Random Value: 1913272334
Thread: 74 Random Value: 327371651
Thread: 93 Random Value: -759953590
Thread: 58 Random Value: 1093121183
Thread: 84 Random Value: -1766382673
Thread: 42 Random Value: -1319116908
Thread: 79 Random Value: -1445105346
Thread: 45 Random Value: -281251119
Thread: 32 Random Value: -1655506649
Thread: 27 Random Value: -145327309
Thread: 62 Random Value: -1088008562
Thread: 97 Random Value: 1693605023
Thread: 61 Random Value: -1119938744
Press any key to continue . . .

```

Figure 1.3: Output from Random Numbers Application

1.4 Using Lambda (λ) Expressions

Lambda (λ) expressions are another new feature of C++11, and are becoming a popular feature in object-oriented languages in general (C# has them, Java 8 will have them). λ expressions come from functional style languages (for example F#, Haskell, etc.). They allow us to create function objects which we can apply parameters to and get a result.

1.4.1 What is a λ Expression?

A λ expression is essentially a function. However, it has some properties that allow us to manipulate the function. For example, if we define a function as follows:

$$\text{add}(x, y) = x + y$$

We can then use this function object to create a new function object that adds 3 to any parameter:

$$\text{add3}(y) = \text{add}(3)$$

We can then use the *add3* function as a new function. We can then get the results from function calls as follows:

$$\begin{aligned}\text{add}(10, 5) &= 15 \\ \text{add3}(10) &= 13 \\ \text{add}(2, 0) &= 2 \\ \text{add4}(y) &= \text{add}(4) \\ \text{add4}(6) &= 10\end{aligned}$$

1.4.2 λ Expressions in C++11

One advantage of C++11 λ expressions is that they allow us to create functions using fewer lines of code. Traditionally we would create an entire function / method definition with parameters and return type. In C++11, we can simplify this. For example, we can create an add function as shown in Figure 1.13.

```
1 auto add = [](int x, int y) { return x + y; };
```

Listing 1.13: Add λ Expression in C++

The `[]` allows us to pass parameters into the function (we will look at this capability later). We define the parameters we want to pass to the function. Finally, in the curly brackets we define the λ expression. We can then use the add function as a normal function as shown in Figure 1.14.

```
1 auto x = add(10, 12);
```

Listing 1.14: Using the Add Function

1.4.3 Example Application

We will now look at a collection of different ways to use λ expressions in C++11. We are brushing the surface of lambda expressions here, but it is enough for what we are doing in the module.

Simple λ Expression

We have already looked at this add example. Listing 1.15 shows the example completely.

```
1 // Create lambda expression
2 auto add = [](int x, int y) { return x + y; };
3 // Call the defined function
4 auto x = add(10, 12);
5 // Display answer - should be 22
6 cout << "10 + 12 = " << x << endl;
```

Listing 1.15: Creating and Using a Simple Add Expression

This is the simplest form of lambda expression in C++11. We use `auto` to allow us not to worry about the type of the expression the compiler will generate it for us.

Function Objects

Another method is to create an actual function object, defining the signature accordingly. To do this, we need to include the `functional` header as shown in Listing 1.16.

```
1 #include <functional>
```

Listing 1.16: Including the `functional` Header

We can then create a function object as shown in Listing 1.17.

```
1 // Create function object with same lambda expression
2 function<int (int, int)> add_function = [](int x, int y){ return x
  + y; };
3 // Call the function object
4 x = add_function(20, 12);
5 // Display the answer - should be 32
6 cout << "20 + 12 = " << x << endl;
```

Listing 1.17: Creating a Function Object

Notice that this is essentially the same as the previous example, except we are defining the type of the function. If you hover over the two definitions (`add` and `add_function`), they will be different. The `add_function` is a function object, whereas `add` just looks like a normal function.

Fixed Values

We already mentioned that we can pass fixed values within the square brackets (`[]`). This is shown in Listing 1.18.

```
1 int a = 5;
2 int b = 10;
3 // Define the values passed to the function
4 auto add_fixed = [a, b]{ return a + b; };
5 // Call the function
6 x = add_fixed();
7 // Display the answer - should be 15
8 cout << "5 + 10 = " << x << endl;
```

Listing 1.18: Fixed Values in λ Expressions

A point to note here is that if we change the values of `a` and `b`, the output of `add_fixed` does not change. So, if we do the following:

```
1 // Change the values of a and b
2 a = 20; b = 30;
3 // Call the fixed function again
4 x = add_fixed();
5 // Display the answer - will come out as 15
6 cout << "20 + 30 = " << x << endl;
```

The output of `add_fixed` will still be 15, not 50.

Reference Values

Although not a good idea (especially if we are trying to work in a functional style), we can pass values to the function as references. We can therefore change the function definition to that shown in Listing 1.19.

```

1 // Define the values passed to the function, but pass as a
  reference
2 auto add_reference = [&a, &b] { return a + b; };
3 // Call the function
4 x = add_reference();
5 // Display the answer - should be 50
6 cout << "20 + 30 = " << x << endl;
7 // Change values of a and b
8 a = 30; b = 5;
9 // Call the reference based function again
10 x = add_reference();
11 // Display the answer - should be 35
12 cout << "30 + 5 = " << x << endl;

```

Listing 1.19: Reference Values in λ Expressions

Note the use of `&` to denote we are passing by reference. When we change the values of `a` and `b` now, the output of the function call is also changed.

Complete Application

For completeness, the complete λ expression application is given in Listing

```

1 #include <iostream>
2 #include <functional>
3
4 using namespace std;
5
6 int main()
7 {
8     // Create lambda expression
9     auto add = [](int x, int y) { return x + y; };
10    // Call the defined function
11    auto x = add(10, 12);
12    // Display answer - should be 22
13    cout << "10 + 12 = " << x << endl;
14
15    // Create function object with same lambda expression
16    function<int (int, int)> add_function = [](int x, int y) {
17        return x + y; };
18    // Call the function object
19    x = add_function(20, 12);
20    // Display the answer - should be 32
21    cout << "20 + 12 = " << x << endl;
22
23    int a = 5;
24    int b = 10;
25    // Define the values passed to the function
26    auto add_fixed = [a, b] { return a + b; };
27    // Call the function
28    x = add_fixed();
29    // Display the answer - should be 15
30    cout << "5 + 10 = " << x << endl;
31
32    // Change values of a and b

```

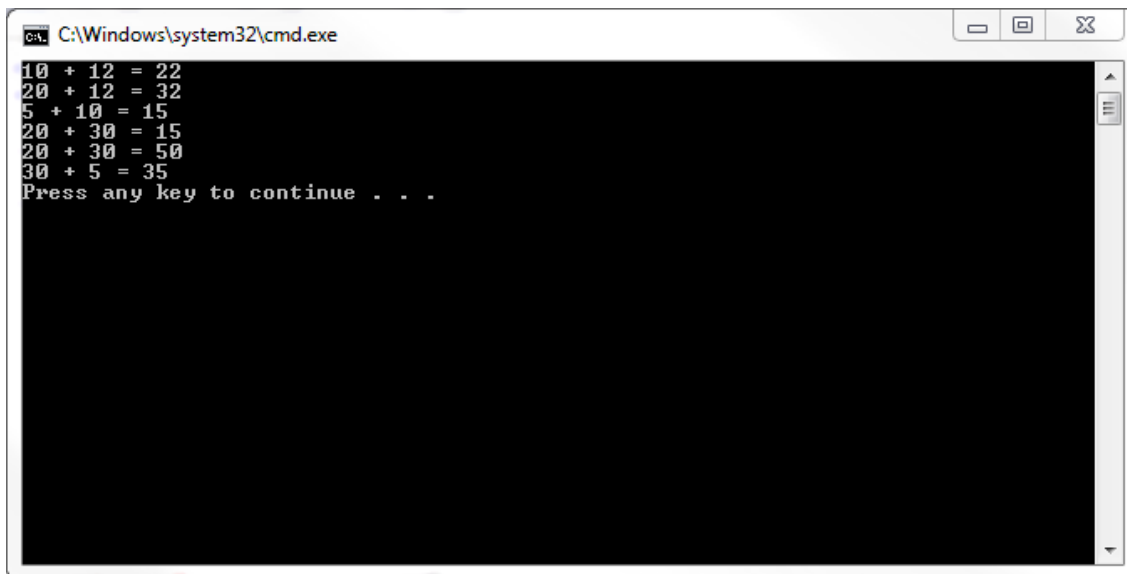
```

32     a = 20; b = 30;
33     // Call the fixed function again
34     x = add_fixed();
35     // Display the answer - will come out as 15
36     cout << "20 + 30 = " << x << endl;
37
38     // Define the values passed to the function, but pass as
        reference
39     auto add_reference = [&a, &b] { return a + b; };
40     // Call the function
41     x = add_reference();
42     // Display the answer - should be 50
43     cout << "20 + 30 = " << x << endl;
44
45     // Change the values of a and b
46     a = 30; b = 5;
47     // Call the reference based function again
48     x = add_reference();
49     // Display the answer - should be 35
50     cout << "30 + 5 = " << x << endl;
51
52     return 0;
53 }

```

Listing 1.20: Complete λ Expression Example

The output is shown in Figure 1.4



```

C:\Windows\system32\cmd.exe
10 + 12 = 22
20 + 12 = 32
5 + 10 = 15
20 + 30 = 15
20 + 30 = 50
30 + 5 = 35
Press any key to continue . . .

```

Figure 1.4: Output from λ Expression Application

1.5 λ Expressions and Threads

We can use λ expressions to create threads, making our code more compact (but not necessarily easier to read). We can repeat our hello world thread application using λ expressions as shown in Listing 1.21.

```

1 #include <thread>
2 #include <iostream>
3

```



```
4 using namespace std;
5
6 int main()
7 {
8     // Create a thread using a lambda expression
9     thread t([]{cout << "Hello from lambda thread!" << endl; });
10    // Join thread
11    t.join();
12
13    return 0;
14 }
```

Listing 1.21: Hello World Thread Using λ Expressions

It is that simple. For the rest of the module, it is up to you whether you use λ expressions or not. We will work interchangeably, and there is no effect on your grades for your choice.

1.6 Gathering Data

Throughout this module, we will be gathering profiling data to allow us to analyse performance of our applications, and in particular the speed up. To do this, we will be outputting data into what is called a comma separated value (.csv) file. This is a file that we can load easily into Excel and generate a chart from. In this first application, we will just get the average of 100 iterations, and our work will be simple (it will just spin the processor and do nothing).

1.6.1 Work Function

Our work function is shown in Listing 1.22.

```
1 void work()
2 {
3     // Do some spinning - no actual processing but will make the CPU
4     // work
5     int n = 0;
6     for (int i = 0; i < 10000000; ++i)
7         ++n;
8 }
```

Listing 1.22: Work Function

The work will simply increment a value 1 million times. This should not take much time for the processor.

1.6.2 Creating a File

Creating a file in C++ is easy, and you should know this by now. As a reminder, we create an output file as shown in Listing 1.23.

```
1 ofstream data("data.csv", ofstream::out);
```

Listing 1.23: Creating a File

This will create an output file called `data.csv` which we can write to. We can treat the file just as any output stream (e.g. `cout`).

1.6.3 Capturing Times

To capture times in C++11, we can use the new `system_clock` object. We have to gather a start time, and end time, and then calculate the total time by subtracting the first from the second. Typically, we will also want to cast the output to milliseconds (ms). An example is shown in Listing 1.24.

```

1 // Get start time
2 auto start = system_clock::now();
3 // ... do some work
4 // Get end time
5 auto end = system_clock::now();
6 // Calculate total time
7 auto total = end - start;
8 // Output total time in ms to the file
9 data << duration_cast<milliseconds>(total).count() << endl;

```

Listing 1.24: Gathering Timings

1.6.4 Complete Application

Our complete application is given in Listing 1.25.

```

1 #include <thread>
2 #include <chrono>
3 #include <iostream>
4 #include <fstream>
5
6 using namespace std;
7 using namespace std::chrono;
8
9 void work()
10 {
11     // Do some spinning - no actual processing but will make the
12     // CPU work
13     int n = 0;
14     for (int i = 0; i < 1000000; ++i)
15         ++n;
16 }
17
18 int main()
19 {
20     // Create a new file
21     ofstream data("data.csv", ofstream::out);
22     // We're going to gather 100 readings, so create a thread and
23     // join it 100 times
24     for (int i = 0; i < 100; ++i)
25     {
26         // Get start time
27         auto start = system_clock::now();
28         // Execute thread
29         thread t(work);
30         t.join();
31         // Get end time
32         auto end = system_clock::now();
33         // Calculate the duration
34         auto total = end - start;
35         // Get that time in ms. Write to file
36         data << duration_cast<milliseconds>(total).count() << endl;
37     }
38 }

```

```

36 | // 100 iterations complete. Close file
37 | data.close();
38 | return 0;
39 | }

```

Listing 1.25: Gathering Data Application

1.6.5 Getting the Data

Once the application is complete, you should have a `.csv` file. Open this with Excel. Then get the mean of the 100 stored values (there is a very good chance that a number of the recorded values are 0).

You also want to document the specification of your machine. This is very important. You can find this in the device manager of your computer. For example, the result from my test is 0.27ms. The hardware I ran the test on is:

CPU Intel i5-2500 @ 3.3GHz

Memory 8GB

OS Windows 7 64bit

This is the only pertinent information at the moment. As we progress through the module, we will find that the other pieces of information about your machine will become useful.

1.7 Monte Carlo π

We now move onto our first case study the approximation of π using a Monte Carlo method. A Monte Carlo method is one where we use random number generation to compute a value. Because we can just increase the number of random values we test, the problem we use can be scaled to however many computations we wish. This will allow us to actually perform a test that we can push our CPU with.

1.7.1 Theory

The theory behind why we can calculate using a Monte Carlo method can best be described using Figure 1.5.

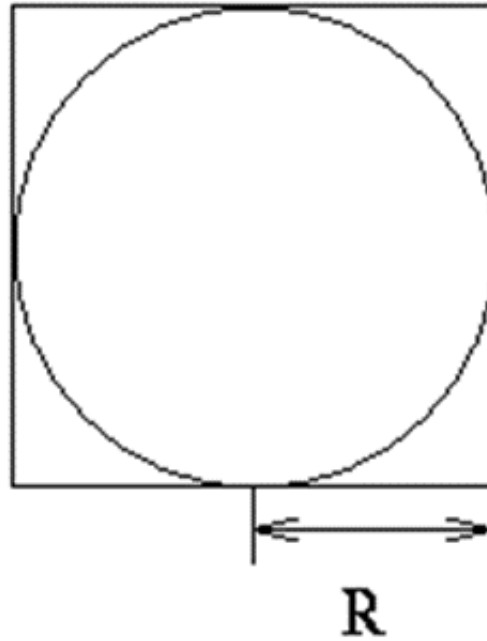
The radius of the circle is defined as R . We know that the area of a circle can be calculated using the following equation:

$$\pi R^2$$

We can also calculate the area of the square as follows:

$$4R^2$$

Now let us imagine that $R=1$, or in other words we have a unit circle. If we pick a random point within the square, we can determine whether it is in the circle by calculating the length of the vector equivalent of the point. If the length is 1 or less, the point is within the circle. If it is greater than 1, then it is in the square but not the circle. The ratio of random points tested to ones in the circle allows us to approximate π . This is because:

Figure 1.5: A Circle of Radius R in a Square of Side $2R$

$$\begin{aligned}
 Area_C &= \pi R^2 \\
 Area_S &= \pi 4R^2 \\
 Ratio &= \frac{\pi R^2}{4R^2} = \frac{\pi}{4}
 \end{aligned}$$

We can then create an algorithm to approximate π as shown in Algorithm 1.

Algorithm 1 Monte Carlo π

```

begin
  attempts := N
  in_circle := M
  ratio :=  $\frac{M}{N}$ 
   $\frac{\pi}{4} \approx \frac{M}{N} \implies \pi \approx 4 \times \frac{M}{N}$ 
end

```

We can therefore approximate π by using generating a collection of random points (we will work in the range $[0.0, 1.0]$ for each coordinate), checking the length and if it is less than or equal to 1 we count this as a hit in the circle.

1.7.2 Distributions for Random Numbers

Before looking at our C++11 implementation of the Monte Carlo π algorithm, we need to introduce the concept of distributions for random numbers. A distribution allows us to define the type of values we get from our random engine and the range of the values. There are a number of distribution types. We will be using a real

distribution (i.e. for double values) and we want the numbers to be uniformly distributed (i.e. they will not tend towards any particular value). We also define the range from 0.0 to 1.0. Our call to do this is shown in Listing 1.26.

```

1 // Create a distribution
2 uniform_real_distribution<double> distribution(0.0, 1.0);
3 // Use this to get a random value from our random engine e
4 auto x = distribution(e);

```

Listing 1.26: Using Distributions for Randoms

1.7.3 Monte Carlo π Algorithm

Our algorithm implementation in C++ is shown in Listing 1.27.

```

1 void monte_carlo_pi(unsigned int iterations)
2 {
3     // Create a random engine
4     auto millis = duration_cast<milliseconds>(system_clock::now().
5         time_since_epoch());
6     default_random_engine e(millis.count());
7     // Create a distribution - we want doubles between 0.0 and 1.0
8     uniform_real_distribution<double> distribution(0.0, 1.0);
9
10    // Keep track of number of points in circle
11    unsigned int in_circle = 0;
12    // Iterate
13    for (unsigned int i = 0; i < iterations; ++i)
14    {
15        // Generate random point
16        auto x = distribution(e);
17        auto y = distribution(e);
18        // Get length of vector defined - use Pythagarous
19        auto length = sqrt((x * x) + (y * y));
20        // Check if in circle
21        if (length <= 1.0)
22            ++in_circle;
23    }
24    // Calculate pi
25    auto pi = (4.0 * in_circle) / static_cast<double>(iterations);

```

Listing 1.27: Monte Carlo π in C++

We have used all the pieces of this algorithm by now, so you should understand it. Notice that we are not actually getting the return value for π at the moment you can print it out if you want to test the accuracy. We will look at how we can get the result from a task in later tutorials.

1.7.4 Main Application

For our main application we are going to capture the amount of time it takes to perform 2^{24} iterations of the Monte Carlo π calculation. We are also going to spread the work across a varying number of threads based on the powers of 2. This means that each thread will perform less work the more threads there are. Table 1.1 will give you an idea of how this works.

Using powers of 2 when performing computation experiments are the norm as computers work on base 2. It also allows us to test for a particular multi-core

Number of Threads	Iterations	Iterations per Thread
$2^0 = 1$	2^{24}	$\frac{2^{24}}{2^0} = 2^{24}$
$2^1 = 2$	2^{24}	$\frac{2^{24}}{2^1} = 2^{23}$
$2^2 = 4$	2^{24}	$\frac{2^{24}}{2^2} = 2^{22}$
$2^3 = 8$	2^{24}	$\frac{2^{24}}{2^3} = 2^{21}$
$2^4 = 16$	2^{24}	$\frac{2^{24}}{2^4} = 2^{20}$
$2^5 = 32$	2^{24}	$\frac{2^{24}}{2^5} = 2^{19}$
$2^6 = 64$	2^{24}	$\frac{2^{24}}{2^6} = 2^{18}$

Table 1.1: Monte Carlo π Iterations

configuration (e.g. dual, quad, etc.). This means we will test for the configuration for your own computer.

The main application code is defined in Listing 1.28.

```

1 int main()
2 {
3     // Create data file
4     ofstream data("montecarlo.csv", ofstream::out);
5
6     for (unsigned int num_threads = 0; num_threads <= 6; ++
          num_threads)
7     {
8         auto total_threads = static_cast<unsigned int>(pow(2.0,
          num_threads));
9         // Write number of threads
10        cout << "Number of threads = " << total_threads << endl;
11        // Write number of threads to the file
12        data << "num_threads_" << total_threads;
13        // Now execute 100 iterations
14        for (unsigned int iters = 0; iters < 100; ++iters)
15        {
16            // Get the start time
17            auto start = system_clock::now();
18            // We need to create total_threads threads
19            vector<thread> threads;
20            for (unsigned int n = 0; n < total_threads; ++n)
21                // Working in base 2 to make things a bit easier
22                threads.push_back(thread(monte_carlo_pi,
          static_cast<unsigned int>(pow(2.0, 24.0 -
          num_threads))));
23            // Join the threads (wait for them to finish)
24            for (auto &t : threads)
25                t.join();
26            // Get the end time
27            auto end = system_clock::now();
28            // Get the total time
29            auto total = end - start;
30            // Convert to milliseconds and output to file
31            data << ", " << duration_cast<milliseconds>(total).
          count();
32        }
33        data << endl;
34    }
35    // Close the file
36    data.close();
37    return 0;

```

38 | }

Listing 1.28: Monte Carlo π Main Application

Again, we have covered every part of this application. You will need to add the relevant include files as well as using statements. Once you have run the application, you will get another `.csv` file.

1.7.5 Results

You should now record your results and the hardware configuration you used. For example, my results are shown in Table 1.2.

Number of Threads	Iterations	Iterations per Thread	Time ms
$2^0 = 1$	2^{24}	$\frac{2^{24}}{2^0} = 2^{24}$	91.32
$2^1 = 2$	2^{24}	$\frac{2^{24}}{2^1} = 2^{23}$	45.8
$2^2 = 4$	2^{24}	$\frac{2^{24}}{2^2} = 2^{22}$	35.96
$2^3 = 8$	2^{24}	$\frac{2^{24}}{2^3} = 2^{21}$	33.64
$2^4 = 16$	2^{24}	$\frac{2^{24}}{2^4} = 2^{20}$	31.07
$2^5 = 32$	2^{24}	$\frac{2^{24}}{2^5} = 2^{19}$	29.78
$2^6 = 64$	2^{24}	$\frac{2^{24}}{2^6} = 2^{18}$	30.92

Table 1.2: Monte Carlo π Timings

Notice that my best time levels out at around 4 to 8 threads. The processor this was run on just happens to be a quad core. Notice as well that performance actually starts to drop as the number of threads increases. This is because threads (and in particular switching threads on the CPU cores) have an overhead. The more threads you create, the greater the overhead. We will analyse this further in a later tutorial.

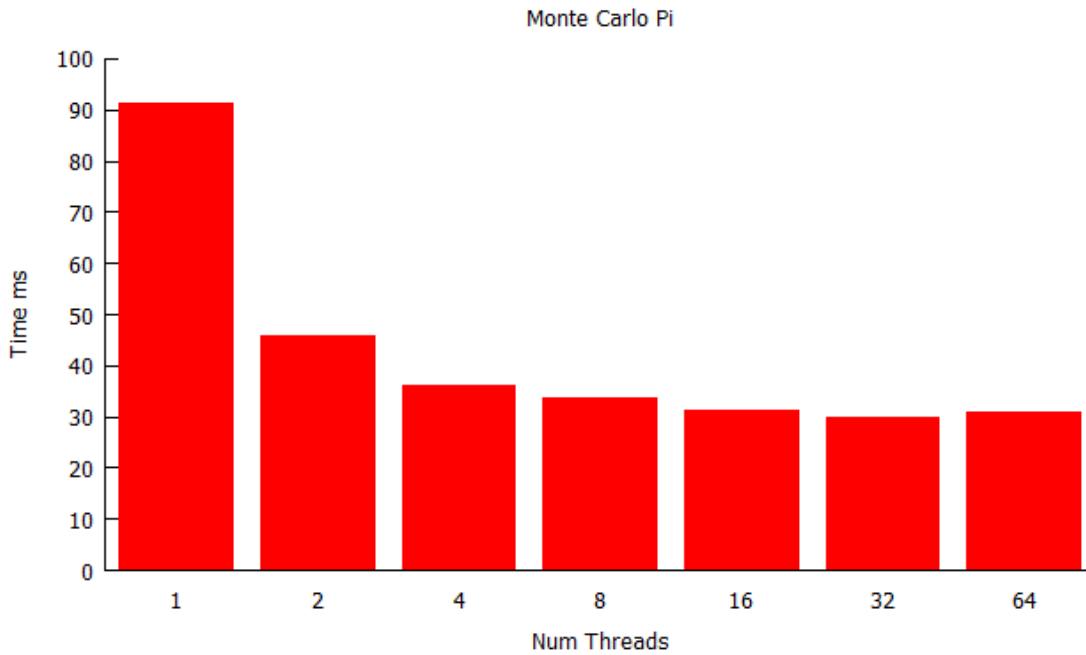
We can also produce a bar chart to illustrate the results (you should do this also in Excel). An example is shown in Figure 1.6.

From now on, you are expected to gather timing results and produce tables and charts as shown in this tutorial. You should keep a logbook of some form to do this. The logbook will be checked every week to see how you are progressing.

1.8 Exercises

These exercises are designed to test your understanding and capabilities in what we have covered so far. As such, they will take time to complete.

1. Create an application that will first generate 10 million random numbers and store them in a vector. Your task is now to find the maximum and minimum values in the vector. As with Monte Carlo π , time the application using different thread configurations. You should also determine some method to store the maximum and minimum values and thus display it. As your first run will use one thread you can test that each different configuration returns the correct maximum and minimum.
2. Write an application that will multiply a matrix of dimensions 1000×1000 by a vector with 1000 components. If you do not remember how we perform a

Figure 1.6: Monte Carlo π Results

matrix-vector multiplication, remember that each component of the resulting 1000 component vector is calculated as follows (assuming 0-indexing):

$$y_i = \sum_{j=0}^{999} a_{ij}x_j$$

If you are unsure about how to do the calculation then do some research into matrix multiplication.

3. Modify the Monte Carlo test to use a λ expression instead. This is relatively easy once you understand λ expressions. Again, gather timing data as before and see if there is any difference in performance.
4. Some of the timings you are gathering will sometimes come out at 0ms. A possible solution is the `high_resolution_clock` and taking the timings in microseconds (abbreviated μs). Change the Monte Carlo π application to use these techniques to get a finer grained analysis.

1.8.1 Class Challenge - Monte Carlo π Calculation

This class has an on-going challenge who can calculate π most accurately in the shortest time using the Monte Carlo method. There are a few rules that are in place for this:

1. Your method must use the Monte Carlo technique described. No other approach of calculating will be accepted.
2. Your method has to work every time. It cannot be a single occurrence that happens unpredictably.
3. Your method has to be witnessed by the lecturer (who will run it multiple times to test it).

4. Your code must be made available for analysis by the lecturer. Your code will be rebuilt and tested again during this process, and will have to perform equivalently.
5. The average of at least 100 iterations must be taken. Your data file must be saved in `.csv` format and the data accessible at the end of the application run. This file will be examined to ensure correct calculation of the mean.
6. For each digit of accuracy, you may increase the amount of time taken by a factor of 10 (an order of magnitude). For example, if you can get 3.14 in 10 seconds, then this is equivalent to someone taking 100 seconds to get 3.141. If someone can get 3.141 in 99.99 seconds, then this is better than 3.14 in 10 seconds.
7. Your method must run on the PCs provided in the Games Lab (B56). This discounts VPNing into your own machine and laptops. However, it does not discount using distributed resources outside B56 or the university as long as the application that gathers and displays the result is within B56.
8. Other rules may be added if necessary to ensure fairness.

1.9 Reading

The reading after each tutorial is very important. It will cement your understanding of the material covered thus far.

This week, you should read chapters 1 and 2 of C++ Concurrency in Action.

Unit 2

Controlling Multi-threaded Applications

In the last tutorial we looked at how we create threads. In this tutorial, we will look more at how we can control multi-threaded applications. Control of multi-threaded applications is really important due to the problems encountered with shared memory. However, controlling multi-threaded applications also leads to problems. The control of multi-threaded applications is really where the concept of concurrency comes in.

The second half of this tutorial will look at some more control concepts atomics and futures. Atomics are a simple method of controlling individual values which are shared between threads. Futures allow us to spin off some work and carry on doing other things, getting the result later.

2.1 Shared Memory Problems

First we will see why shared memory is a problem when dealing with multi-threaded applications. We are going to look at a simple application which increments a value, using a pointer to share the value between all the threads involved. The example is somewhat contrived, but illustrates the problem.

2.1.1 Shared Pointers in C++11

Before looking at the example let us look at another new feature of C++11 smart pointers. Smart pointers have made the world of the C++ programmer much easier in that you no longer need to worry about memory management (well not as much anyway). In this example, we will use a `shared_ptr`. A `shared_ptr` is a pointer that is reference counted. That is, every time you create a copy of the pointer (i.e. provide access to a part of the program), a counter is incremented. Every time the pointer is not required by a part of the program, the counter is decremented. Whenever the counter hits 0, the memory allocated is automatically freed. This does have a (very slight) overhead, but will make our life easier.

For this example, we are going to use the `make_shared` function. `make_shared` will call the relevant constructor for the defined type based on the parameters passed. For example, we can create a shared integer as shown in Listing 2.1.

```
1 auto value = make_shared<int>(0);
```

Listing 2.1: Creating an `int shared_ptr`

The type of value (automatically determined by the compiler) is `shared_ptr<int>`. We can now happily share this value within our programme (with multi-threaded risks) by passing the value around.

To use smart pointers, we need to include the `memory` header as shown in Listing 2.2.

```
1 #include <memory>
```

Listing 2.2: Including the `memory` Header

2.1.2 Application

Our application is quite simple - it will increment our shared value. Listing 2.3 illustrates.

```
1 void increment(shared_ptr<int> value)
2 {
3     // Loop 1 million times, incrementing value
4     for (unsigned int i = 0; i < 1000000; ++i)
5         // Increment value
6         *value = *value + 1;
7 }
```

Listing 2.3: Increment Function

Notice what we do on line 6. We dereference the pointer (this still works for `shared_ptr`) to get the value stored in the `shared_ptr` and add one to the value, setting the `shared_ptr` value with this new incremented value. Our main application is shown in Listing 2.4.

```
1 int main()
2 {
3     // Create a shared int value
4     auto value = make_shared<int>(0);
5
6     // Create number of threads hardware natively supports
7     auto num_threads = thread::hardware_concurrency();
8     vector<thread> threads;
9     for (unsigned int i = 0; i < num_threads; ++i)
10         threads.push_back(thread(increment, value));
11
12     // Join the threads
13     for (auto &t : threads)
14         t.join();
15
16     // Display the value
17     cout << "Value = " << *value << endl;
18 }
```

Listing 2.4: Shared Memory Problem Application

Notice on line 7 we use a call to `thread::hardware_concurrency`. This value is the number of threads that your hardware can handle natively a useful value if you understand our results from the last tutorial. As increment adds 1 million to the shared value, depending on your hardware configuration you might expect a printed value of 1 million, 2 million, 4 million (maybe even 8 million). However, running this application on my hardware I get the result shown in Figure 2.1.

Somehow I lost almost 2 million increments. So where did they go?

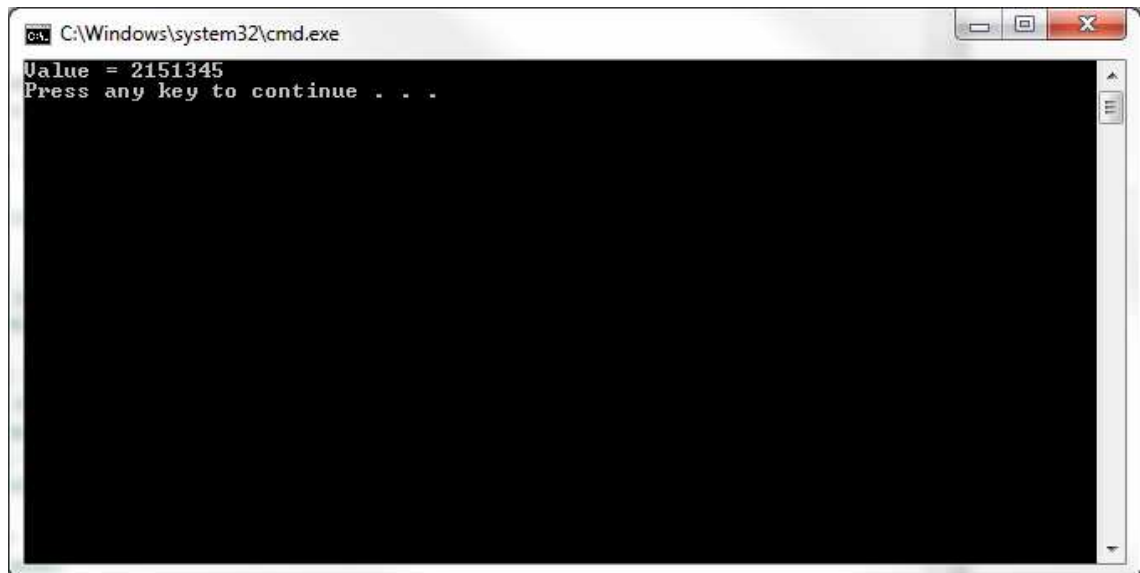


Figure 2.1: Shared Memory Increment Output

This is the problem of shared resources and multi-threading. We do not actually know which order the threads are accessing the resource. For example, if we consider a 4 thread application, something like the Figure 2.2 may happen:

Time	Thread 0	Thread 1	Thread 2	Thread 3	Value
1	Get (0)				0
2	Add (1)	Get (0)	Get (0)		0
3	Store (1)	Add (1)	Add (1)	Get (0)	1
4	Get (1)	Store (1)		Add (1)	1
5		Get (1)		Store (1)	1
6	Add (2)		Get (1)		1
7	Store (2)	Add (2)		Get (1)	2
8		Store (2)	Add (1)		2
9			Store (2)	Add (2)	2
10				Store (2)	2

Figure 2.2: Interleaving Threads

Even though each thread has incremented the value twice (you can check) and the value should be 8, the actual value is 2. Each thread is competing for the resource. This is what we call a **race condition** (or race hazard). This is an important concept to grasp and you need to understand that sharing resources in multi-threaded applications is inherently dangerous if you do not add some control (control does bring its own problems).

2.2 Mutex

The first approach to controlling multi-threaded applications is using what is called a **mutex**. A mutex (mutual exclusion) allows us to guard particular pieces of code so that only one thread can operate within it at a time. If we share the mutex in some manner (mutexes cannot be copied so have to be shared in some manner) then

we can have different sections of code which we protect, ensuring that only one of these sections is running at a time.

To use a mutex in C++11, we need to include the `mutex` header as shown in Listing 2.5.

```
1 #include <mutex>
```

Listing 2.5: Including the `mutex` Header

We will also need a `mutex` as a global variable for our program:

```
1 mutex mut;
```

We can then simply update our increment method from the previous application to use the mutex. The two important methods of the mutex we will be using are `lock` (to lock the mutex thus not allowing any other thread to successfully lock) and `unlock` (freeing the mutex, allowing another thread to call `lock`). Any thread which cannot lock the mutex must wait until the mutex is unlocked and it successfully acquires the lock (there might be a queue of waiting threads).

With a mutex in place, we can update our increment function to that shown in Listing 2.6.

```
1 void increment(shared_ptr<int> value)
2 {
3     // Loop 1 million times, incrementing value
4     for (unsigned int i = 0; i < 1000000; ++i)
5     {
6         // Acquire the lock
7         mut.lock();
8         // Increment value
9         *value = *value + 1;
10        // Release the lock
11        mut.unlock();
12    }
13 }
```

Listing 2.6: Using a `mutex`

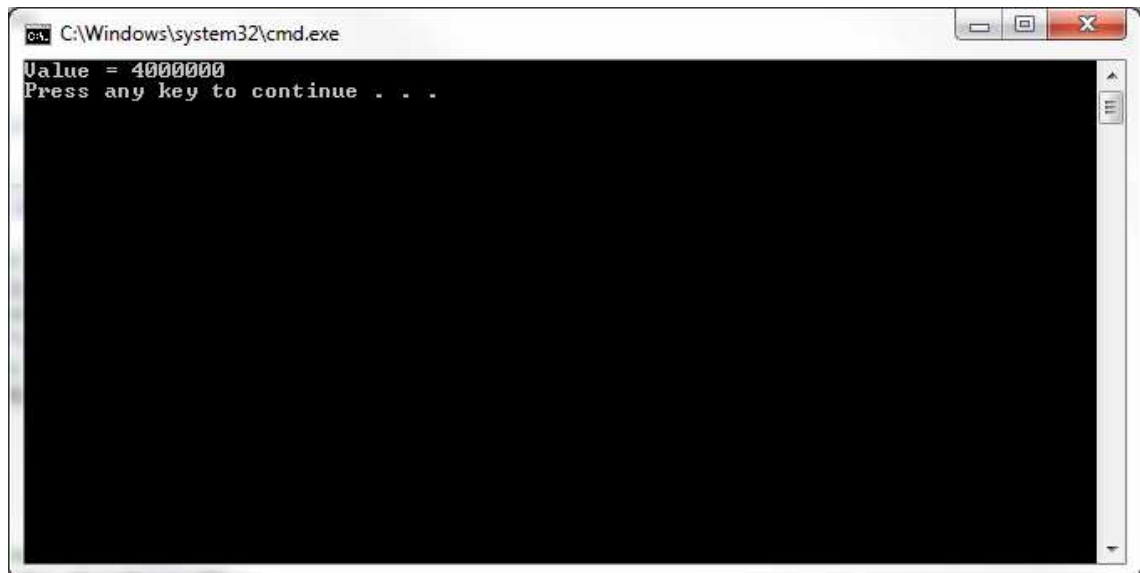
If you run this version of your application (and note that it takes a little longer) your result should be as expected. For example, I get the output shown in Figure 2.3.

2.3 Lock Guards

Remembering to lock and unlock a mutex can cause problems (especially when you forget to lock and unlock) and does not necessary make your code simple to follow. Other problems that can occur come from methods having multiple exit points (including exceptions) which means you might miss when to unlock a mutex. Thankfully, thanks to C++'s object deconstruction at the end of scopes, we can utilise what is known as a lock guard to automatically lock and unlock a mutex for us.

Modifying our application to use lock guards is very simple. A change to the increment function is required as shown in Listing 2.7.

```
1 void increment(shared_ptr<int> value)
2 {
3     // Loop 1 million times, incrementing value
4     for (unsigned int i = 0; i < 1000000; ++i)
5     {
```

Figure 2.3: Output from `mutex` Protected Increment

```

6      // Create the lock guard - automatically acquires mutex
7      lock_guard<mutex> lock(mut);
8      // Increment value
9      *value = *value + 1;
10     // lock guard is automatically destroyed at the end of the
        loop scope
11     // This will release the lock
12 }
13 }

```

Listing 2.7: Using a `lock_guard`

Using mutexes and lock guards is the simplest method of protecting sections of code to ensure shared resources are protected. However, their simplicity does lead to other problems (such as **deadlock**) which we need to overcome.

2.4 Condition Variables

One limitation when using mutexes is that we are only really controlling access to certain sections of the application to try and protect shared resources. You can think of it as having a gate. We let one person in through the gate at any one time, and do not let anyone else enter until the person has left. This is all well and good, but we have no control over what happens outside the gate (where arguments over who is next in line may occur).

Another approach we might want to take is waiting for a signal a sign to state that we can perform some action. We could happily wait, and when the signal is activated we stop waiting and carry on doing some work. This is a technique originally defined using what was called a **semaphore**.

Consider what happens at a set of traffic lights at a cross roads. We have two streams of traffic that wish to use the intersection at the same time. By having a set of lights (think of red as meaning wait and green as meaning signal), we control access to the cross roads. A semaphore (and in C++11 a condition variable) allows us to control access in a similar manner to a set of traffic lights at a cross roads.

2.4.1 Using Condition Variables

To use condition variables, we need to include the `condition_variable` header as shown in Listing 2.8.

```
1 #include <condition_variable>
```

Listing 2.8: Including the `condition_variable` Header

We are going to look at three operations to use with a condition variable. First there is `wait` (Listing 2.9).

```
1 condition.wait(unique_lock<mutex>(mut));
```

Listing 2.9: Waiting on a `condition_variable`

This will enable us to wait until a signal has been received. We have to pass a lock to the `wait` method, and as we are not using the lock anywhere else we use `unique_lock`. This ensures that we are waiting on the mutex.

The next method we are interested in is `notify_one` (Listing 2.10).

```
1 condition.notify_one();
```

Listing 2.10: Notifying a `condition_variable`

This will notify one thread that is currently waiting on the condition variable. There is a similar method called `notify_all` which will signal all waiting threads.

The final method we are interested in is `wait_for` (Listing 2.11).

```
1 if (condition.wait_for(unique_lock<mutex>(mut), seconds(3)))
```

Listing 2.11: Waiting for a Set Time

As you can see, `wait_for` returns a `true` or `false` value. If we are signalled (notified) before the time runs out, then `wait_for` returns `true`. Otherwise, it returns `false`. There is a similar method called `wait_until` which allows you to set the absolute time to stop waiting at.

2.4.2 Application

We are going to create two threads which wait and signal each other, allowing interaction between the threads. Our first thread runs the code in Listing 2.12.

```
1 void task_1(condition_variable &condition)
2 {
3     // Task one will initially sleep for a few seconds
4     cout << "Task 1 sleeping for 3 seconds" << endl;
5     sleep_for(seconds(3));
6     // Notify waiting thread
7     cout << "Task 1 notifying waiting thread" << endl;
8     condition.notify_one();
9     // Now wait for notification
10    cout << "Task 1 waiting for notification" << endl;
11    condition.wait(unique_lock<mutex>(mut));
12    // We are free to continue
13    cout << "Task 1 notified" << endl;
14    // Sleep for 3 seconds
15    cout << "Task 1 sleeping for 3 seconds" << endl;
16    sleep_for(seconds(3));
17    // Notify any waiting thread
18    cout << "Task 1 notifying waiting thread" << endl;
```



```

19 |     condition.notify_one();
20 |     // Now wait 3 seconds for notification
21 |     cout << "Task 1 waiting 3 seconds for notification" << endl;
22 |     if (condition.wait_for(unique_lock<mutex>(mut), seconds(3)))
23 |         cout << "Task 1 notified before 3 seconds" << endl;
24 |     else
25 |         cout << "Task 1 got tired waiting" << endl;
26 |     // Print finished message
27 |     cout << "Task 1 finished" << endl;
28 | }

```

Listing 2.12: Task 1

Our second thread will run code in Listing 2.13.

```

1 | void task_2(condition_variable &condition)
2 | {
3 |     // Task two will initially wait for notification
4 |     cout << "Task 2 waiting for notification" << endl;
5 |     condition.wait(unique_lock<mutex>(mut));
6 |     // We are free to continue
7 |     cout << "Task 2 notified" << endl;
8 |     // Sleep for 5 seconds
9 |     cout << "Task 2 sleeping for 5 seconds" << endl;
10 |    sleep_for(seconds(5));
11 |    // Notify waiting thread
12 |    cout << "Task 2 notifying waiting thread" << endl;
13 |    condition.notify_one();
14 |    // Now wait 5 seconds for notification
15 |    cout << "Task 2 waiting 5 seconds for notification" << endl;
16 |    if (condition.wait_for(unique_lock<mutex>(mut), seconds(5)))
17 |        cout << "Task 2 notified before 5 seconds" << endl;
18 |    else
19 |        cout << "Task 2 got tired waiting" << endl;
20 |    // Sleep for 5 seconds
21 |    cout << "Task 2 sleeping for 5 seconds" << endl;
22 |    sleep_for(seconds(5));
23 |    // Notify any waiting thread
24 |    cout << "Task 2 notifying waiting thread" << endl;
25 |    condition.notify_one();
26 |    // Print finished message
27 |    cout << "Task 2 finished" << endl;
28 | }

```

Listing 2.13: Task 2

Finally, our main application is given in Listing 2.14.

```

1 | int main()
2 | {
3 |     // Create condition variable
4 |     condition_variable condition;
5 |
6 |     // Create two threads
7 |     thread t1(task_1, ref(condition));
8 |     thread t2(task_2, ref(condition));
9 |
10 |    // Join two threads
11 |    t1.join();
12 |    t2.join();
13 |
14 |    return 0;

```

15 }

Listing 2.14: condition_variable Main Application

Note the use of the `ref` function on lines 7 and 8. This is how we create a reference to pass into our thread functions. The interactions between these threads have been organised so that you will get output shown in Figure 2.4 (double check to make sure).

```

C:\Windows\system32\cmd.exe
Task 1 sleeping for 3 seconds
Task 2 waiting for notification
Task 1 notifying waiting thread
Task 1 waiting for notification
Task 2 notified
Task 2 sleeping for 5 seconds
Task 2 notifying waiting thread
Task 2 waiting 5 seconds for notification
Task 1 notified
Task 1 sleeping for 3 seconds
Task 1 notifying waiting thread
Task 1 waiting 3 seconds for notification
Task 2 notified before 5 seconds
Task 2 sleeping for 5 seconds
Task 1 got tired waiting
Task 1 finished
Task 2 notifying waiting thread
Task 2 finished
Press any key to continue . . .

```

Figure 2.4: Output from condition_variable Application

2.5 Guarded Objects

Now that we know how to protect sections of code, and how to signal threads, let us consider how we go about doing thread safe(ish) code in an object-oriented manner. We are going to modify our increment example so that an object controls the counter. First of all we need to define a header file (`guarded.h`). Listing 2.15 provides the contents.

```

1 #include <mutex>
2
3 class guarded
4 {
5 private:
6     std::mutex mut;
7     int value;
8 public:
9     guarded() : value(0) { }
10    ~guarded() { }
11    int get_value() const { return value; }
12    void increment();
13 };

```

Listing 2.15: Guarded Object Header

Notice that we provide the object with its own `mutex`. This is to protect access to our value. We then use a `lock_guard` to control access to the increment method. This goes in our cpp file (`guarded.cpp`). The contents of this file is provided in Listing 2.16.

```
1 #include "guarded.h"
2
3 void guarded::increment()
4 {
5     std::lock_guard<std::mutex> lock(mut);
6     int x = value;
7     x = x + 1;
8     value = x;
9 }
```

Listing 2.16: Guarded Object Definition

The method is somewhat contrived to force multiple operations within the method. Finally, our main method is shown in Listing 2.17.

```
1 #include "guarded.h"
2 #include <iostream>
3 #include <memory>
4 #include <thread>
5 #include <vector>
6
7 using namespace std;
8
9 const unsigned int NUM_ITERATIONS = 1000000;
10 const unsigned int NUM_THREADS = 4;
11
12 void task(shared_ptr<guarded> g)
13 {
14     // Increment guarded object NUM_ITERATIONS times
15     for (unsigned int i = 0; i < NUM_ITERATIONS; ++i)
16         g->increment();
17 }
18
19 int main()
20 {
21     // Create guarded object
22     auto g = make_shared<guarded>();
23
24     // Create threads
25     vector<thread> threads;
26     for (unsigned int i = 0; i < NUM_THREADS; ++i)
27         threads.push_back(thread(task, g));
28     // Join threads
29     for (auto &t : threads)
30         t.join();
31
32     // Display value stored in guarded object
33     cout << "Value = " << g->get_value() << endl;
34
35     return 0;
36 }
```

Listing 2.17: Guarded Object Main Application

Your output window should state that value equals 4 million. The use of a `lock_guard` and an object level `mutex` is the best method to control access to an object. This will ensure that methods are only called when permitted by competing threads.

2.6 Thread Safe Data Structures

To end the first part of this tutorial, we will look at how we can implement a thread safe stack. Data structures are the normal method of storing data, but are the most susceptible to multi-threading problems. This example is taken from C++ Concurrency in Action (slightly modified).

2.6.1 Overview

A stack is one of the simplest data structures. We simply have a stack of values which we can add to the top of (push) or remove from the top of (pop). Our implementation will be very primitive, and will just wrap a standard stack in an object with thread safe operations.

You will need to create a new header file called `threadsafe_stack.h`. First we declare our class and constructors as shown in Listing 2.18.

```

1 #pragma once
2
3 #include <exception>
4 #include <stack>
5 #include <memory>
6 #include <mutex>
7
8 template<typename T>
9 class threadsafe_stack
10 {
11 private:
12     // The actual stack object we are using
13     std::stack<T> data;
14     // The mutex to control access
15     mutable std::mutex mut;
16 public:
17     // Normal constructor
18     threadsafe_stack() { }
19     // Copy constructor
20     threadsafe_stack(const threadsafe_stack &other)
21     {
22         // We need to copy the data from the other stack. Lock
23         // other stack
24         std::lock_guard<std::mutex> lock(other.mut);
25         data = other.data;
26     }

```

Listing 2.18: `threadsafe_stack` Constructors

Notice that we have a copy constructor. When it is copying it must lock the other stack to ensure its copy is correct. Also note the use of the keyword `mutable` on line 15. This keyword indicates that the `mutex` can be modified in `const` methods (where normally we do not mutate an object's state). This is a convenience as our `mutex` does not affect other classes, but we still want to have `const` methods.

2.6.2 Push

Our push method is quite trivial all we need to do is lock the stack for usage. The code is provided in Listing 2.19.

```

1 // Push method. Adds to the stack

```

```

2|     void push(T value)
3|     {
4|         // Lock access to the object
5|         std::lock_guard<std::mutex> lock(mut);
6|         // Push value onto the internal stack
7|         data.push(value);
8|     }

```

Listing 2.19: threadsafe_stack push

2.6.3 Pop

Pop is a little bit more involved. We still have to lock the object, but we must also check if the stack is empty before attempting to return a value. The code is provided in Listing 2.20.

```

1|     // Pop method.  Removes from the stack
2|     T pop()
3|     {
4|         // Lock access to the object
5|         std::lock_guard<std::mutex> lock(mut);
6|         // Check if stack is empty
7|         if (data.empty()) throw std::exception("Stack is empty");
8|         // Access value at the top of the stack.
9|         auto res = data.top();
10|        // Remove the top item from the stack
11|        data.pop();
12|        // Return resource
13|        return res;
14|    }

```

Listing 2.20: threadsafe_stack pop

On line 7 we check if the internal stack is empty, and if so throw an exception. There is a very good chance you have never worked with exceptions in C++ before (it is newish but prior to C++11). They essentially work the same as Java and C#.

2.6.4 Empty

Our final method allows us to check if the stack is empty (Listing 2.21).

```

1|     // Checks if the stack is empty
2|     bool empty() const
3|     {
4|         std::lock_guard<std::mutex> lock(mut);
5|         return data.empty();
6|     }

```

Listing 2.21: threadsafe_stack empty

2.6.5 Tasks

Our test application is going to have one thread add 1 million values to the stack, and the other extract 1 million values from the stack. The approach is not the most efficient (using exceptions to determine if the stack is empty is not a good idea really), but will provide you with an example of exception handling in C++.

Our first task is called **pusher** its job is to push values onto the stack. The code is in Listing 2.22.

```

1 void pusher(shared_ptr<threadsafe_stack<unsigned int>> stack)
2 {
3     // Pusher will push 1 million values onto the stack
4     for (unsigned int i = 0; i < 1000000; ++i)
5     {
6         stack->push(i);
7         // Make the pusher yield. Will give priority to another
           thread
8         this_thread::yield();
9     }
10 }

```

Listing 2.22: pusher Task

Notice the use of `yield` on line 8. This means that the thread will let another thread in front of it if one is waiting. We are adding this to make the pusher yield to our other task (`popper`), meaning the stack will appear empty sometimes.

`popper` is defined in Listing 2.23.

```

1 void popper(shared_ptr<threadsafe_stack<unsigned int>> stack)
2 {
3     // Popper will pop 1 million values from the stack.
4     // We do this using a counter and a while loop
5     unsigned int count = 0;
6     while (count < 1000000)
7     {
8         // Try and pop a value
9         try
10        {
11            auto val = stack->pop();
12            // Item popped. Increment count
13            ++count;
14        }
15        catch (exception e)
16        {
17            // Item not popped. Display message
18            cout << e.what() << endl;
19        }
20    }
21 }

```

Listing 2.23: popper Task

`popper` will try and pop a value from the stack. If it is empty, it will catch an exception and print it. The try-catch construct is similar to Java and C#.

2.6.6 Main Application

Our main application just needs to create our resources, start the two tasks, and then check if the stack is empty at the end (1 million values pushed minus 1 million values popped). Our main application is defined in Listing 2.24.

```

1 int main()
2 {
3     // Create a threadsafe_stack
4     auto stack = make_shared<threadsafe_stack<unsigned int>>();
5
6     // Create two threads
7     thread t1(popper, stack);
8     thread t2(pusher, stack);

```

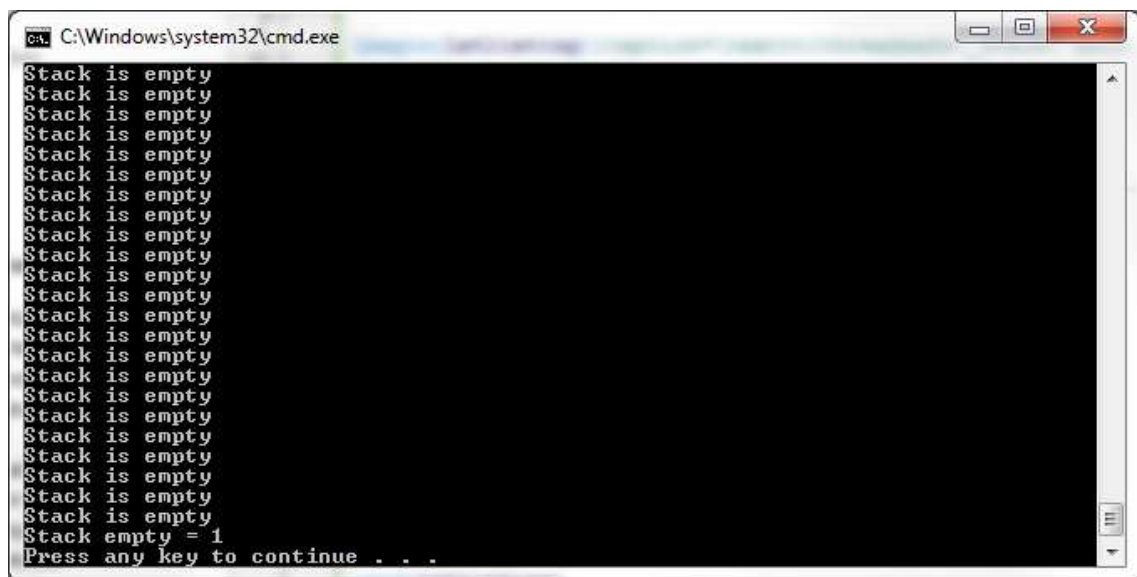
```

9|
10| // Join two threads
11| t1.join();
12| t2.join();
13|
14| // Check if stack is empty
15| cout << "Stack empty = " << stack->empty() << endl;
16|
17| return 0;
18| }

```

Listing 2.24: threadsafe_stack Main Application

Running this application will give the output shown in Figure 2.5.



```

C:\Windows\system32\cmd.exe
Stack is empty
Stack is empty
Stack is empty
Stack is empty
Stack is empty
Stack is empty
Stack is empty
Stack is empty
Stack is empty
Stack is empty
Stack is empty
Stack is empty
Stack is empty
Stack is empty
Stack is empty
Stack is empty
Stack is empty
Stack is empty
Stack is empty
Stack is empty
Stack empty = 1
Press any key to continue . . .

```

Figure 2.5: threadsafe_stack Output

Remember that 1 equals true, so the stack is empty.

2.7 Exercises Part 1

1. Why does the increment application take significantly longer when using a mutex? What is happening on the CPU?
2. You should now be able to repeat the Monte Carlo π experiment but actually gather the result. Your task is to run the experiment multiple times with different iteration values and determine the accuracy of your result. This has to be a consistent result, not a only one run. Run the number of threads your hardware natively supports. You should produce a table. For example, with 2^{24} iterations:

Iterations	Accuracy of π	Time
2^{24}	3.14 (3 digits)	50ms
2^{26}	3.141 (4 digits)	205ms
...

3. Write an application that simulates deadlock. You will need to have at least two locks and two threads to do this in a realistic manner (although it is possible with 1 thread and 1 lock).

4. The C++ Concurrency in Action book has a number of other examples of thread safe data structures, including lock free ones. You should investigate these further to understand some of the techniques usable in object-oriented threaded applications.

2.8 Atomics

Atomics in C++11 are simple data types and related operations. It is actually the operations that are of interest here. An atomic operation is one that cannot be split apart. An example should help determine what we mean.

In our first example in this tutorial, we invoked (a very contrived) data race condition. The problem we had was that our actual increment operation performed the following internal operations:

1. Get value
2. Add 1 to value
3. Store value

You can make a simple observation that there are two “holes” within our increment (assuming serial computation) between 1 and 2, and between 2 and 3. Within these holes, anything could be happening on the machine (another increment for example).

An atomic operation has no “holes”. For example, an increment operation acts as a single indivisible operation there is only before the operation and after, not in the middle of. This means that we do not have to worry about data race conditions (to a certain extent).

The one limiting factor of atomic operations is that they only provide simple operations such as add, load, store, swap, etc. These operations are simple enough to be used with atomic types.

A CPU may also natively support some atomic operations, making life even easier. However, in some cases there may be a lock involved internally (and hidden away) to perform the atomic action.

2.8.1 Sample Atomic Application

We are going to recreate our increment application, but this time using atomics rather than a mutex to protect the data. Our application does not change too much from our original non-mutex version, apart from the use of the atomic value.

To use an atomic in your application, you need the following `atomic` header file as shown in Listing 2.25.

```
1 #include <atomic>
```

Listing 2.25: Including the `atomic` Header

An atomic itself is just a template value in our application. For example, we can define an `atomic int` as follows:

```
1 atomic<int> value;
```


Operation	Equivalent	Description
load		Loads the current atomic value into a variable
store		Stores a value in the atomic variable
exchange		Gets the current atomic value while storing another variable in its place
fetch_add	<code>+=</code>	Gets the atomic value while adding to it
fetch_sub	<code>-=</code>	Gets the atomic value while subtracting from it
fetch_or	<code> =</code>	Gets the atomic value while performing a bitwise or upon it
fetch_and	<code>&=</code>	Gets the atomic value while performing a bitwise and upon it
fetch_xor	<code>^=</code>	Gets the atomic value while performing a bitwise xor upon it
<code>++</code>		Increments the atomic value
<code>--</code>		Decrements the atomic value

Table 2.1: Atomic Operations

Depending on the type of the atomic, we have a number of different operations defined. Table 2.1 provides the common ones we are interested in.

In many regards, we can treat integral (i.e. number) atomic values as normal integral values. We merely state that they are atomic. As such, we can rewrite our increment operation as shown in Listing 2.26.

```

1 void increment(shared_ptr<atomic<int>> value)
2 {
3     // Loop 1 million times, incrementing value
4     for (unsigned int i = 0; i < 1000000; ++i)
5         // Increment value
6         (*value)++;
7 }

```

Listing 2.26: Atomic Increment Function

As we are using a `shared_ptr`, we must dereference it to increment (line 6). A main application to test this increment is shown in Listing 2.27. You might notice that this version of increment is faster than the mutex approach (you should measure it to check).

```

1 int main()
2 {
3     // Create a shared int value
4     auto value = make_shared<atomic<int>>();
5
6     // Create number of threads hardware natively supports
7     auto num_threads = thread::hardware_concurrency();
8     vector<thread> threads;
9     for (unsigned int i = 0; i < num_threads; ++i)
10         threads.push_back(thread(increment, value));
11
12     // Join the threads
13     for (auto &t : threads)
14         t.join();
15
16     // Display the value
17     cout << "Value = " << *value << endl;

```

18 }

Listing 2.27: Atomic Increment Main Application

2.8.2 atomic_flag

Another (slightly unique) atomic construct is something called an `atomic_flag`. An `atomic_flag` can be considered like a Boolean value, although it is more akin to having a signal (either it is set or it is not). An `atomic_flag` provides two methods of interest:

`test_and_set` tests if the flag is set. If it is, then returns `false`. If it is not, sets the flag and returns `true`.

`clear` clears the set flag.

These operations can be very fast on supported CPUs, and are therefore very good for situations where we do not want to put a thread to sleep (wait) while we perform an operation, but rather would like the thread to keep “spinning” (doing something maybe even testing the flag again).

As an example application, consider Listing 2.28.

```

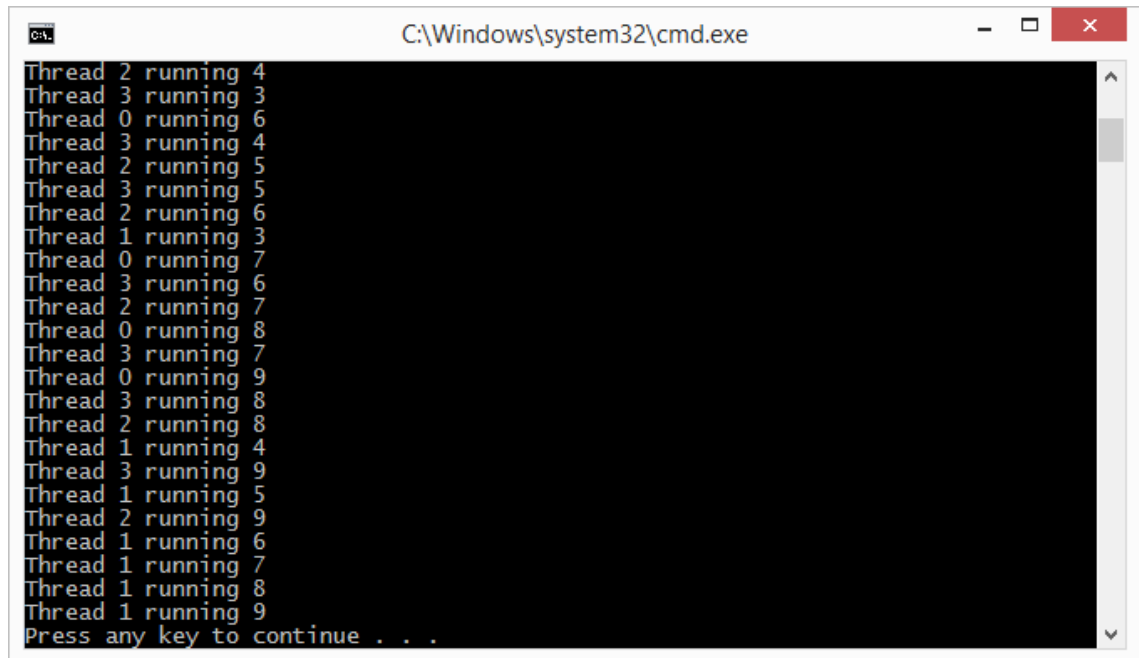
1 void task(unsigned int id, shared_ptr<atomic_flag> flag)
2 {
3     // Do 10 iterations
4     for (unsigned int i = 0; i < 10; ++i)
5     {
6         // Test the flag is available, and grab when it is
7         // Notice this while loops keeps spinning until flag is clear
8         while (flag->test_and_set());
9         // Flag is available. Thread displays message
10        cout << "Thread " << id << " running " << i << endl;
11        // Sleep for 1 second
12        this_thread::sleep_for(seconds(1));
13        // Clear the flag
14        flag->clear();
15    }
16 }
17
18 int main()
19 {
20     // Create shared flag
21     auto flag = make_shared<atomic_flag>();
22
23     // Get number of hardware threads
24     auto num_threads = thread::hardware_concurrency();
25
26     // Create threads
27     vector<thread> threads;
28     for (unsigned int i = 0; i < num_threads; ++i)
29         threads.push_back(thread(task, i, flag));
30
31     // Join threads
32     for (auto &t : threads)
33         t.join();
34
35     return 0;

```

36|}

Listing 2.28: Using `atomic_flag`

The interesting line of code is line 8. Notice that our while loop will keep spinning until the flag can be set by the calling thread. The output from the application is given in Figure 2.6.

Figure 2.6: `atomic_flag` Output

The threads interleave so that only one has the flag at any one time. However, if you look at your CPU utilisation in the Task Manager, you will get something similar to Figure 2.7.

89% CPU utilisation for an application that essentially does nothing. This is because we are using what is known as a spin lock, or busy wait, approach. Putting a thread to sleep does have an overhead, and so a busy wait might be more efficient (particularly for real-time applications). However, it comes with a CPU overhead. We will look at this more closely next week.

2.9 Futures

The final concurrency construct we will look at is futures. Futures are a great way of starting some work, going off to do something else, and then retrieving the result when we are ready. This makes futures a useful method for background processing or for spinning off tasks (such as when we build GUI applications).

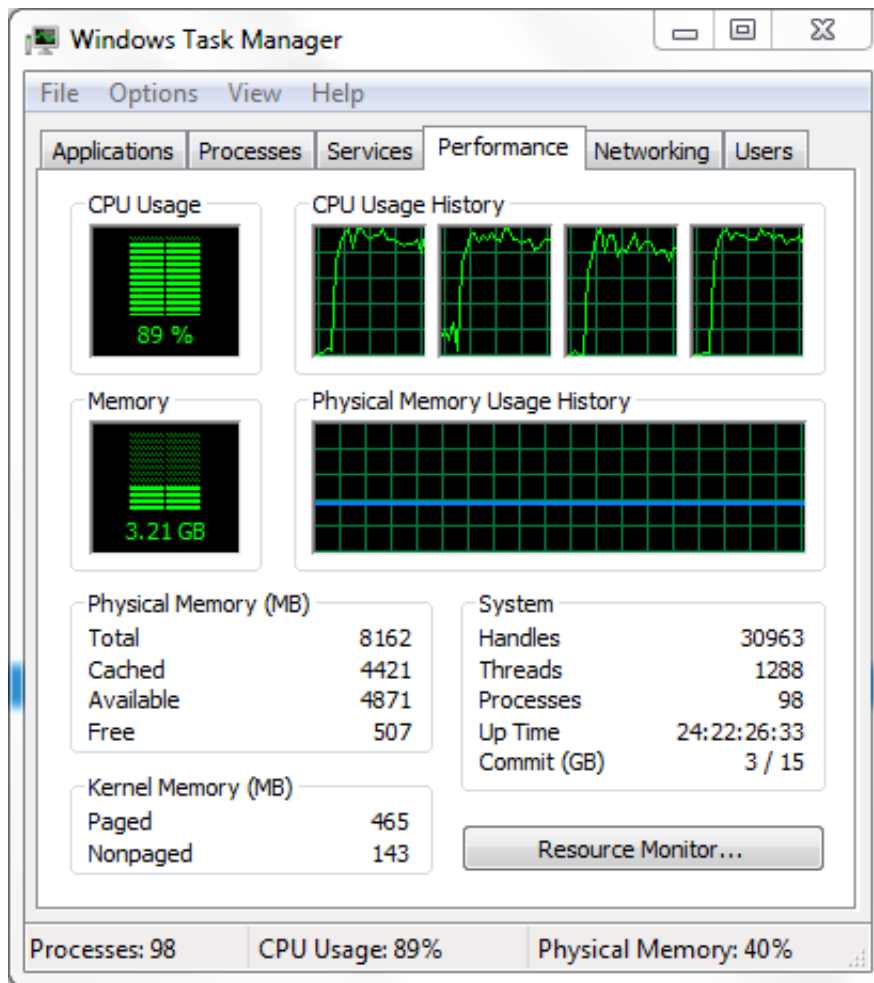
To create a future we need the `future` header as shown in Listing 2.29.

```
1 #include <future>
```

Listing 2.29: Including the `future` Header

There are a number of ways to create a future (see this week's reading). For our purposes, we are going to use the `async` function as shown in Listing 2.30.

```
1 auto f = async(find_max);
```

Figure 2.7: Task Manager Output from `atomic_flag` Application

Listing 2.30: Using `async` to Create a Future

This will create a future using the given function (in this example `find_max`). We can also pass in parameters as we do with thread creation.

To access the result of a future we use the `get` method on the future object as shown in Listing 2.31.

```
1 auto result = f.get();
```

Listing 2.31: Getting the Result from a Future

WARNING - you can only get the result from a future once, so you will want to store it when you do. Calling `get` more than once will throw an exception.

2.9.1 Example Future Application

We will create a very simple example of using futures. Our application will attempt to find the maximum value in a vector of random values by splitting the work across the CPU. To do this, we get the supported hardware concurrency (`n`) and create $n - 1$ futures. The main application will execute on the left over hardware thread.

Think of it this way. If we have 4 hardware threads and are searching a vector with 16 values, we get the following configuration:

1. Future 1 search elements 0 to 3
2. Future 2 search elements 4 to 7
3. Future 3 search elements 8 to 11
4. Main thread search elements 12 to 15

Our `find_max` function is defined in Listing 2.32.

```

1 unsigned int find_max(const vector<unsigned int> &data, unsigned
  int start, unsigned int end)
2 {
3     // Set max initially to 0
4     unsigned int max = 0;
5     // Iterate across vector from start to end position, setting max
  accordingly
6     for (unsigned int i = start; i < end; ++i)
7         if (data.at(i) > max)
8             max = data.at(i);
9
10    // Return max
11    return max;
12 }
```

Listing 2.32: `find_max` Function

This function should be straight forward to understand. Our main application simply creates our futures and gets the maximum from each, displaying the overall maximum. It is shown in Listing 2.33.

```

1 int main()
2 {
3     // Get the number of supported threads
4     auto num_threads = thread::hardware_concurrency();
5
6     // Create a vector with 2^24 random values
7     vector<unsigned int> values;
8     auto millis = duration_cast<milliseconds>(system_clock::now().
  time_since_epoch());
9     default_random_engine e(static_cast<unsigned int>(millis.count())
  );
10    for (unsigned int i = 0; i < pow(2, 24); ++i)
11        values.push_back(e());
12
13    // Create num_threads - 1 futures
14    vector<future<unsigned int>> futures;
15    auto range = static_cast<unsigned int>(pow(2, 24) / num_threads);
16    for (unsigned int i = 0; i < num_threads - 1; ++i)
17        // Range is used to determine number of values to process
18        futures.push_back(async(find_max, ref(values), i * range, (i +
  1) * range));
19
20    // Main application thread will process the end of the list
21    auto max = find_max(values, (num_threads - 1) * range,
  num_threads * range);
22
23    // Now get the results from the futures, setting max accordingly
24    for (auto &f : futures)
25    {
26        auto result = f.get();
```

```

27     if (result > max)
28         max = result;
29     }
30
31     cout << "Maximum value found: " << max << endl;
32
33     return 0;
34 }

```

Listing 2.33: Futures Main Application

Line 18 is where we create our futures, and line 26 where we get the result. The structure of the application is similar to how we have been creating threads, although now we also perform some work in the main application (line 21).

2.10 Fractals

To provide a better understanding of futures we are going to introduce another common problem that can be parallelised - the generation of fractals. A fractal (from a simple point of view - use Wikipedia for a more in depth description) can be used to generate an image that is self-similar at different scales. That is, a part of the image can be zoomed in upon and the image still looks the same. Zooming further the image still looks the same, and so on. The actual definition is more complicated than this, but for our purposes we are using a function that will provide us with the values we need at (x, y) coordinates of an image to produce a fractal image. This week, we are going to generate the Mandelbrot fractal.

2.10.1 Mandelbrot

The Mandelbrot set (which defines the Mandelbrot fractal) is perhaps the most famous fractal. The Mandelbrot fractal is shown in Figure 2.8.

More information on the Mandelbrot set can be found on the Wikipedia page (http://en.wikipedia.org/wiki/Mandelbrot_set) - you will also find the pseudocode for the algorithm we are going to use there.

The Mandelbrot set can be determined on a per pixel basis. This means that we can distribute the pixels to separate threads and therefore parallelise the algorithm. We can also scale the problem by increasing the resolution of the image. This makes Mandelbrot fractal generation ideal for parallel work.

For our approach we are going to split the Mandelbrot image into strips as shown in Figure 2.9.

2.10.2 Mandelbrot Algorithm

There are a few methods to calculate the Mandelbrot set - we are going to use one that relies on the escape value of a loop. That is, based on the number of iterations we perform up to a maximum provides us with the pixel value (0.0 to 1.0). First we need some global values for our application as shown in Listing 2.34.

```

1 // Number of iterations to perform to find pixel value
2 const unsigned int max_iterations = 1000;
3
4 // Dimension of the image (in pixels) to generate
5 const unsigned int dim = 8192;
6

```

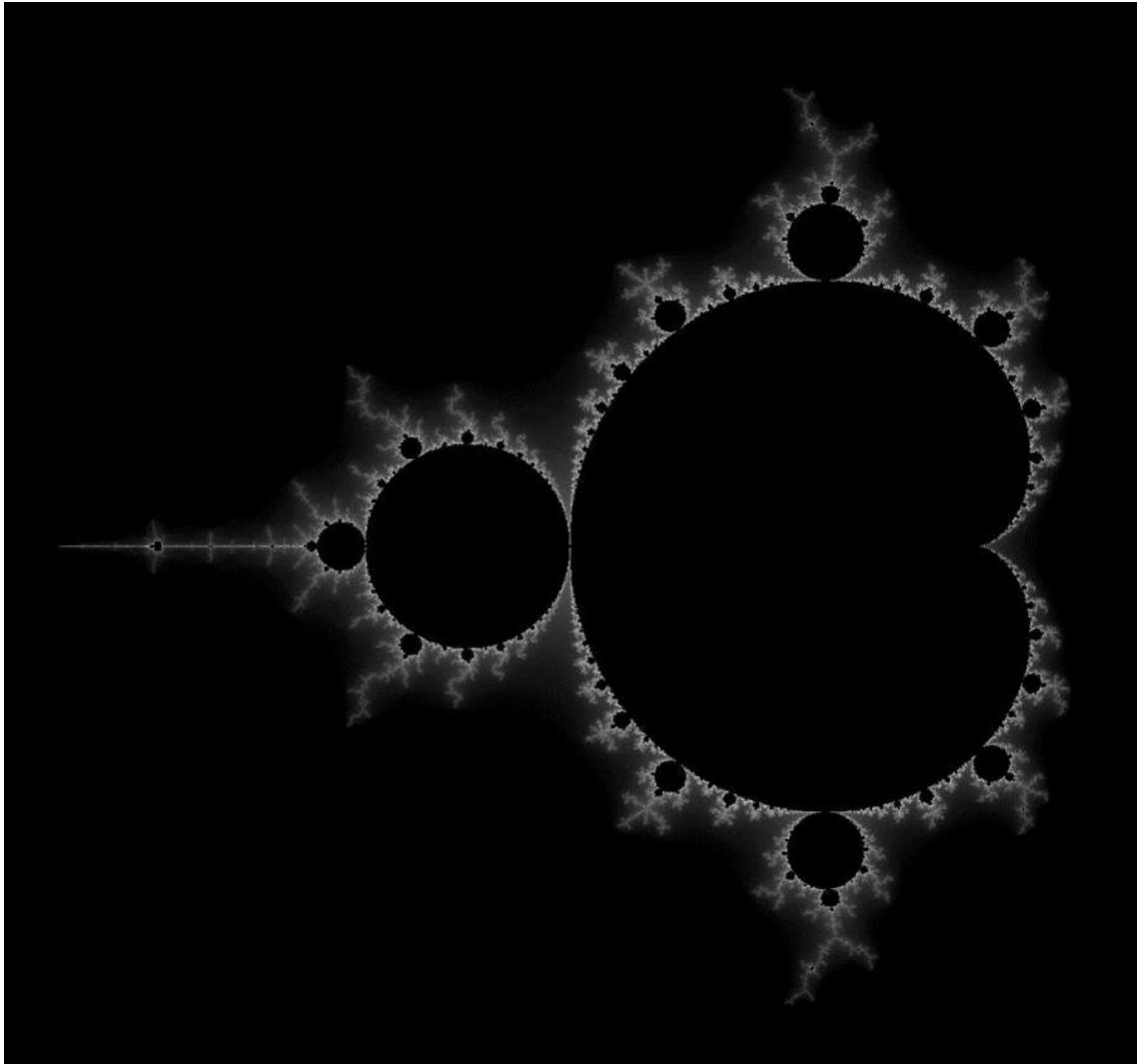


Figure 2.8: Mandelbrot Fractal

```

7 // Mandelbrot dimensions are ([-2.1, 1.0], [-1.3, 1.3])
8 const double xmin = -2.1;
9 const double xmax = 1.0;
10 const double ymin = -1.3;
11 const double ymax = 1.3;
12
13 // The conversion from Mandelbrot coordinate to image coordinate
14 const double integral_x = (xmax - xmin) / static_cast<double>(dim);
15 const double integral_y = (ymax - ymin) / static_cast<double>(dim);

```

Listing 2.34: Constants for Mandelbrot Application

Our algorithm uses these values to determine our Mandelbrot value. The interesting part is the while loop starting on line 20 of Listing 2.35.

```

1 vector<double> mandelbrot(unsigned int start_y, unsigned int end_y)
2 {
3     // Declare values we will use
4     double x, y, x1, y1, xx = 0.0;
5     unsigned int loop_count = 0;
6     // Where to store the results
7     vector<double> results;
8
9     // Loop through each line

```

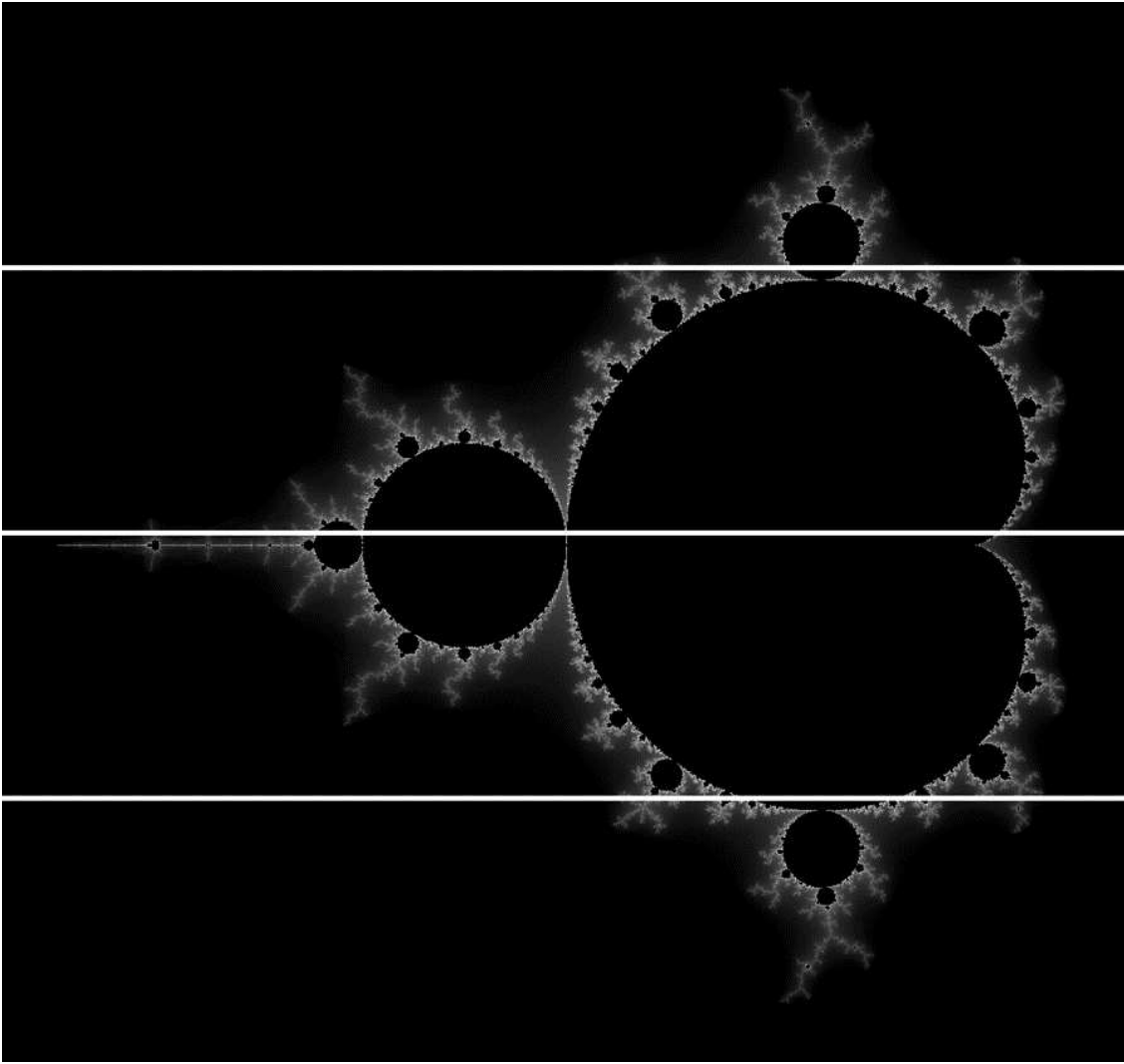


Figure 2.9: Split Mandelbrot Fractal

```

10 | y = ymin + (start_y * integral_y);
11 | for (unsigned int y_coord = start_y; y_coord < end_y; ++y_coord)
12 | {
13 |     x = xmin;
14 |     // Loop through each pixel on the line
15 |     for (unsigned int x_coord = 0; x_coord < dim; ++x_coord)
16 |     {
17 |         x1 = 0.0, y1 = 0.0;
18 |         loop_count = 0;
19 |         // Calculate Mandelbrot value
20 |         while (loop_count < max_iterations && sqrt((x1 * x1) + (y1 *
21 |             y1)) < 2.0)
22 |         {
23 |             ++loop_count;
24 |             xx = (x1 * x1) - (y1 * y1) + x;
25 |             y1 = 2 * x1 * y1 + y;
26 |             x1 = xx;
27 |         }
28 |         // Get value where loop completed
29 |         auto val = static_cast<double>(loop_count) / static_cast<
30 |             double>(max_iterations);
31 |         // Push this value onto the vector
32 |         results.push_back(val);

```



```

31     // Increase x based on integral
32     x += integral_x;
33 }
34 // Increase y based on integral
35 y += integral_y;
36 }
37 // Return vector
38 return results;
39 }

```

Listing 2.35: Mandelbrot Algorithm

The Mandelbrot set works on real numbers in the range $[-2.1, 1.0]$ on the x-dimension and $[-1.3, 1.3]$ on the y-dimension. Each individual pixel is converted to be within this range (the x and y values in the algorithm). Based on the x and y value, the loop runs up to `max_iterations` times. We then use the number of iterations to determine the escape factor of the loop (line 28) to determine the individual pixel value, pushing this onto the vector.

To store the results, our main application must create a number of futures, and then gather the results. We will create `num_threads` futures in this example. The main application is given in Listing 2.36.

```

1 int main()
2 {
3     // Get the number of supported threads
4     auto num_threads = thread::hardware_concurrency();
5
6     // Determine strip height
7     auto strip_height = dim / num_threads;
8
9     // Create futures
10    vector<future<vector<double>>> futures;
11    for (unsigned int i = 0; i < num_threads; ++i)
12        // Range is used to determine number of values to process
13        futures.push_back(async(mandelbrot, i * strip_height, (i + 1) *
14                                strip_height));
15
16    // Vector to store results
17    vector<vector<double>> results;
18    // Get results
19    for (auto &f : futures)
20        results.push_back(f.get());
21
22    return 0;
23 }

```

Listing 2.36: Mandelbrot Main Application

Again, this follows our standard approach to spreading work across our CPU. This application will take a bit of time to complete, so be patient. To check your result, you will have to save the image (see the exercises).

2.11 Exercises Part 2

1. The atomic version of increment runs significantly faster than the mutex version of increment. Why is this? What does it tell you about atomic int values on standard PC CPUs?

2. You should now be able to use atomics to perform the Monte Carlo π experiment. This should give you a performance increase over using standard locks. Produce another table (as in Exercise 2 of the Part 1 Exercises of this Unit) and compare the times with the lock and atomic versions.
3. You should also be able to create a version of Monte Carlo π using futures. Do this as well, and again gather some timing information. Now create some charts analysing the different application runtimes for Monte Carlo π using the different techniques, different thread configurations and different iteration values (problem size). Can you draw any definite conclusions from your data?
4. To show the result from the Mandelbrot experiment, use an image saving library (such as freeimage) to convert your result into an image. You should produce something like this:
5. Gather timing information for splitting the Mandelbrot problem. Experiment with different thread configurations and different methods of splitting up the work. Create charts to present your results.
6. You will probably find that 8192×8192 is the largest Mandelbrot image resolution you can create due to working memory limitations. You can however split the task further by generating separate images. Try and create a very high resolution (e.g. 65536×65536) image and zoom in to see the smaller details of the fractal.

2.11.1 Challenge - A Synchronous Channel

For this week's challenge your goal is to create another concurrency construct not supported directly by C++11 - a synchronous communication channel. This is a construct where a thread can send a message (some type of data) to another thread. If the receiving thread is not ready to receive when the sender sends, the sending thread must wait. If the sending thread is not ready to send when the receiver tries to receive, the receiving thread must also wait. For those of you who have undertaken the Fundamentals of Parallel Systems module, you should understand what a channel is.

The source code for JCSP (a Java library supporting such constructs) is freely available online. However it does more than just provide channels so you will have to work out what is happening internally if you want to extract just the channel functionality.

2.12 Reading

This week you should be reading the following:

- C++ Concurrency in Action - chapter 3, 4 and 5.
- An Introduction to Parallel Programming - chapters 1 and 2. Chapter 4 (using pthreads) gives some insight into multithreaded code considerations.

Unit 3

OpenMP

We are now going to move away from using basic C++11 concurrency concepts and move into using a particular framework to support multi-processing - OpenMP. OpenMP is API that supports shared-memory parallel programming, so can enable us to work with our CPU as a multi-core device. OpenMP provides some different constructs to normal C++11 concurrency, and is itself quite a mature platform. We will look at some example applications using OpenMP that investigates these constructs before doing some analysis work.

3.1 First OpenMP Application

Before building an OpenMP application, you need to make sure that your compiler supports it. In Visual Studio you will find an OpenMP option in the project properties (under **C++** → **Language**). You need to enable OpenMP here to ensure that our compiled application is using OpenMP.

Our first OpenMP application is going to be a very trivial Hello World example. The whole code is given in Listing 3.1.

```
1 #include <iostream>
2 #include <omp.h>
3
4 using namespace std;
5
6 // Number of threads to run
7 const int THREADS = 10;
8
9 void hello()
10 {
11     // Get the thread number
12     auto my_rank = omp_get_thread_num();
13     // Get the number of threads in operation
14     auto thread_count = omp_get_num_threads();
15     // Display a message
16     cout << "Hello from thread " << my_rank << " of " << thread_count
17         << endl;
18 }
19
20 int main()
21 {
22     // Run hello THREADS times
23     #pragma omp parallel num_threads(THREADS)
24     hello();
```

```

25|     return 0;
26| }

```

Listing 3.1: Hello OpenMP

The first thing to note is the use of the OpenMP header (`omp.h`) on line 2. You will need this for some of the OpenMP functions we use, such as getting the thread number (line 12) and the number of threads (line 14).

`hello` is our operation we are running multiple times. Line 22 shows how we do this. We are using a pre-processor to tell the compiler that OpenMP code should be generated here. We are running the operation in `parallel`, and using `num_threads` to tell OpenMP how many copies to run. If you run this application you should get the output shown in Figure 3.1.

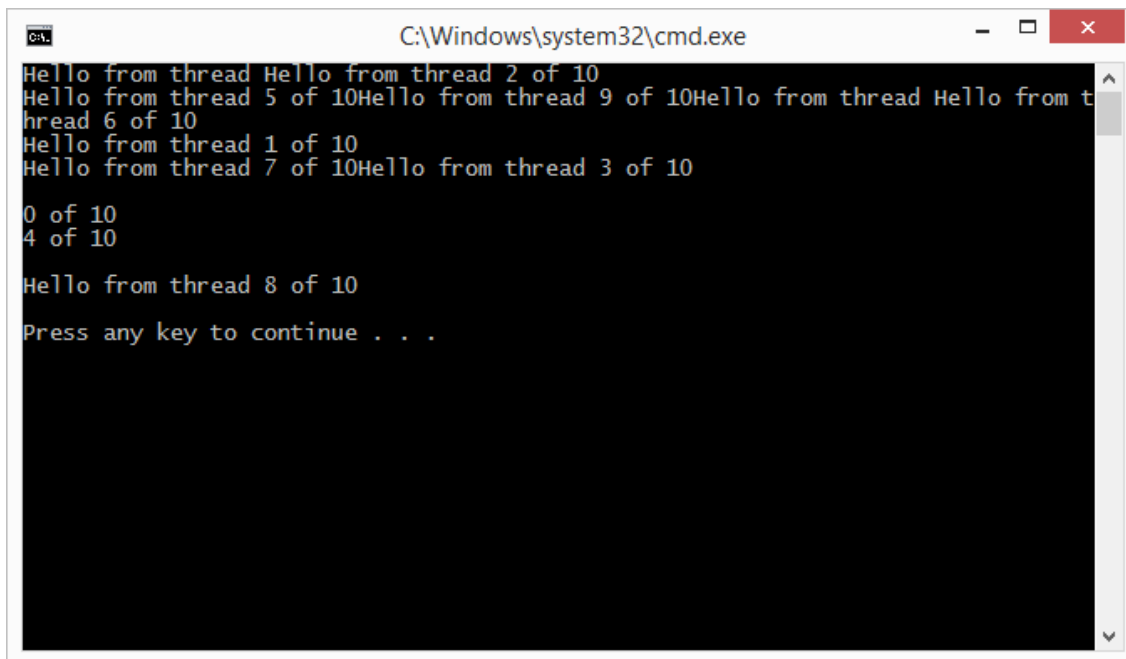


Figure 3.1: Output from Hello OpenMP Application

As we are not controlling the output, there is some conflict with the threads trying to output at the same time.

3.2 parallel for

OpenMP tries to extract away the idea of having threads. They still exist but they are hidden from the application developer. One of the powerful constructs OpenMP provides is `parallel for`. This allows us to create threads by executing a `for` loop. Each iteration uses a thread to compute thus providing speedup. You have to think a little when using `parallel for` but it can be useful.

3.2.1 Calculating π (not using Monte Carlo Simulation)

Using Monte Carlo simulation to calculate π is great for testing performance and speedup, but it is not really the most efficient method of calculating π . A better method to approximate π is to use the following formula:

$$4 \sum_{k=0}^{\infty} \frac{-1^k}{(2k+1)} = 4(1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \dots)$$

We are going to use this method in a `parallel for` to see how we can calculate π .

3.2.2 Parallel For Implementation of π Approximation

Listing 3.2 shows how we implement this in OpenMP using a `parallel for`.

```

1 int main()
2 {
3     // Get number of supported threads
4     auto num_threads = thread::hardware_concurrency();
5
6     // Number of iteration values to perform
7     const int n = static_cast<int>(pow(2, 30));
8     // Factor value
9     double factor = 0.0;
10    // Calculated pi
11    double pi = 0.0;
12
13    // Parallelised for loop that does the work
14    #pragma omp parallel for num_threads(num_threads) reduction(+:pi)
15        private(factor)
16        for (int k = 0; k < n; ++k)
17        {
18            // Determine sign of factor
19            if (k % 2 == 0)
20                factor = 1.0;
21            else
22                factor = -1.0;
23            // Add this iteration value to pi sum
24            pi += factor / (2.0 * k + 1);
25        }
26
27    // Get the final value of pi
28    pi *= 4.0;
29
30    // Show more percision of pi
31    cout.precision(numeric_limits<double>::digits10);
32    cout << "pi = " << pi << endl;
33
34    return 0;
35 }
```

Listing 3.2: Calculating π With `parallel for`

OK, we are using quite a bit of new ideas in the pre-processor comment. First of all, the general `parallel for` looks very similar to a standard `parallel` but with the keyword `for` added. The two new parts are at the end of the pre-processor:

reduction we will be covering reduction in more detail when we look at map-reduce in MPI later in the module. What we are saying here is that addition on the `pi` variable should be controlled to add all the `for` loops together.

private this indicates that each `for` loop has a private copy of the `factor` value. Each `for` loop can modify the value independently and not cause corruption to another `for` loop.

If you run this application you will get the output (accurate to 10 decimal places) shown in Figure 3.2.

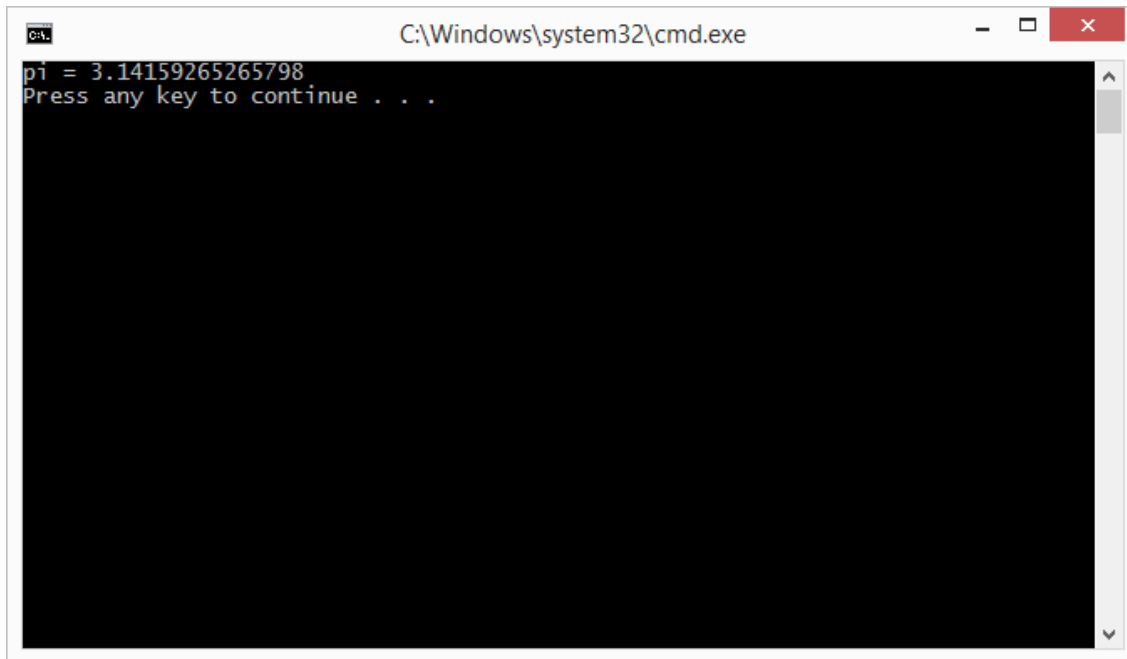


Figure 3.2: Output from `parallel` for π Calculation

3.3 Bubble Sort

We are now going to diverge for a bit and look at sequential sorting using a bubble sort. The reason we are doing this is because we are going to build a parallel sorting mechanism and then compare the performance. You should hopefully all be familiar with what is meant by a bubble sort by now.

We are going to build two new functions to support our application:

1. `generate_values`
2. `bubble_sort`

Our `parallel_sort` will also use value generation

3.3.1 `generate_values`

Listing 3.3 is the function that will generate a vector full of values. It simply uses a random engine to do this.

```

1 // Generates a vector of random values
2 vector<unsigned int> generate_values(unsigned int size)
3 {
4     // Create random engine
5     auto millis = duration_cast<milliseconds>(system_clock::now().
        time_since_epoch());
6     default_random_engine e(static_cast<unsigned int>(millis.count()));
7     // Generate random numbers
8     vector<unsigned int> data;
```

```

9 |   for (unsigned int i = 0; i < size; ++i)
10 |       data.push_back(e());
11 |
12 |   return data;
13 | }

```

Listing 3.3: generate_values Function

3.3.2 bubble_sort

Bubble sort is a straight forward algorithm and is shown in Algorithm 2. We bubble up through the values, swapping them as we go to move a value towards the top. You should be able to implement this algorithm in C++ by now.

Algorithm 2 Bubble Sort Algorithm

```

var values
begin
  for count := values.size() to 2 step -1 do
    for i := 0 to count - 1 step 1 do
      if values[i] > values[i + 1]
        then
          tmp := values[i]
          values[i] := values[i + 1]
          values[i + 1] := tmp
        fi
      od
    od
  end

```

3.3.3 Main Application

Our main application will time the implementation of bubble_sort using vectors of different sizes. The main application is shown in Listing 3.4.

```

1 | int main()
2 | {
3 |     // Create results file
4 |     ofstream results("bubble.csv", ofstream::out);
5 |     // Gather results for 2^8 to 2^16 results
6 |     for (unsigned int size = 8; size <= 16; ++size)
7 |     {
8 |         // Output data size
9 |         results << pow(2, size) << ", ";
10 |        // Gather 100 results
11 |        for (unsigned int i = 0; i < 100; ++i)
12 |        {
13 |            // Generate vector of random values
14 |            cout << "Generating " << i << " for " << pow(2, size) << "
15 |                << " values" << endl;
16 |            auto data = generate_values(static_cast<unsigned int>(pow(2,
17 |                size)));
18 |            // Sort the vector
19 |            cout << "Sorting" << endl;

```

```

18     auto start = system_clock::now();
19     bubble_sort(data);
20     auto end = system_clock::now();
21     auto total = duration_cast<milliseconds>(end - start).count()
22     ;
23     // Output time
24     results << total << ", ";
25 }
26 results << endl;
27 }
28 results.close();
29 return 0;
30 }

```

Listing 3.4: Bubble Sort Main Application

If you run this application you should be able to produce a graph as shown in Figure 3.3. However, you should change the y-axis scale to use a \log_2 scale, as shown in Figure 3.4. This gives a nice straight line.

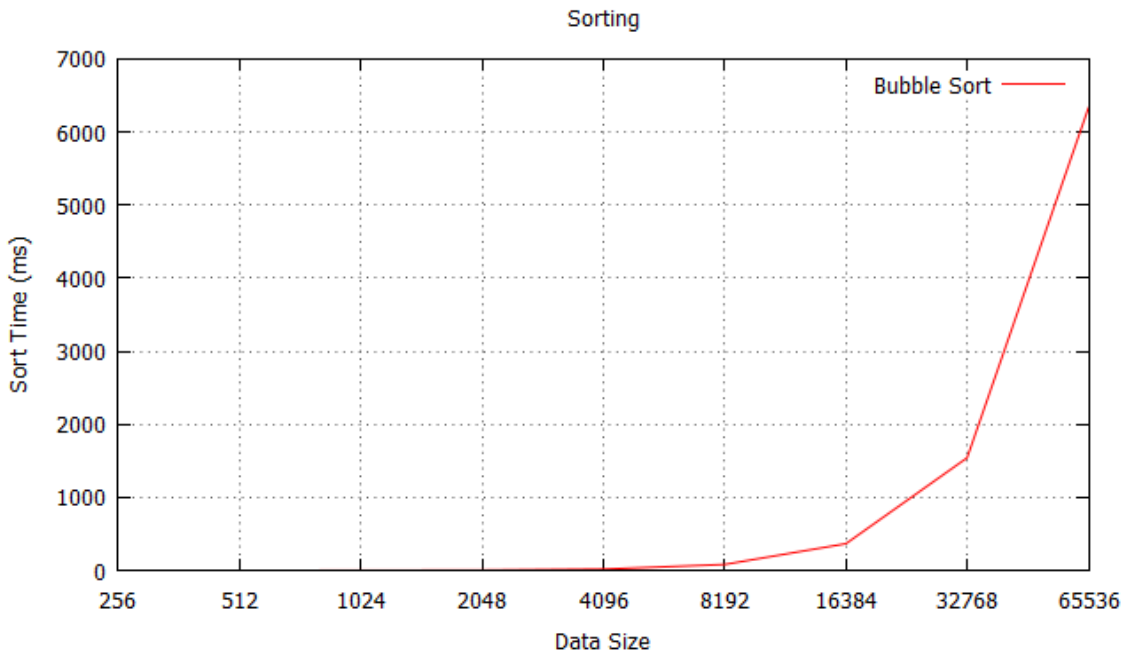


Figure 3.3: Bubble Sort Performance (no Log Scale)

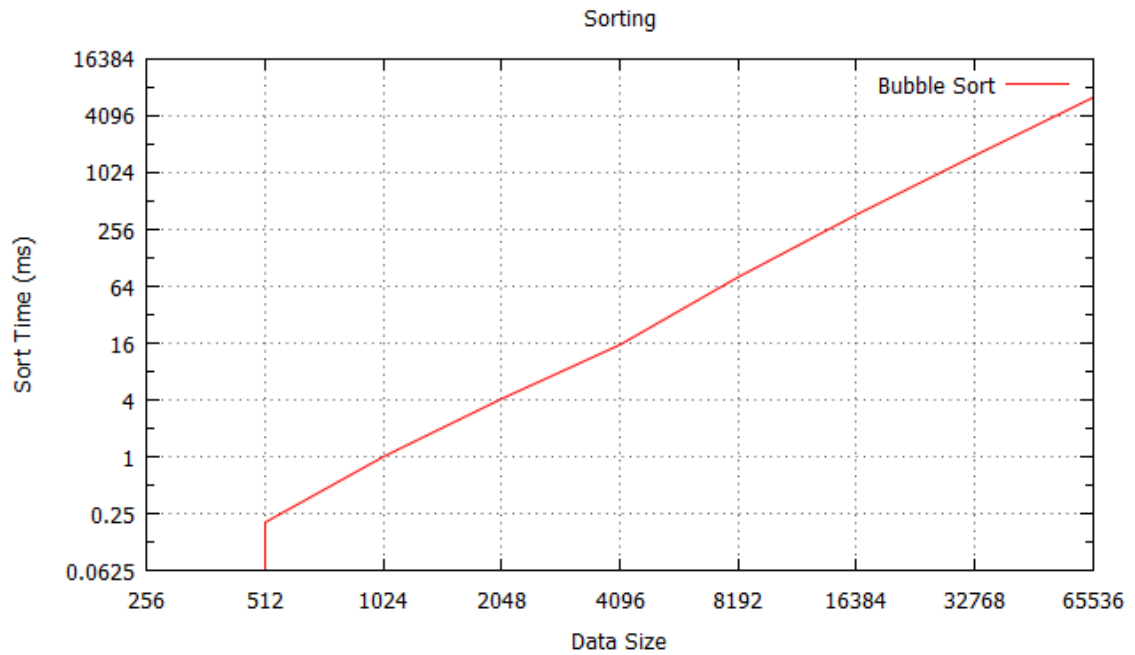
3.4 Parallel Sort

All we need to do for a `parallel_for` is change one method the sort. Listing 3.5 is a `parallel_sort` taken from Introduction to Parallel Programming. An in-depth description of its development is available in the book (in the OpenMP section).

```

1 void parallel_sort(vector<unsigned int>& values)
2 {
3     // Get the number of threads
4     auto num_threads = thread::hardware_concurrency();
5     // Get the number of elements in the vector
6     auto n = values.size();

```


Figure 3.4: Bubble Sort Performance (y-axis \log_2 Scale)

```

7 // Declare the variables used in the loop
8 int i, tmp, phase;
9 // Declare parallel section
10 #pragma omp parallel num_threads(num_threads) default(none) shared(
    values, n) private(i, tmp, phase)
11 for (phase = 0; phase < n; ++phase)
12 {
13     // Determine which phase of the sort we are in
14     if (phase % 2 == 0)
15     {
16         // Parallel for loop. Each thread jumps forward 2 so no
            conflict
17 #pragma omp for
18         for (i = 1; i < n; i += 2)
19         {
20             // Check if we should swap values
21             if (values[i - 1] > values[i])
22             {
23                 // Swap values
24                 tmp = values[i - 1];
25                 values[i - 1] = values[i];
26                 values[i] = tmp;
27             }
28         }
29     }
30     else
31     {
32         // Parallel for loop. Each thread jumps forward 2 so no
            conflict
33 #pragma omp for
34         for (i = 1; i < n; i += 2)
35         {
36             // Check is we should swap values
37             if (values[i] > values[i + 1])
38             {

```

```

39      // Swap values
40      tmp = values[i + 1];
41      values[i + 1] = values[i];
42      values[i] = tmp;
43  }
44  }
45  }
46  }
47  }

```

Listing 3.5: parallel_sort Function

Of more interest is the results. Figure 3.5 is a graph combining the results from Bubble Sort and Parallel Sort with no logarithmic scaling, and Figure 3.6 provides the results with a \log_2 scale.

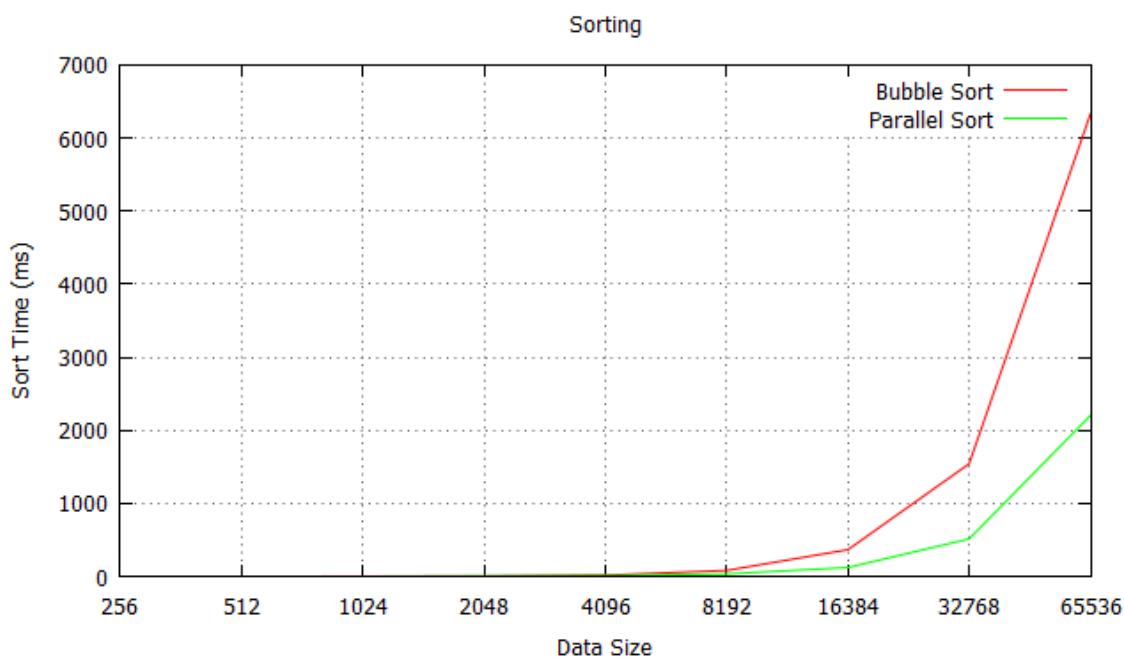


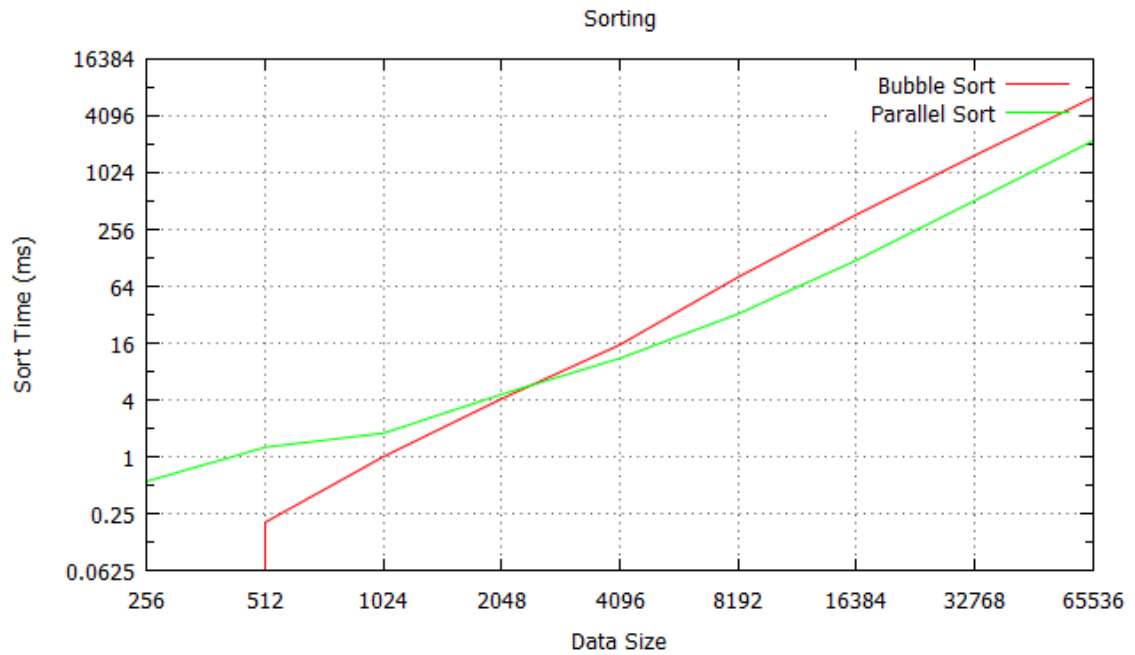
Figure 3.5: Parallel Sort Performance (no Log Scale)

Notice that for small vector sizes throwing parallelism at the problem has not given us a performance boost - in fact we are slower. Granted we are using a slightly different algorithm, but hopefully you can see that the problem set is too small to get any speed up - in fact the setup and control of the OpenMP program is having an effect. Once our sort space is large enough we gain performance - 3+ times as much (the CPU is dual core with 4 hardware threads so this seems reasonable).

3.5 The Trapezoidal Rule

Our next use of OpenMP will look at something called the trapezoidal rule. This technique can be used to approximate the area under a curve. It operates as shown in Figure 3.7.

We select a number of points on the curve and measure their value. We then use this to generate a number of trapezoids. We can then calculate the area of the trapezoids and get an approximate value for the area under the curve. The more points we use on the curve, the better the result.

Figure 3.6: Parallel Sort Performance (y-axis \log_2 Scale)

For our purposes we do not need to worry about why we want to do this - the point is we can parallelise the problem by calculating more trapezoids.

3.5.1 Trapezoidal Function

Listing 3.6 provides the function to work out a section of the area under a curve using the trapezoidal rule.

```

1 void trap(function<double(double)> f, double start, double end,
2   unsigned int iterations, shared_ptr<double> p)
3 {
4   // Get thread number
5   auto my_rank = omp_get_thread_num();
6   // Get number of threads
7   auto thread_count = omp_get_num_threads();
8   // Calculation iteration slice size
9   auto slice_size = (end - start) / iterations;
10  // Calculate number of iterations per thread
11  auto iterations_thread = iterations / thread_count;
12  // Calculate this thread's start point
13  auto local_start = start + ((my_rank * iterations_thread) *
14    slice_size);
15  // Calculate this thread's end point
16  auto local_end = local_start + iterations_thread * slice_size;
17  // Calculate initial result
18  auto my_result = (f(local_start) + f(local_end)) / 2.0;
19
20  // Declare x before the loop - stops it being allocated and
21  // destroyed each iteration
22  double x;
23  // Sum each iteration
24  for (unsigned int i = 0; i <= iterations_thread - 1; ++i)
25  {
26    // Calculate next slice to calculate
27    x = local_start + i * slice_size;

```

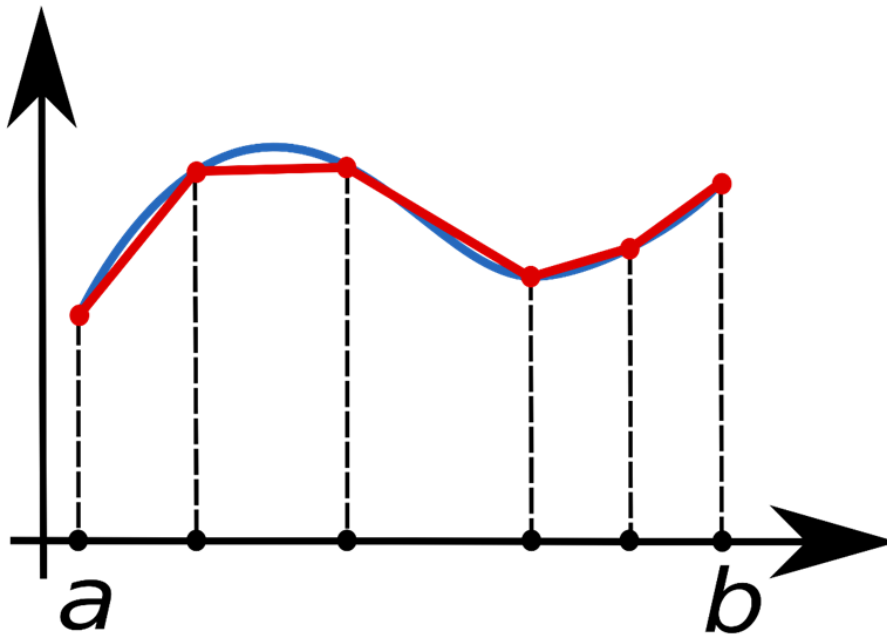


Figure 3.7: Trapezoidal Rule

```

25     // Add to current result
26     my_result += f(x);
27 }
28 // Multiply the result by the slice size
29 my_result *= slice_size;
30
31 // Critical section - add to the shared data
32 #pragma omp critical
33 *p += my_result;
34 }

```

Listing 3.6: trap Function for Trapezoidal Rule

The incoming parameters are as follows:

f the function we are using to generate the curve

start the starting value we will place in the function

end the end value we will place in the function

iterations the number of iterations (or trapezoids) we will generate

p a shared piece of data to store the result

You should be able to follow the algorithm using the comments. The new part we have introduced from OpenMP is line 32 - a **critical** section. A **critical** section is just a piece of code that only one thread can access at a time - it is controlled by a mutex. We use the **critical** section to control the adding of the local result to the global result.

3.5.2 Testing the Trapezoidal Algorithm

There is a simple test we can perform to check our algorithm using the standard trigonometric functions. For example, the cosine function is shown in Figure 3.8.

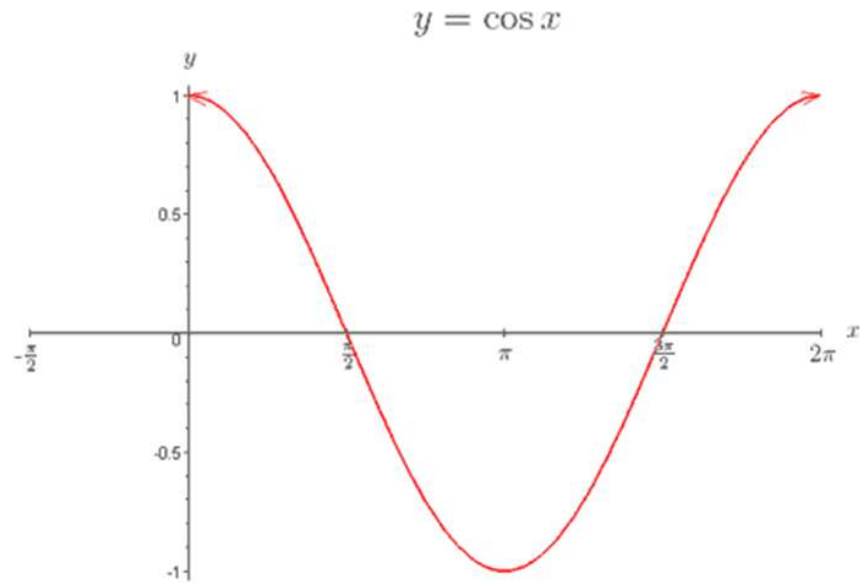


Figure 3.8: Cosine Function

The area under the curve between 0 and π radians should equal 0 — it is equal parts above and below the line over this period. For a sine function we have that shown in Figure 3.9.

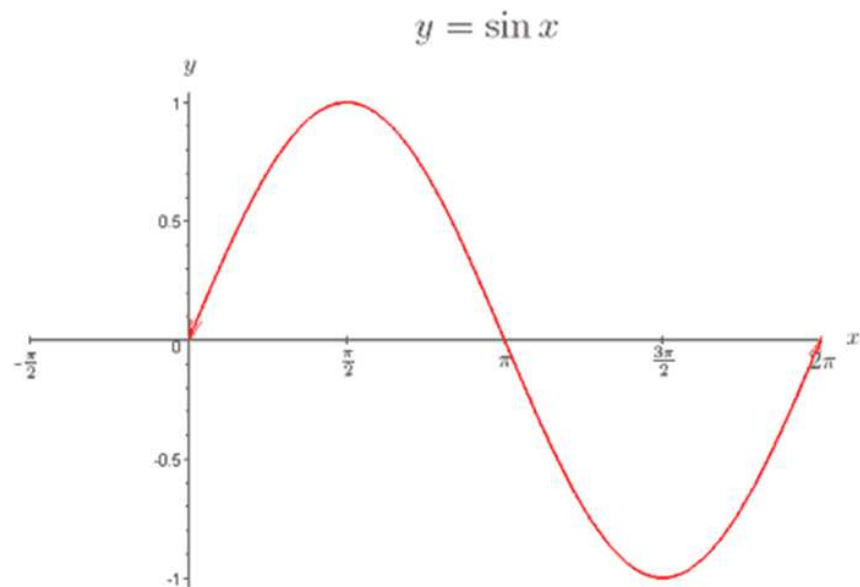


Figure 3.9: Sine Function

The area under the curve here is 2. Let us first test the cosine function. This is shown in Listing 3.7.

```

1 int main()
2 {
3     // Declare shared result
4     auto result = make_shared<double>(0.0);
5     // Define start and end values

```

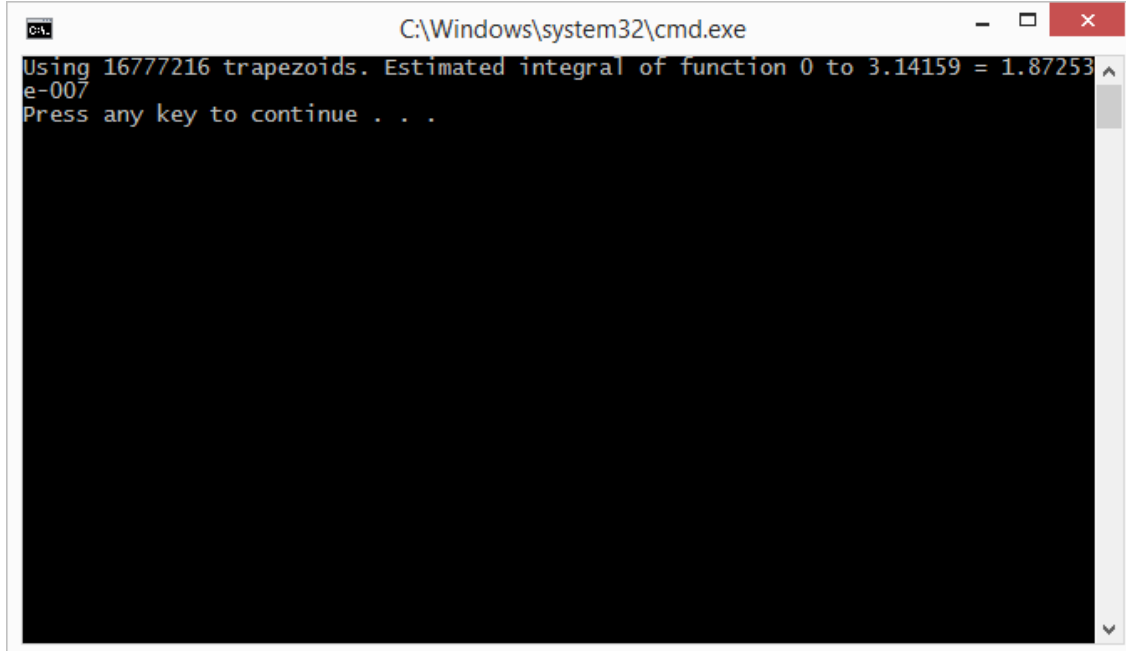
```

6   auto start = 0.0;
7   auto end = PI;
8   // Defined number of trapezoids to generation
9   unsigned int trapezoids = static_cast<unsigned int>(pow(2, 24));
10  // Get number of threads
11  auto thread_count = thread::hardware_concurrency();
12
13  // Create function to calculate integral of. Use cos
14  auto f = [](double x){ return cos(x); };
15
16  // Run trap in parallel
17 #pragma omp parallel num_threads(thread_count)
18   trap(f, start, end, trapezoids, result);
19
20  // Output result
21  cout << "Using " << trapezoids << " trapezoids. ";
22  cout << "Estimated integral of function " << start << " to " <<
    end << " = " << *result << endl;
23
24  return 0;
25 }

```

Listing 3.7: Trapezoidal Rule Main Function

We set our function as a λ expression on line 14 and pass this into our trap algorithm on line 18. If you run this you will get the output shown in Figure 3.10.



```

C:\Windows\system32\cmd.exe
Using 16777216 trapezoids. Estimated integral of function 0 to 3.14159 = 1.87253
e-007
Press any key to continue . . .

```

Figure 3.10: Output from Trapezoidal Rule Application Using Cosine Function

So our result is 0.000000187253 or pretty close to 0 for an estimate with rounding errors. If you change the application to use the sine function we get the output shown in Figure 3.11.

Which is the answer we expect.

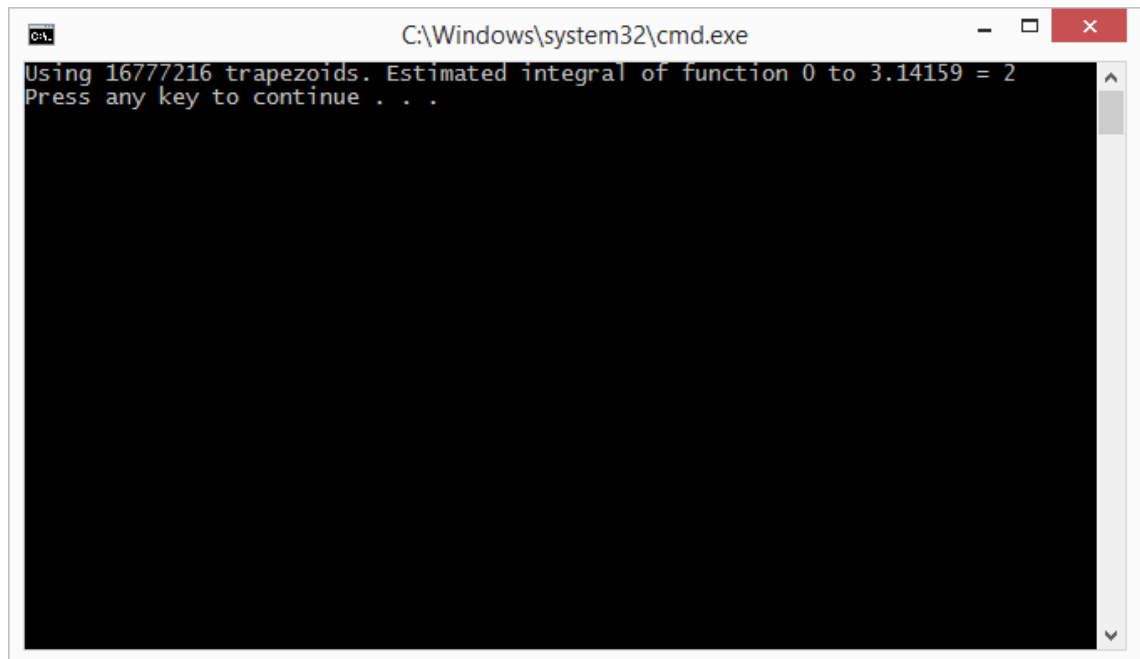


Figure 3.11: Output from Trapezoidal Rule Application Using Sine Function

3.6 Scheduling

Our next concept we will introduce is the idea of scheduling our work in OpenMP. Scheduling involves us telling OpenMP how to divide up the work in a `parallel for`. At the moment, each thread is given a chunk of work in order. For example, if we have 1024 iterations and we have 4 threads, our work is divided as follows:

Thread 1 iterations 0 to 255

Thread 2 iterations 256 to 511

Thread 3 iterations 512 to 767

Thread 4 iterations 768 to 1023

For many problems that divide simply, this works ideally. However, many problems do not divide like this. Scheduling in OpenMP allows us to divide up our work in different manners. Chapter 5 of Introduction to Parallel Programming gives more details.

The scheduling method we will use is called `static`. This allows us to allocate work to threads in a round robin manner. For example, a schedule of 1 allocates the work to thread in blocks of 1:

Thread 1 0, 4, 8, 12, ...

Thread 2 1, 5, 9, 13, ...

Thread 3 2, 6, 10, 14, ...

Thread 4 3, 7, 11, 15, ...

Using a schedule of 2 allocates work to threads in blocks of 2:

Thread 1 0, 1, 8, 9, ...

Thread 2 2, 3, 10, 11, ...

Thread 3 4, 5, 12, 13, ...

Thread 4 6, 7, 14, 15, ...

And so on.

3.6.1 Test Function

Listing 3.8 is a function that can test the effect of scheduling for us. It runs based on the value of `i` passed in.

```

1 // Let's create a function that relies on i to determine the amount
  of work
2 double f(unsigned int i)
3 {
4     // Calculate start and end values
5     auto start = i * (i + 1) / 2;
6     auto end = start + i;
7     // Declare return value
8     auto result = 0.0;
9
10    // Loop for number of iterations, calculating sin
11    for (auto j = start; j <= end; ++j)
12        result += sin(j);
13
14    // Return result
15    return result;
16 }
```

Listing 3.8: Test Function for schedule

3.6.2 Main Application

Listing 3.9 is our test application. We use the `schedule` function in the pre-processor argument to control the division of work. Your task here is to manipulate the `schedule` value and see the effect.

```

1 int main()
2 {
3     // Get number of hardware threads
4     auto thread_count = thread::hardware_concurrency();
5     // Define number of iterations to calculate
6     int n = static_cast<int>(pow(2, 14));
7     // Declare sum value
8     auto sum = 0.0;
9
10    // Get start time
11    auto start = system_clock::now();
12    #pragma omp parallel for num_threads(thread_count) reduction(+:sum)
      schedule(static, 1)
13    for (auto i = 0; i <= n; ++i)
14        sum += f(i);
15    // Get end time
16    auto end = system_clock::now();
17    // Calculate and output total time
18    auto total = duration_cast<milliseconds>(end - start).count();
19    cout << "Total time: " << total << "ms" << endl;
```



```

20|
21|     return 0;
22| }

```

Listing 3.9: Testing `schedule`

Running this application will output a timing value - test the scheduling value and chart the difference in performance.

3.7 Concurrency Visualizer

We will end this practical by looking at the **Concurrency Visualizer** in Visual Studio. In Visual Studio 2013 you will need to install the Concurrency Visualizer via **Extensions and Updates**. To run the Concurrency Visualizer select **Analyse** in the Visual Studio menu and select Concurrency Visualizer. To run for the current project, select **Current Project**.

Once run, you will get a report. Figure 3.12 is the first view you will see.

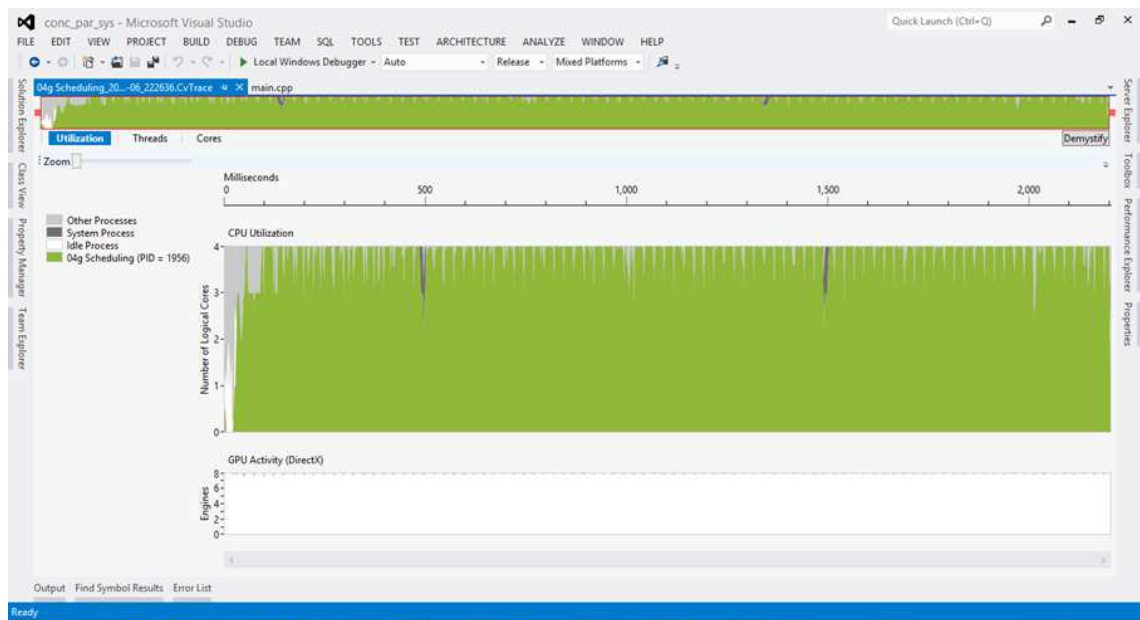


Figure 3.12: Concurrency Visualizer Overview

This view looks at the utilization of your process across the processor. If you click on **Threads** you can view the amount of work a thread does as shown in Figure 3.13.

Of particular interest is the amount of work the threads are doing - shown by the coloured blocks. Notice that in the example 4% of the time was spent on pre-emption. This means that 4% of our applications runtime was taken up by the threads being switched. Not too bad, but illustrates the problem with pre-emptive scheduling.

Selecting the **Cores** view allows us to view how our work was divided amongst the cores as shown in Figure 3.14.

Our threads are being switched across the four logical cores - which will lead to some of the pre-emption. Logical Core 3 looks like it has the best division of work (long chunks of work). Playing around with the schedule clause could improve things.

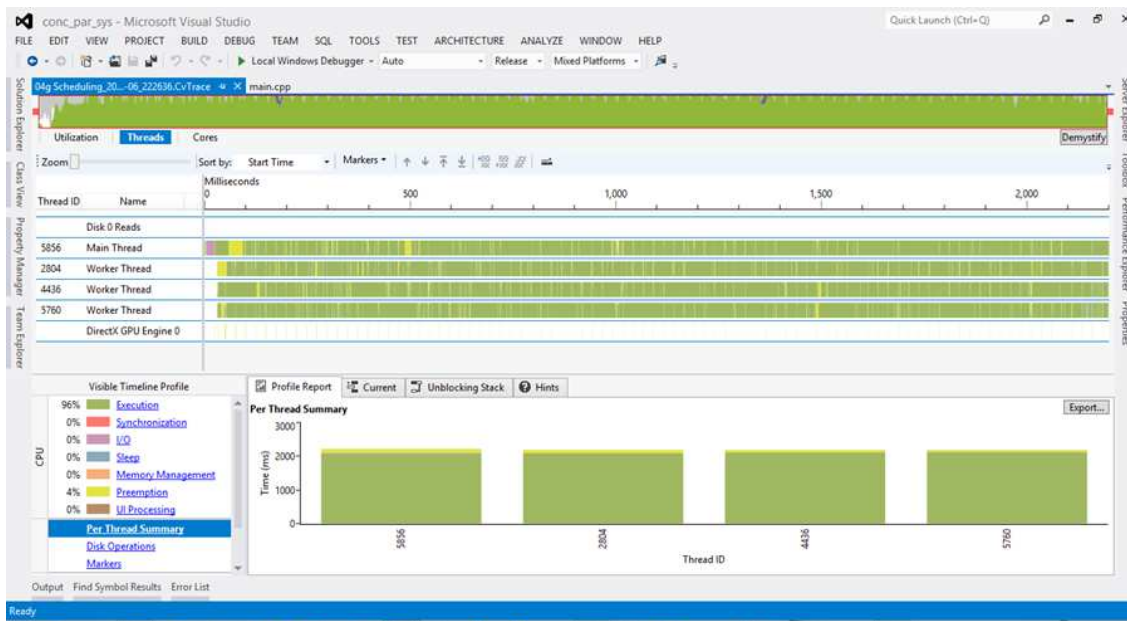


Figure 3.13: Concurrency Visualizer Threads View

3.8 Exercises

1. Try using the schedule technique to split up the work for the Mandelbort fractal this will allow you even more control over how the work is divided and should enable some speedup. You need to understand where the likely bottlenecks are in the algorithm in relation to the image produced to work out how best to split it up using OpenMP.
2. You now have enough information to build a queue to act as a message passing interface. Build one either using standard C++11 threading or OpenMP or both and show it works using a basic producer-consumer model.
3. Experiment with the Concurrency Visualizer. You have enough applications now to really explore what is happening. Try and create hundreds of threads to see how the pre-emption can change.

3.9 Reading

You should be reading Chapter 5 of Introduction to Parallel Programming for more information on OpenMP.

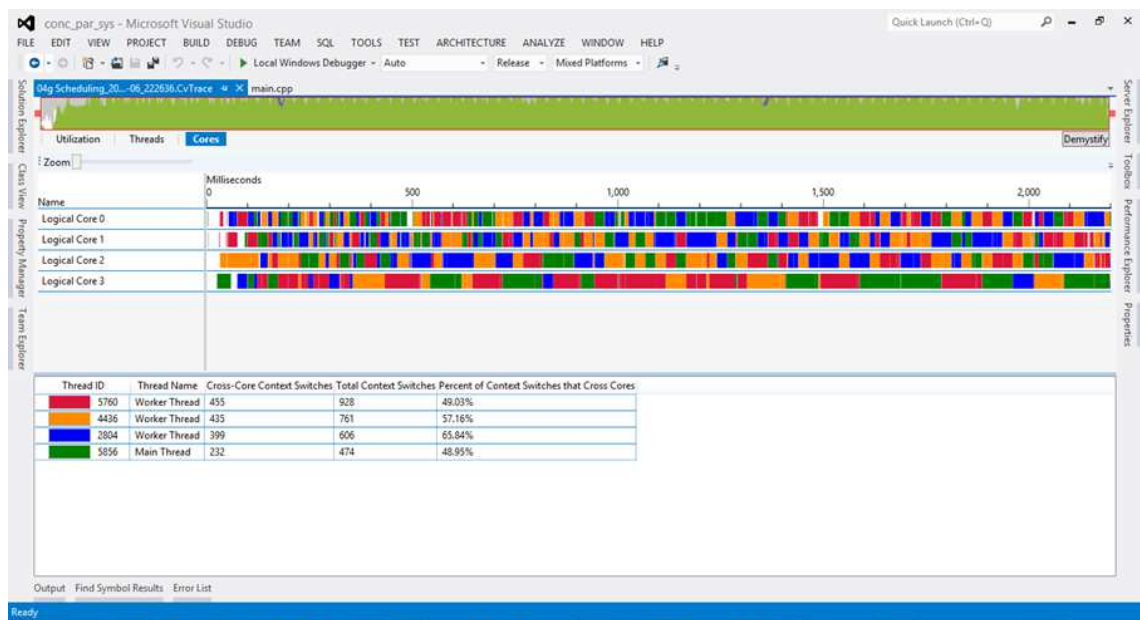


Figure 3.14: Concurrency Visualizer Cores View

Unit 4

CPU Instructions

We are now going to go in the complete opposite direction from the last practical and go very low level. We are going to look at how we can get the CPU to utilise its internal 128-bit registers to calculate values for us more efficiently.

A word of warning! The techniques we are going to use are supported on Intel and AMD based hardware, and some of the approaches are specific to Microsoft's C++ compiler. Visual Studio also turns on certain optimisations by default - you will need to disable them to see a difference in performance (Visual Studio enables some automatic usage of the techniques we are going to use). You can find the setting in the project properties in the C++ section, under **Optimization**. You will want to disable optimization. Otherwise you might not see a performance difference.

4.1 Memory Alignment

Our first application will illustrate how we can align memory correctly in C++ (in particular Microsoft C++). The code for the first application is given in Listing 4.1.

```
1 int main()
2 {
3     // Declare a single 128-bit value aligned to 16 bytes (size of
4     // 128-bits)
5     __declspec(align(16)) __m128 x;
6     // We can treat x as a collection of four floats
7     // Or other combinations of values for 128-bits
8     x.m128_f32[0] = 10.0f;
9     x.m128_f32[1] = 20.0f;
10    x.m128_f32[2] = 30.0f;
11    x.m128_f32[3] = 40.0f;
12
13    // We can print out individual values
14    cout << x.m128_f32[0] << endl;
15
16    // The key is that the memory is aligned - it is faster to access
17    // in blocks
18
19    // Create an array of SIZE floats aligned to 4 bytes (size of a
20    // float)
21    float* data = (float*)_aligned_malloc(SIZE * sizeof(float), 4);
22
23    // Access just like an array
24    cout << data[0] << endl;
25
26    // Create an array of SIZE 128-bit values aligned to 16 bytes
```

```

24  __m128* big_data = (__m128*)_aligned_malloc(SIZE * sizeof(__m128)
    , 16);
25
26  // Access just like an array of __m128
27  cout << big_data[0].m128_f32[0] << endl;
28
29  // Free the data - ALWAYS REMEMBER TO FREE YOUR MEMORY
30  // We are dealing at a C level here
31  _aligned_free(data);
32  _aligned_free(big_data);
33
34  return 0;
35 }

```

Listing 4.1: Declaring Aligned Memory in Microsoft Visual C++

We have a few new ideas going on here. Firstly on line 4 we declare a new type of variable - one of type `__m128`. This means that the value takes up 128-bits of memory. Notice the declaration at the start. This is used to ensure that the data is aligned in memory in a 16 byte block - or 128-bits. This is our first foray into having memory aligned data.

On lines 7 to 10 we see that we can access the 128-bit value as a collection of floats. This can be useful, although we will just treat it as a pointer to float data.

Another method to create memory aligned data is to use the `_aligned_malloc` function (`malloc` is short for memory allocation). We see this on line 18 where we create a `float` using the aligned value.

We can also use this technique to create an array of memory aligned 128-bit values. The `SIZE` value can be set to any number for the size of our array. This is what we do on line 24.

Finally any malloced memory must be freed using the `_align_free` function. This will free the memory.

Running this application will not do much - it will print out a couple of values. The point is to introduce to the methods of allocating memory aligned data. We are going quite low level here, so are working on a C based level.

4.2 SIMD Operations

Allocating memory is just one step of our work with the processor. The next stage is to use SIMD based operations. There are quite a few - you can find a description via MSDN [http://msdn.microsoft.com/en-us/library/x5c07e2a\(v=vs.100\).aspx](http://msdn.microsoft.com/en-us/library/x5c07e2a(v=vs.100).aspx). We will only use a couple of these here for illustration purposes. The rest is up to you.

Our first test application will simply add 4 to a collection of floating point values. We will create an array of floating point values, assign them the value 1.0f and then add 4.0f to them. We will do this using a SIMD operation which will allow us to add to 4 floats at a time (128-bits). The code is given in Listing 4.2.

```

1  // Size of data to allocate - divide by four to get number of
    vectors
2  const unsigned int SIZE = static_cast<unsigned int>(pow(2, 24));
3  const unsigned int NUM_VECTORS = SIZE / 4;
4
5  int main()
6  {
7      // Data - aligned to 16 bytes (128-bits)

```

```

8 | auto data = (float*)_aligned_malloc(SIZE * sizeof(float), 16);
9 | // Initialise data
10 | for (unsigned int i = 0; i < SIZE; ++i)
11 |     // Set all values to 1
12 |     data[i] = 1.0f;
13 |
14 | // Value to add to all values
15 | auto value = _mm_set1_ps(4.0f);
16 | // __m128 pointer to the data
17 | auto stream_data = (__m128*)data;
18 | // Start timer. Use high resolution clock
19 | auto start = high_resolution_clock::now();
20 | // Add value to stream data
21 | for (unsigned int i = 0; i < NUM_VECTORS; ++i)
22 |     stream_data[i] = _mm_add_ps(stream_data[i], value);
23 | // End timer
24 | auto end = high_resolution_clock::now();
25 | // Calculate time and display
26 | auto total = duration_cast<microseconds>(end - start).count();
27 | cout << "SIMD: " << total << "micros" << endl;
28 | // Free memory
29 | _aligned_free(data);
30 |
31 | // Declare standard data
32 | data = new float[SIZE];
33 | // Set all values to 1
34 | for (unsigned int i = 0; i < SIZE; ++i)
35 |     data[i] = 1.0f;
36 |
37 | // Start timer
38 | start = high_resolution_clock::now();
39 | for (unsigned int i = 0; i < SIZE; ++i)
40 |     data[i] = data[i] + 4.0f;
41 | // End timer
42 | end = high_resolution_clock::now();
43 | // Calculate time and display
44 | total = duration_cast<microseconds>(end - start).count();
45 | cout << "Non-SIMD: " << total << "micros" << endl;
46 | // Free memory
47 | delete[] data;
48 |
49 | return 0;
50 | }

```

Listing 4.2: Adding Using SIMD Operations

We use a particular method to add two 4-value vectors at once - `_mm_add_ps`. There are instructions for adding, multiplying, etc. As a comparison, we do the same operation using standard floats. You will need to ensure that optimisation is off when you run the application to get a result similar to that in Figure 4.1.

We are getting a more than 4 times speedup using this approach. This is the power of SIMD operations - we can use them to speed up all sorts of applications.

4.3 Normalizing a Vector

We are now going to use a SIMD approach to normalizing some 4-dimensional vectors - the same technique could be used for 3-dimensional vectors easily enough. To do this we are going to use SIMD instructions to parallelise the process. Algorithm 3

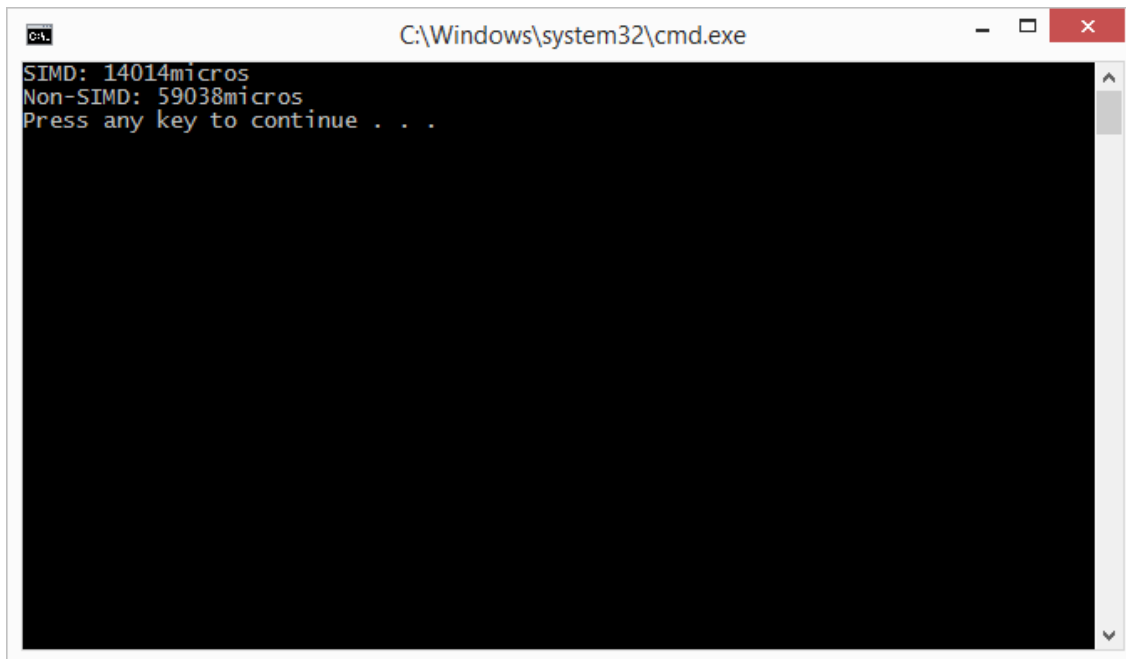


Figure 4.1: Output form SIMD Add Application

provides the pseudocode for this technique.

First of all we are going to generate a collection of random values. This function is given in Listing 4.3.

```

1 // Randomly generate vector values
2 void generate_data(float *data, unsigned int num_values)
3 {
4     // Create random engine
5     auto millis = duration_cast<milliseconds>(system_clock::now().
6         time_since_epoch());
7     default_random_engine e(static_cast<unsigned int>(millis.count()))
8         );
9     // Fill data
10    for (unsigned int i = 0; i < num_values; ++i)
11        data[i] = e();
12 }

```

Listing 4.3: Generate Data for SIMD Normalisation

Our normalization process is provided in Listing 4.4.

```

1 // Normalises the vector
2 void normalise_vectors(__m128 *data, __m128 *result, unsigned int
3     num_vectors)
4 {
5     // Normalise the vectors
6     for (unsigned int i = 0; i < num_vectors; ++i)
7     {
8         // Square each component - simply multiply the vectors by
9         // themselves
10        result[i] = _mm_mul_ps(data[i], data[i]);
11        // Calculate sum of the components. Store in all
12        result[i].m128_f32[0] = result[i].m128_f32[1] = result[i].
13            m128_f32[2] = result[i].m128_f32[3] =
14            result[i].m128_f32[0] + result[i].m128_f32[1] + result[i].
15            m128_f32[2] + result[i].m128_f32[3];
16        // Calculate reciprical square root of the values

```


Algorithm 3 SIMD Vector Normalization

```

var  $v, r$ 
begin
    comment: Square each component - single SIMD operation
     $r_x := v_x^2$ 
     $r_y := v_y^2$ 
     $r_z := v_z^2$ 
     $r_w := v_w^2$ 
    comment: Sum the components - store in each component
     $r_x := r_x + r_y + r_z + r_w$ 
    ...
    comment: Calculate the reciprocal square root of the sum of squares  $\frac{1.0}{\sqrt{r_x}}$ 
     $r_x := \frac{1.0}{\sqrt{r_x}}$ 
    ...
    comment: Multiply by original value
     $r_x := r_x \times v_x$ 
    ...
end

```

```

13 // That is 1.0f / sqrt(value) - or the length of the vector
14 result[i] = _mm_rsqrt_ps(result[i]);
15 // Multiply result by the original data
16 // As we have the reciprocal it is the same as dividing each
   component
17 // by the length
18 result[i] = _mm_mul_ps(data[i], result[i]);
19 }
20 // All vectors now normalised
21 }

```

Listing 4.4: Normalise Vector Using SIMD

We can check the results using the code in Listing 4.5 - it will print out some results for us to compare.

```

1 // Check the first 100 results
2 void check_results(__m128 *data, __m128 *result)
3 {
4     // Convert to floats
5     auto float_data = (float*)data;
6     auto float_res = (float*)result;
7     // Check first 100 values
8     for (unsigned int i = 0; i < 100; ++i)
9     {
10         // Calculate the length of the vector
11         float l = 0.0f;
12         // Square each component and add to l
13         for (unsigned int j = 0; j < 4; ++j)
14             l += powf(float_data[(i * 4) + j], 2.0f);
15         // Square the length
16         l = sqrtf(l);
17         // Now check that the individual results
18         for (unsigned int j = 0; j < 4; ++j)
19             cout << float_data[(i * 4) + j] / l << " : " << float_res[(i
               * 4) + j] << endl;

```

```

20 | }
21 | }

```

Listing 4.5: Checking Normalisation

Running this application will produce the output shown in Figure 4.2.

```

C:\Windows\system32\cmd.exe
0.328646 : 0.328613
0.574221 : 0.574163
0.729979 : 0.729905
0.171442 : 0.171425
0.274767 : 0.274804
0.138731 : 0.13875
0.568267 : 0.568343
0.763105 : 0.763206
0.779347 : 0.77928
0.54634 : 0.546293
0.131444 : 0.131432
0.277224 : 0.2772
0.789006 : 0.789129
0.253382 : 0.253422
0.512998 : 0.513078
0.223831 : 0.223866
0.0156367 : 0.0156372
0.533944 : 0.533961
0.465985 : 0.466
0.705348 : 0.705371
0.0626172 : 0.0626065
0.521768 : 0.521678
0.220827 : 0.220789
0.821628 : 0.821487
Press any key to continue . . .

```

Figure 4.2: Output from SIMD Normalisation Application

You will notice that the values do not quite match. This is because we used a reciprocal method to normalize the vectors. This involved a fast multiplication over a slow division - but does mean the value is less accurate. However, as you can see from the data our results are close enough.

4.4 Exercises

We have really only scratched the surface of what is possible with SIMD instructions. The following exercises should help you fill out your knowledge.

1. Try some of the other SIMD instructions and see how they behave.
2. Calculate some timing information for the normalization application
3. Try and parallelize the normalization application using multithreading or OpenMP. Compare the time taken against SIMD.
4. Monte Carlo π works on the principal of the length of a random vector. Can you build a version of Monte Carlo π using SIMD instructions? Can you also parallelize it further using OpenMP or multithreading?

Unit 5

Distributed Parallelism with MPI

For the next two practicals it might be useful to work with a partner so you can get work on the distributed work we are undertaking. There is quite a bit of setup to do in this practical, so take your time and ensure everything is done correctly.

5.1 Installing MPI

We are going to use Microsofts HPC SDK to support our MPI work. To do this you need the following three items installed:

- Microsoft HPC Pack 2012 Client Components
- Microsoft HPC Pack 2012 SDK
- Microsoft HPC Pack 2012 MS-MPI Redistributable Pack

You will need to install these on every machine you plan to use in your application. You will also need to add the relevant include and library folders to your project - these will be found in the **Program Files** folder. The library that you need to link against is called `msmpi.lib`.

5.2 First MPI Application

Our first application will just initialise MPI, display some local information, and then shutdown. It is shown in Listing 5.1.

```
1 #include <iostream>
2 #include <mpi.h>
3
4 using namespace std;
5
6 int main()
7 {
8     // Initialise MPI
9     auto result = MPI_Init(nullptr, nullptr);
10    // Check that we initialised correctly
11    if (result != MPI_SUCCESS)
12    {
13        // Display error and abort
14        cout << "ERROR - initialising MPI!" << endl;
15        MPI_Abort(MPI_COMM_WORLD, result);
16        return -1;
17    }
18 }
```

```
17 }
18
19 // Get MPI information
20 int num_procs, rank, length;
21 char host_name[MPI_MAX_PROCESSOR_NAME];
22 MPI_Comm_size(MPI_COMM_WORLD, &num_procs);
23 MPI_Comm_rank(MPI_COMM_WORLD, &rank);
24 MPI_Get_processor_name(host_name, &length);
25
26 // Display information
27 cout << "Number of processors = " << num_procs << endl;
28 cout << "My rank = " << rank << endl;
29 cout << "Running on = " << host_name << endl;
30
31 // Shutdown MPI
32 MPI_Finalize();
33
34 return 0;
35 }
```

Listing 5.1: Hello MPI

The methods of interest are `MPI_Init` (initialises MPI), `MPI_Comm_size` (gets the number of processes in the application), `MPI_Comm_rank` (gets the ID of this process) and `MPI_Finalize` (shuts down MPI).

At the moment you should just build this application - running an MPI application takes a bit more work.

5.3 Running an MPI Application

You will need to open a command prompt in the directory where your built application is. Once you have done this, you can run the following command to execute the application locally in parallel:

```
mpiexec /np 4 "exe_name.exe"
```

Making sure to use the name of your application. The `/np` denotes the number of processes to use (here I use 4 - the number of logical cores on my machine). Running this command will give you an output similar to that shown in Figure 5.1.

5.4 Using a Remote Host

Running an MPI application in this method is all well and good, but we are not really doing any distributed parallelism. What we want to do is use one or more remote machines to do our processing. First you will need to find the IP address of the machine you want to use as the remote node. We do this using the `ipconfig` command:

```
ipconfig
```

This will give you the output similar to that shown in Figure 5.2.

```

C:\Windows\system32\cmd.exe
mpiexec -hosts 1 server1 master : -n 8 worker
For a complete list of options, run mpiexec -help2
For a list of environment variables, run mpiexec -help3
C:\Users\Kevin\Source\Repos\cps\Release>mpiexec /np 4 "05a Hello MPI.exe"
Number of processors = 4
Number of processors = 4
Number of processors = 4
My rank = 0
My rank = 2
My rank = 3
Number of processors = 4
Running on = kevin-ultrabook
Running on = kevin-ultrabook
Running on = kevin-ultrabook
My rank = 1
Running on = kevin-ultrabook
C:\Users\Kevin\Source\Repos\cps\Release>

```

Figure 5.1: Output from Initial MPI Application

The value you want is the IPv4 Address 192.168.1.70 in Figure 5.2. Next you want to run the following command on the remote machine:

```
smpd -d
```

This is called a **Single-Program Multiple-Data** task. It will listen on the machine and wait for us to allocate a job. To do this, we run `mpiexec` with a few more commands:

```
mpiexec /np 4 /host <ip-address> <application>
```

We now tell MPI which host to run on (we can also define multiple hosts). You will also need to copy the application to the other machine. Running this version will give a similar output to before.

It is worth at this point to look at the different flags for the `mpiexec`. You can find these at [http://technet.microsoft.com/en-us/library/cc947675\(v=ws.10\).aspx](http://technet.microsoft.com/en-us/library/cc947675(v=ws.10).aspx).

5.5 Sending and Receiving

The next few short examples will look at the different methods for communication. First of all we will use the standard send and receive messages. See Listing 5.2.

```

1 const unsigned int MAX_STRING = 100;
2
3 int main()
4 {
5     int num_procs, my_rank;
6
7     // Initialise MPI
8     auto result = MPI_Init(nullptr, nullptr);

```

```

C:\Windows\system32\cmd.exe

Connection-specific DNS Suffix  . : home
Link-local IPv6 Address . . . . . : fe80::704b:ecbd:84a8:a775%4
IPv4 Address. . . . . : 192.168.1.70
Subnet Mask . . . . . : 255.255.255.0
Default Gateway . . . . . : 192.168.1.254

Ethernet adapter Ethernet:

    Media State . . . . . : Media disconnected
    Connection-specific DNS Suffix  . :

Tunnel adapter Teredo Tunneling Pseudo-Interface:

    Connection-specific DNS Suffix  . :
    IPv6 Address. . . . . : 2001:0:9d38:6ab8:3887:2b4f:a958:b77
    Link-local IPv6 Address . . . . . : fe80::3887:2b4f:a958:b77%8
    Default Gateway . . . . . : ::

Tunnel adapter isatap.home:

    Media State . . . . . : Media disconnected
    Connection-specific DNS Suffix  . : home

C:\Users\Kevin\Source\Repos\cps\Release>

```

Figure 5.2: Output from ipconfig

```

9  if (result != MPI_SUCCESS)
10 {
11     cout << "ERROR - initialising MPI" << endl;
12     MPI_Abort(MPI_COMM_WORLD, result);
13     return -1;
14 }
15
16 // Get MPI Information
17 MPI_Comm_size(MPI_COMM_WORLD, &num_procs);
18 MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
19
20 // Check if we are the main process
21 if (my_rank != 0)
22 {
23     // Not main process - send message
24     // Generate message
25     stringstream buffer;
26     buffer << "Greeting from process " << my_rank << " of " <<
        num_procs << "!";
27     // Get the character array from the string
28     auto data = buffer.str().c_str();
29     // Send to the main node
30     MPI_Send((void*)data, buffer.str().length() + 1, MPI_CHAR, 0,
        0, MPI_COMM_WORLD);
31 }
32 else
33 {
34     // Main process - print message
35     cout << "Greetings from process " << my_rank << " of " <<
        num_procs << "!" << endl;
36     // Read in data from each worker process
37     char message[MAX_STRING];
38     for (int i = 1; i < num_procs; ++i)
39     {
40         // Receive message into buffer
41         MPI_Recv(message, MAX_STRING, MPI_CHAR, i, 0, MPI_COMM_WORLD,

```

```
        MPI_STATUS_IGNORE);  
42     // Display message  
43     cout << message << endl;  
44 }  
45 }  
46  
47 // Shutdown MPI  
48 MPI_Finalize();  
49  
50 return 0;  
51 }
```

Listing 5.2: Using MPI_Send and MPI_Recv

Here we are using the process rank to determine which process does what. The process with rank 0 we consider the main process, and its job is to receive messages (lines 32 to 45). Each other process will just send a message to the main process (lines 21 to 31).

Here we are using two new commands:

`MPI_Send` requires the data to be sent, the size of data (we make sure we send an extra byte for a string the null terminator), the type of data, the destination (0 the main process), a tag (we will not be using tags), and the communicator.

`MPI_Recv` requires a buffer to store the message, the maximum size of the buffer, the process to receive from, the tag, the communicator to use, and status conditions.

Running this application will give you an output similar to that shown in Figure 5.3.

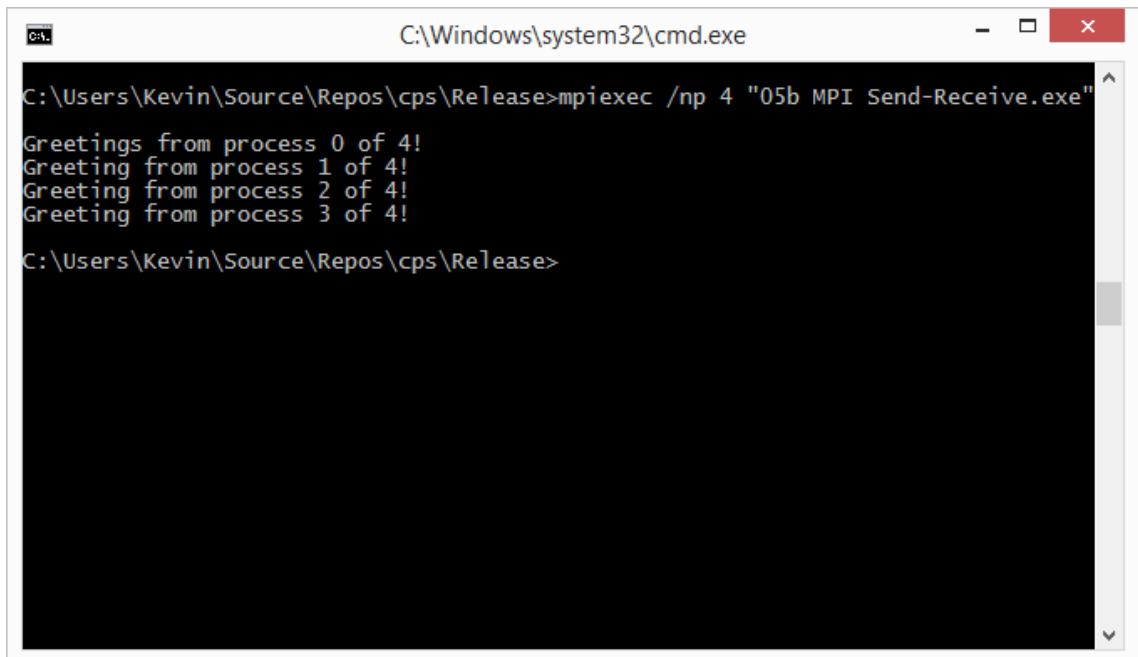


Figure 5.3: Output from MPI Send-Receive Application

There are a number of different data types MPI can use beyond `MPI_CHAR` - the Introduction to Parallel Programming book will explain these further.

5.6 Map-Reduce

Another approach to communication we can use is map-reduce. For this you will have to use a Monte-Carlo π application. The code snippet you require to perform the map-reduce approach is given in Listing 5.3.

```

1 double local_sum, global_sum;
2
3 // Calculate local sum - use previously defined function
4 local_sum = monte_carlo_pi(static_cast<unsigned int>(pow(2, 24)));
5 // Print out local sum
6 cout.precision(numeric_limits<double>::digits10);
7 cout << my_rank << ":" << local_sum << endl;
8 // Reduce
9 MPI_Reduce(&local_sum, &global_sum, 1, MPI_DOUBLE, MPI_SUM, 0,
10           MPI_COMM_WORLD);
11
12 // If main process display global reduced sum
13 if (my_rank == 0)
14 {
15     global_sum /= 4.0;
16     cout << "Pi=" << global_sum << endl;
17 }

```

Listing 5.3: MPI Map-Reduce Monte Carlo π

`MPI_Reduce` takes the following values:

- The value to send
- The value to reduce into
- The number of elements in the send buffer
- The type of the send buffer
- The reduction operation - we are using `MPI_SUM` to sum
- The rank of the process that collects the reduction operation - we use rank 0 as the main process
- The communicator used

Notice that we only use our main application to calculate π (lines 70 to 74). Running this application will produce an output similar to that shown in Figure 5.4.

5.7 Scatter-Gather

Scatter-gather involves us taking an array of data and distributing it evenly amongst the processes. Gathering involves us gathering the results back again at the end.

For scatter-gather we are going to implement our vector normalization application. You will need the two helper functions in Listing 5.4 to generate and normalize data.

```

1 // Randomly generate vector values
2 void generate_data(vector<float> &data)
3 {
4     // Create random engine

```



```

C:\Windows\system32\cmd.exe
C:\Users\Kevin\Source\Repos\cps\Release>mpiexec /np 4 "05c MPI Monte-Carlo Pi.exe"
2:3.14220428466797
3:3.14110040664673
0:3.14220428466797
1:3.14110040664673
Pi=3.14165234565735
C:\Users\Kevin\Source\Repos\cps\Release>

```

Figure 5.4: Output from Monte-Carlo π MPI Map-Reduce

```

5 | auto millis = duration_cast<milliseconds>(system_clock::now().
   |     time_since_epoch());
6 | default_random_engine e(static_cast<unsigned int>(millis.count())
   | );
7 | // Fill data
8 | for (unsigned int i = 0; i < data.size(); ++i)
9 |     data[i] = e();
10| }
11|
12| // Normalises 4D vectors
13| void normalise_vector(vector<float> &data)
14| {
15|     // Iterate through each 4-dimensional vector
16|     for (unsigned int i = 0; i < (data.size() / 4); ++i)
17|     {
18|         // Sum the squares of the 4 components
19|         float sum = 0.0f;
20|         for (unsigned int j = 0; j < 4; ++j)
21|             sum += powf(data[(i * 4) + j], 2.0f);
22|         // Get the square root of the result
23|         sum = sqrtf(sum);
24|         // Divide each component by sum
25|         for (unsigned int j = 0; j < 4; ++j)
26|             data[(i * 4) + j] /= sum;
27|     }
28| }

```

Listing 5.4: Helper Methods for MPI Scatter-Gather Vector Normalisation

Our main application just needs to call scatter, normalize, gather. Listing 5.5 is the snippet you require.

```

1 | // Vector containing values to normalise
2 | vector<float> data;
3 | // Local storage. Allocate enough space
4 | vector<float> my_data(SIZE / num_procs);
5 |

```

```

6 // Check if main process
7 if (my_rank == 0)
8 {
9     // Generate data
10    data.resize(SIZE);
11    generate_data(data);
12 }
13
14 // Scatter the data
15 MPI_Scatter(&data[0], SIZE / num_procs, MPI_FLOAT, // Source
16            &my_data[0], SIZE / num_procs, MPI_FLOAT, // Destination
17            0, MPI_COMM_WORLD);
18 // Normalise local data
19 normalise_vector(my_data);
20 // Gather the results
21 MPI_Gather(&my_data[0], SIZE / num_procs, MPI_FLOAT, // Source
22            &data[0], SIZE / num_procs, MPI_FLOAT, // Dest
23            0, MPI_COMM_WORLD);
24
25 // Check if main process
26 if (my_rank == 0)
27 {
28     // Display results - first 10
29     for (unsigned int i = 0; i < 10; ++i)
30     {
31         cout << "<";
32         for (unsigned int j = 0; j < 3; ++j)
33             cout << data[(i * 4) + j] << ", ";
34         cout << data[(i * 4) + 3] << ">" << endl;
35     }
36 }

```

Listing 5.5: Using MPI Scatter-Gather to Normalise a Vector

The `MPI_Scatter` command has the following parameters:

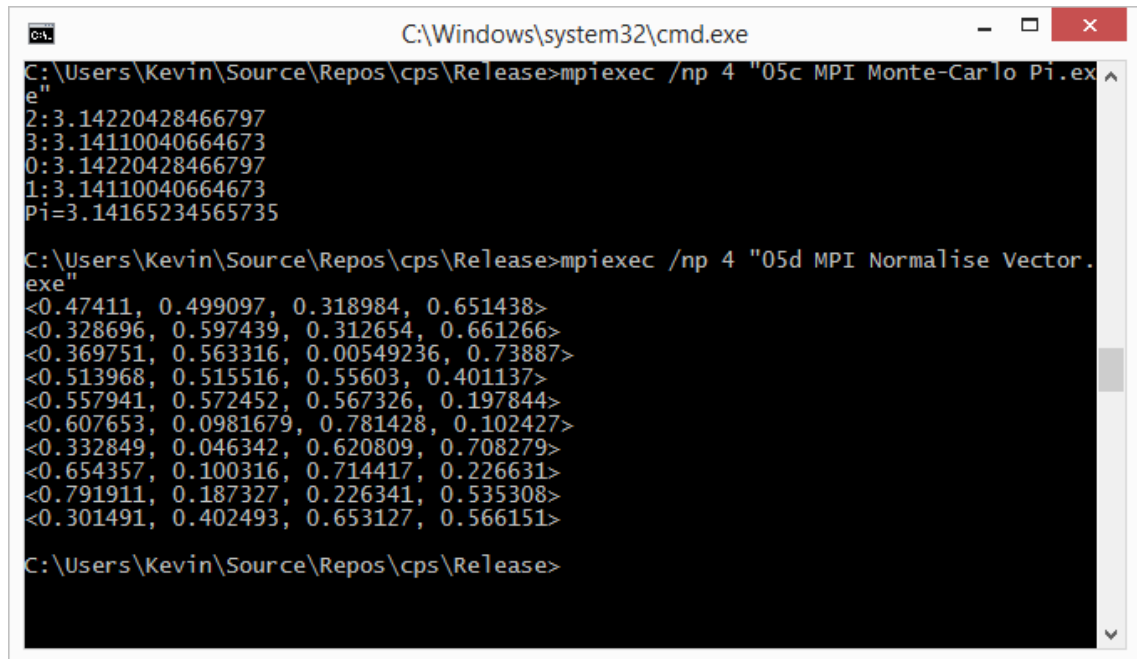
- The data to scatter - only relevant on the root process
- The count of data to send to each process
- The type of data sent
- The memory to receive the data into on each process
- The count of data to receive at each process
- The type of data received
- The root process (sender)
- The communicator used

`MPI_Gather` is essentially this in reverse:

- The local data to send
- The count of data to send from each process
- The type of data sent
- The memory to gather results into

- The count of data to receive from each process
- The type of data received
- The root process
- The communicator used

Running this application will provide the output shown in Figure 5.5.



```

C:\Windows\system32\cmd.exe
C:\Users\Kevin\Source\Repos\cps\Release>mpiexec /np 4 "05c MPI Monte-Carlo Pi.exe"
2:3.14220428466797
3:3.14110040664673
0:3.14220428466797
1:3.14110040664673
Pi=3.14165234565735

C:\Users\Kevin\Source\Repos\cps\Release>mpiexec /np 4 "05d MPI Normalise Vector.exe"
<0.47411, 0.499097, 0.318984, 0.651438>
<0.328696, 0.597439, 0.312654, 0.661266>
<0.369751, 0.563316, 0.00549236, 0.73887>
<0.513968, 0.515516, 0.55603, 0.401137>
<0.557941, 0.572452, 0.567326, 0.197844>
<0.607653, 0.0981679, 0.781428, 0.102427>
<0.332849, 0.046342, 0.620809, 0.708279>
<0.654357, 0.100316, 0.714417, 0.226631>
<0.791911, 0.187327, 0.226341, 0.535308>
<0.301491, 0.402493, 0.653127, 0.566151>

C:\Users\Kevin\Source\Repos\cps\Release>

```

Figure 5.5: Output from MPI Scatter-Gather Application

You can check to see if these vectors are normalised.

5.8 Broadcast

The final communication type we will look at is broadcast. Broadcasting just allows us to send a message from one source to all processes on the communicator. Listing 5.6 demonstrates.

```

1 // Check if main process
2 if (my_rank == 0)
3 {
4     // Broadcast message to workers
5     string str = "Hello World!";
6     MPI_Bcast((void*)&str.c_str()[0], str.length() + 1, MPI_CHAR, 0,
7               MPI_COMM_WORLD);
8 }
9 else
10 {
11     // Receive message from main process
12     char data[100];
13     MPI_Bcast(data, 100, MPI_CHAR, 0, MPI_COMM_WORLD);
14     cout << my_rank << ":" << data << endl;
15 }

```

Listing 5.6: MPI Broadcast Example

`MPI_Bcast` takes the following parameters:

- Data to broadcast (sender sends, receiver reads into this data)
- Count of data to send / receive)
- Data type sent / received
- Root node
- Communicator

5.9 Exercises

1. As always you should be taking timings of your applications
2. The Mandelbrot is quite an interesting application to distribute. In particular you will find that our implementation allows some parts to be processed quickly, and other parts slowly. You should try and divide the work so that you can optimise performance.
3. Try and get an application that works across a number of hosts. Try four machines in the games lab (16 processes in total). Again gather timings. Mandelbrot is another good application here.

Unit 6

More MPI

Our second tutorial on MPI will focus on particular examples of using MPI from our previous work. We will just focus on the examples rather than go into any new MPI as such.

6.1 Mandelbrot

For Mandelbrot our task is quite easy - the implementation we had was designed to run using rank based execution. We just need to modify it to use MPI ranks. All you require is Listing 6.1 to execute the Mandelbrot process.

```
1 // Broadcast dimension around the workers
2 unsigned int dim = 0;
3 if (my_rank == 0)
4 {
5     // Broadcast dimension to all the workers - could read this in
6     // from user
7     dim = 8192;
8     MPI_Bcast(&dim, 1, MPI_UNSIGNED, 0, MPI_COMM_WORLD);
9 }
10 else
11     // Get dimension
12     MPI_Bcast(&dim, 1, MPI_UNSIGNED, 0, MPI_COMM_WORLD);
13 // Perform Mandelbrot
14 auto strip_height = dim / num_procs;
15 auto res = mandelbrot(dim, my_rank * strip_height, (my_rank + 1) *
16     strip_height);
17 // Gather results back
18 vector<float> data;
19 // Check if main process - if so resize data to gather results
20 if (my_rank == 0)
21     data.resize(res.size() * num_procs);
22 // Gather results
23 MPI_Gather(&res[0], res.size(), MPI_FLOAT, &data[0], res.size(),
24     MPI_FLOAT, 0, MPI_COMM_WORLD);
25 // Save image..
```

Listing 6.1: MPI Mandelbrot

We use gather here (line 23) to gather the final results. This is the point where the results are gather back to the host machine.

6.2 Parallel Sort

For parallel sort we have to do quite a lot more work to get things to work.

The parallel sort we used was something called an odd-even sort. This works in a number of phases that requires sharing between processes. The general algorithm is as follows:

1. Sort local data
2. For number of phases
 - (a) Exchange data with phase partner
 - (b) Merge
3. Gather results

Because of this we have a number of new operations we have to define. First, let us define two merge methods - one to merge at the top of the list and one at the bottom. These are given in Listing 6.2.

```

1 // Merges the largest n values in local_data and temp_b into temp_c
2 // temp_c is then copied back into local_data
3 void merge_high(vector<unsigned int> &local_data, vector<unsigned
4   int> &temp_b, vector<unsigned int> &temp_c)
5 {
6   int ai, bi, ci;
7   // Get starting size
8   ai = bi = ci = local_data.size() - 1;
9   // Loop through each value and store relevant largest value in
10  temp_c
11  for (; ci >= 0; --ci)
12  {
13    // Find largest from local data and temp_b
14    if (local_data[ai] >= temp_b[bi])
15    {
16      temp_c[ci] = local_data[ai];
17      --ai;
18    }
19    else
20    {
21      temp_c[ci] = temp_b[bi];
22      --bi;
23    }
24  }
25  // Copy temp_c into local_data
26  copy(temp_c.begin(), temp_c.end(), local_data.begin());
27 }
28 // Merges the smallest n values in local_data and temp_b into
29 temp_c
30 // temp_c is then copied back into local_data
31 void merge_low(vector<unsigned int> &local_data, vector<unsigned
32   int> &temp_b, vector<unsigned int> &temp_c)
33 {
34   int ai, bi, ci;
35   // Start at 0
36   ai = bi = ci = 0;
37   // Loop through each value and store relevant smallest value in
38   temp_c

```

```

35 | for (; ci < local_data.size(); ++ci)
36 | {
37 |     // Find smallest from local data and temp_b
38 |     if (local_data[ai] <= temp_b[bi])
39 |     {
40 |         temp_c[ci] = local_data[ai];
41 |         ++ai;
42 |     }
43 |     else
44 |     {
45 |         temp_c[ci] = temp_b[bi];
46 |         ++bi;
47 |     }
48 | }
49 | // Copy temp_c into local_data
50 | copy(temp_c.begin(), temp_c.end(), local_data.begin());
51 | }

```

Listing 6.2: Merge Algorithms for MPI Parallel Sort

We call these merges during an odd-even iteration where we exchange data between partners. This is illustrated in Listing 6.3.

```

1 | void odd_even_iter(vector<unsigned int> &local_data, vector<
  |   unsigned int> &temp_b, vector<unsigned int> &temp_c, unsigned
  |   int phase, int even_partner, int odd_partner, unsigned int
  |   my_rank, unsigned int num_procs)
2 | {
3 |     // Operate based on phase
4 |     if (phase % 2 == 0)
5 |     {
6 |         // Check if even partner is valid
7 |         if (even_partner >= 0)
8 |         {
9 |             // Exchange data with even partner
10 |            MPI_Sendrecv(&local_data[0], local_data.size(), MPI_UNSIGNED,
  |                        even_partner, 0, &temp_b[0], temp_b.size(), MPI_UNSIGNED,
  |                        even_partner, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
11 |            // Merge results accordingly
12 |            if (my_rank % 2 == 0)
13 |                merge_low(local_data, temp_b, temp_c);
14 |            else
15 |                merge_high(local_data, temp_b, temp_c);
16 |        }
17 |    }
18 |    else
19 |    {
20 |        // Check if odd partner is valid
21 |        if (odd_partner >= 0)
22 |        {
23 |            // Exchange data with odd partner
24 |            MPI_Sendrecv(&local_data[0], local_data.size(), MPI_UNSIGNED,
  |                        odd_partner, 0, &temp_b[0], temp_b.size(), MPI_UNSIGNED,
  |                        odd_partner, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
25 |            // Merge results accordingly
26 |            if (my_rank % 2 == 0)
27 |                merge_high(local_data, temp_b, temp_c);
28 |            else
29 |                merge_low(local_data, temp_b, temp_c);
30 |        }
31 |    }

```

32 }

Listing 6.3: Iteration Merging in MPI Parallel Sort

Depending on the phase and whether we have a partner to the left or right, we exchange data accordingly. We then merge the data with our results, resulting in us having either the sorted upper portion of the two processes data or the sorted lower portion. Eventually each process will end up with its relevant sorted portion which we can send back to the main process.

The sort method itself is below. It merely sorts the local data section before performing the necessary number of phases (which is equal to the number of processes involved). The sort operation is defined in Listing 6.4.

```

1 // Odd-even sort
2 void odd_even_sort(vector<unsigned int> &local_data, unsigned int
   my_rank, unsigned int num_procs)
3 {
4     // Temporary storage
5     vector<unsigned int> temp_b(local_data);
6     vector<unsigned int> temp_c(local_data);
7     // Partners. Even phase look left. Odd phase looks right
8     int even_partner, odd_partner;
9
10    // Find partners
11    if (my_rank % 2 == 0)
12    {
13        even_partner = static_cast<int>(my_rank) + 1;
14        odd_partner = static_cast<int>(my_rank) - 1;
15        // Check that even_partner is valid
16        if (even_partner == num_procs)
17            even_partner = MPI_PROC_NULL;
18    }
19    else
20    {
21        even_partner = static_cast<int>(my_rank) - 1;
22        odd_partner = static_cast<int>(my_rank) + 1;
23        // Check that odd_partner is valid
24        if (odd_partner == num_procs)
25            odd_partner = MPI_PROC_NULL;
26    }
27
28    // Sort this processes share of the data
29    // std::sort is in the algorithm header
30    sort(local_data.begin(), local_data.end());
31
32    // Phased odd-even transposition sort
33    for (unsigned int phase = 0; phase < num_procs; ++phase)
34        odd_even_iter(local_data, temp_b, temp_c, phase, even_partner,
   odd_partner, my_rank, num_procs);
35 }

```

Listing 6.4: Odd-Even Parallel Sort using MPI

The sort and phases are at the bottom of the method. All we need is to call the sort method after scattering out the data. We then gather the data at the end. Listing 6.5 is the main application (apart from the MPI initialisation and finalisation).

```

1 // Data to sort
2 vector<unsigned int> data;

```



```

3 // If main process generate the data
4 if (my_rank == 0)
5     data = generate_values(SIZE);
6
7 // Allocate enough space for local working data
8 vector<unsigned int> local_data(SIZE / num_procs);
9
10 // Scatter the data
11 cout << my_rank << ":Scattering" << endl;
12 MPI_Scatter(&data[0], SIZE / num_procs, MPI_UNSIGNED, &local_data
13           [0], SIZE / num_procs, MPI_UNSIGNED, 0, MPI_COMM_WORLD);
14
15 // Sort the data
16 cout << my_rank << ":Sorting" << endl;
17 odd_even_sort(local_data, my_rank, num_procs);
18
19 // Gather the results
20 cout << my_rank << ":Gathering" << endl;
21 MPI_Gather(&local_data[0], SIZE / num_procs, MPI_UNSIGNED, &data
22           [0], SIZE / num_procs, MPI_UNSIGNED, 0, MPI_COMM_WORLD);
23
24 // If main process output first 1000 values
25 if (my_rank == 0)
26     for (unsigned int i = 0; i < 1000; ++i)
27         cout << data[i] << endl;

```

Listing 6.5: Main MPI Application for Odd-Even Parallel Sort

Running this application will give you the output shown in Figure 6.1.

```

C:\Windows\system32\cmd.exe
257761
257948
258195
258220
258304
258327
258347
258589
259057
259122
259183
259606
259910
259970
260372
261031
261358
261482
261860
261881
261995
262030
262549
C:\Users\Kevin\Source\Repos\cps\Release>

```

Figure 6.1: Output from MPI Parallel Sort Algorithm

6.3 Trapezoidal Rule

For the trapezoidal rule we will use a barrier to synchronise our work. Listing 6.6 is all the code you need.

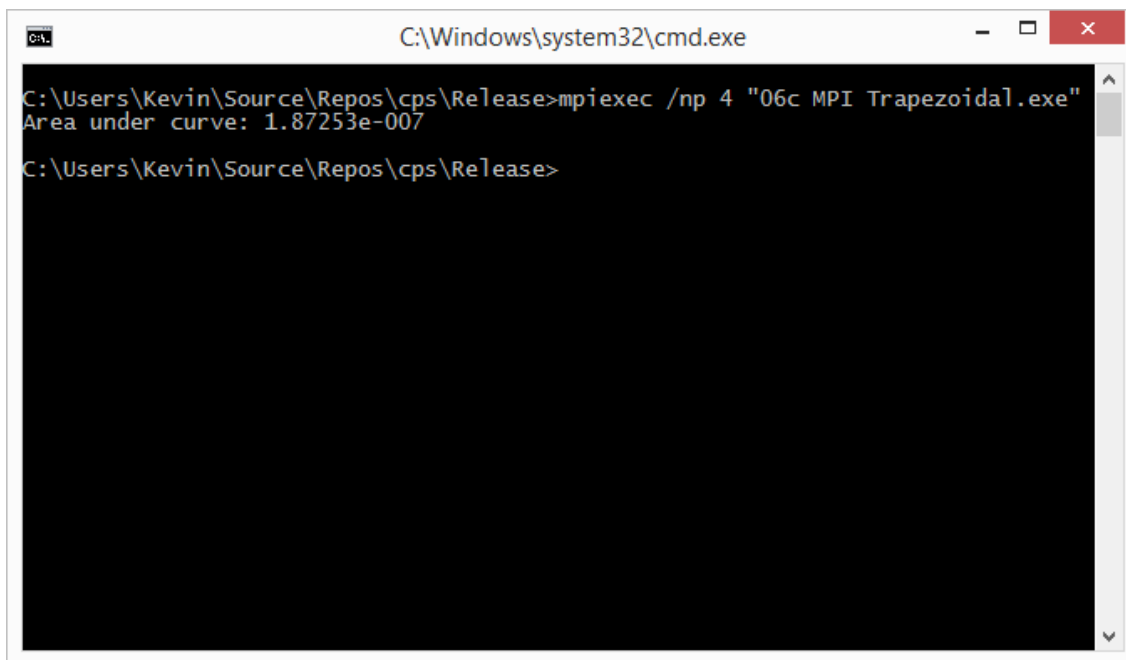
```

1 // Sync
2 MPI_Barrier(MPI_COMM_WORLD);
3
4 // Function to use
5 auto f = [](double x) { return cos(x); };
6
7 // Perform calculation
8 unsigned int iterations = static_cast<unsigned int>(pow(2, 24));
9 auto local_result = trap(f, 0.0, PI, iterations, my_rank, num_procs
10 );
11 // Reduce result
12 double global_result = 0.0;
13 MPI_Reduce(&local_result, &global_result, 1, MPI_DOUBLE, MPI_SUM,
14           0, MPI_COMM_WORLD);
15 // If main process, display result
16 if (my_rank == 0)
17     cout << "Area under curve: " << global_result << endl;

```

Listing 6.6: Trapezoidal Rule using MPI

Running this code will give you an output similar to Figure 6.2.



```

C:\Windows\system32\cmd.exe
C:\Users\Kevin\Source\Repos\cps\Release>mpiexec /np 4 "06c MPI Trapezoidal.exe"
Area under curve: 1.87253e-007
C:\Users\Kevin\Source\Repos\cps\Release>

```

Figure 6.2: Output from MPI Trapezoidal Rule Application

6.4 Performance Evaluation of MPI

We will now look at how we can go about measuring latency and bandwidth using MPI. These values can be useful if you are undertaking any serious distribution of tasks and data communication. However, you will likely find that the stated network speed is what we hit.

6.4.1 Measuring Latency

For latency all you need is the application in Listing 6.7.

```

1 // Perform 100 timings
2 for (unsigned int i = 0; i < 100; ++i)
3 {
4     // Sync with other process
5     MPI_Barrier(MPI_COMM_WORLD);
6
7     // Get start time
8     auto start = system_clock::now();
9     // Perform 100000 ping-pongs
10    if (my_rank == 0)
11    {
12        for (unsigned int j = 0; j < 100000; ++j)
13        {
14            // Ping
15            MPI_Send(&send, 1, MPI_CHAR, 1, 0, MPI_COMM_WORLD);
16            // Pong
17            MPI_Recv(&receive, 1, MPI_CHAR, 1, 0, MPI_COMM_WORLD,
18                    MPI_STATUS_IGNORE);
19        }
20    }
21    else
22    {
23        for (unsigned int j = 0; j < 100000; ++j)
24        {
25            // Pong
26            MPI_Recv(&receive, 1, MPI_CHAR, 0, 0, MPI_COMM_WORLD,
27                    MPI_STATUS_IGNORE);
28            // Ping
29            MPI_Send(&send, 1, MPI_CHAR, 0, 0, MPI_COMM_WORLD);
30        }
31    }
32    // Get time
33    auto end = system_clock::now();
34    auto total = duration_cast<nanoseconds>(end - start).count();
35    // Divide total by number of iterations to get time per iteration
36    auto res = static_cast<double>(total) / 100000.0;
37    // Divide by two to get latency
38    res /= 2.0;
39    // Output result
40    if (my_rank == 0)
41        data << res << ", ";
42 }
43
44 // Close file if main process
45 if (my_rank == 0)
46 {
47     data << endl;
48     data.close();
49 }

```

Listing 6.7: MPI Latency Measurement Application

You will need to run this application across two machines to get a latency time.

6.4.2 Measuring Bandwidth

For bandwidth you just need to change the latency application so that it has bigger data sizes. Use powers of two as normal, and range from approximately 1K to 1MB. You will have to convert from the time taken to send the message to the actual MBit/s.

You should also measure broadcast to see the performance there as well. **REMEMBER** - create the charts and look at the performance. You should be able to predict the performance of an application purely by sequential computation time plus communication time.

6.5 Exercises

1. Time the Mandelbrot. Take into account the data transmission time. Try and optimise the application as much as possible and split across a number of nodes. Also try different data sizes to analyse performance.
2. Now do the same with parallel sort. There is more data exchanges happening here so you will need to think about a number of I/O stages taking place.
3. For the trapezoidal rule you will probably want to seriously increase the number of iterations to get any reasonable idea of timings.
4. Test the latency and the bandwidth of the Games Lab. Does it meet your expectations? Take these values into account and start to estimate the probable performance of Mandelbrot and parallel sort.

6.6 Reading

Read Chapter 3 of Introduction to Parallel Programming for more information on MPI.

Unit 7

GPU Programming with OpenCL

We are now going to move onto programming the GPU to perform data parallel processing. Using the GPU in this manner is a relatively new concept. However, the principals are essentially the same as shader programming.

Before getting started you will need to make sure you have the relevant SDK installed on the machine you are using. This will depend on the hardware you are using. Intel, Nvidia and AMD each provide an SDK (the Intel and AMD ones also support the CPU as an OpenCL device). You will have to work out the setup of your OpenCL projects in Visual Studio. After that, you will be able to run OpenCL applications.

The header we will be using is `CL/cl.h`. There is a C++ header (`cl.hpp`) in some installations but not all - so we will work at a C level rather than C++. The library is `OpenCL.lib`.

7.1 Getting Started with OpenCL

Our first application is purely about setting up OpenCL. To do this we will use the code in Listing 7.1.

```
1 // Initialise OpenCL
2 void initialise_opencl(vector<cl_platform_id> &platforms, vector<
    cl_device_id> &devices, cl_context &context, cl_command_queue &
    cmd_queue)
3 {
4     // Status of OpenCL calls
5     cl_int status;
6
7     // Get the number of platforms
8     cl_uint num_platforms;
9     status = clGetPlatformIDs(0, nullptr, &num_platforms);
10    // Resize vector to store platforms
11    platforms.resize(num_platforms);
12    // Fill in platform vector
13    status = clGetPlatformIDs(num_platforms, &platforms[0], nullptr);
14
15    // Assume platform 0 is the one we want to use
16    // Get devices for platform 0
17    cl_uint num_devices;
18    status = clGetDeviceIDs(platforms[0], CL_DEVICE_TYPE_GPU, 0,
        nullptr, &num_devices);
19    // Resize vector to store devices
20    devices.resize(num_devices);
21    // Fill in devices vector
```

```

22     status = clGetDeviceIDs(platforms[0], CL_DEVICE_TYPE_GPU,
23                             num_devices, &devices[0], nullptr);
24     // Create a context
25     context = clCreateContext(nullptr, num_devices, &devices[0],
26                             nullptr, nullptr, &status);
27     // Create a command queue
28     cmd_queue = clCreateCommandQueue(context, devices[0], 0, &status)
29     ;

```

Listing 7.1: Initialising OpenCL

The first thing we do is get the number of platforms (line 9) and use this to resize a vector to store the platform information (lines 11 to 13). A platform in OpenCL is a different OpenCL runtime - for example your machine could have both Intel and Nvidia platforms.

Next we get the devices supported by Platform 0 - this assumes that Platform 0 is your GPU platform so you might have to modify this code. As we also want to only work with the GPU we use the device type `CL_DEVICE_TYPE_GPU`. You can use the following devices types:

`CL_DEVICE_TYPE_CPU` a CPU device

`CL_DEVICE_TYPE_GPU` a GPU device

`CL_DEVICE_TYPE_ACCELERATOR` a custom OpenCL device

`CL_DEVICE_TYPE_DEFAULT` the default OpenCL device

`CL_DEVICE_TYPE_ALL` all devices on the platform

The final two steps are the creation of a `cl_context` (allows creation of command queues, kernels, and memory) and a `cl_command_queue` (allows sending of commands to the OpenCL device).

To use this code we use Listing ??.

```

1  int main()
2  {
3      // Status of OpenCL calls
4      cl_int status;
5
6      // Initialise OpenCL
7      vector<cl_platform_id> platforms;
8      vector<cl_device_id> devices;
9      cl_context context;
10     cl_command_queue cmd_queue;
11     initialise_opencl(platforms, devices, context, cmd_queue);
12
13     // Free OpenCL resources
14     clReleaseCommandQueue(cmd_queue);
15     clReleaseContext(context);
16
17     return 0;
18 }

```

Listing 7.2: Using OpenCL Initialise

All we are doing at the moment is calling the initialise method (line 11) and then cleaning up the resources (lines 14 and 15). Running this application will not do anything, but it will allow you to check your OpenCL setup seems to be working.

7.2 Getting OpenCL Info

Our next application will print out the information for our OpenCL devices. To do this, we just need to grab the info using some of the OpenCL functions - again it is much like doing so from OpenGL. Listing 7.3 will print out the information for all the devices grabbed from the platform.

```

1 // Helper method to print OpenCL device info
2 void print_opengl_info(vector<cl_device_id> &devices)
3 {
4     // Buffers for device name and vendor
5     char device_name[1024], vendor[1024];
6     // Declare other necessary variables
7     cl_uint num_cores;
8     cl_long memory;
9     cl_uint clock_freq;
10    cl_bool available;
11
12    // Iterate through each device in vector and display information
13    for (auto &d : devices)
14    {
15        // Get info for device
16        clGetDeviceInfo(d, CL_DEVICE_NAME, 1024, device_name, nullptr);
17        clGetDeviceInfo(d, CL_DEVICE_VENDOR, 1024, vendor, nullptr);
18        clGetDeviceInfo(d, CL_DEVICE_MAX_COMPUTE_UNITS, sizeof(cl_uint)
19            , &num_cores, nullptr);
20        clGetDeviceInfo(d, CL_DEVICE_GLOBAL_MEM_SIZE, sizeof(cl_long),
21            &memory, nullptr);
22        clGetDeviceInfo(d, CL_DEVICE_MAX_CLOCK_FREQUENCY, sizeof(
23            cl_uint), &clock_freq, nullptr);
24        clGetDeviceInfo(d, CL_DEVICE_AVAILABLE, sizeof(cl_bool), &
25            available, nullptr);
26
27        // Print info
28        cout << "Device: " << device_name << endl;
29        cout << "Vendor: " << vendor << endl;
30        cout << "Cores: " << num_cores << endl;
31        cout << "Memory: " << memory / (1024 * 1024) << "MB" << endl;
32        cout << "Clock freq: " << clock_freq << "MHz" << endl;
33        cout << "Available: " << available << endl;
34        cout << "*****" << endl << endl;
35    }
36 }

```

Listing 7.3: Printing OpenCL Information

This is a fairly straightforward method and you should know how to update your main application to use it. Running this will give you the information about your OpenCL devices. My office machine provides the output shown in Figure 7.1.

Modifying this to get all the devices for the platform might change your output. If you are using AMD or Intel hardware, you can also retrieve the CPU values. For example, my laptop (running Intel HD4000), provides the following CPU information:

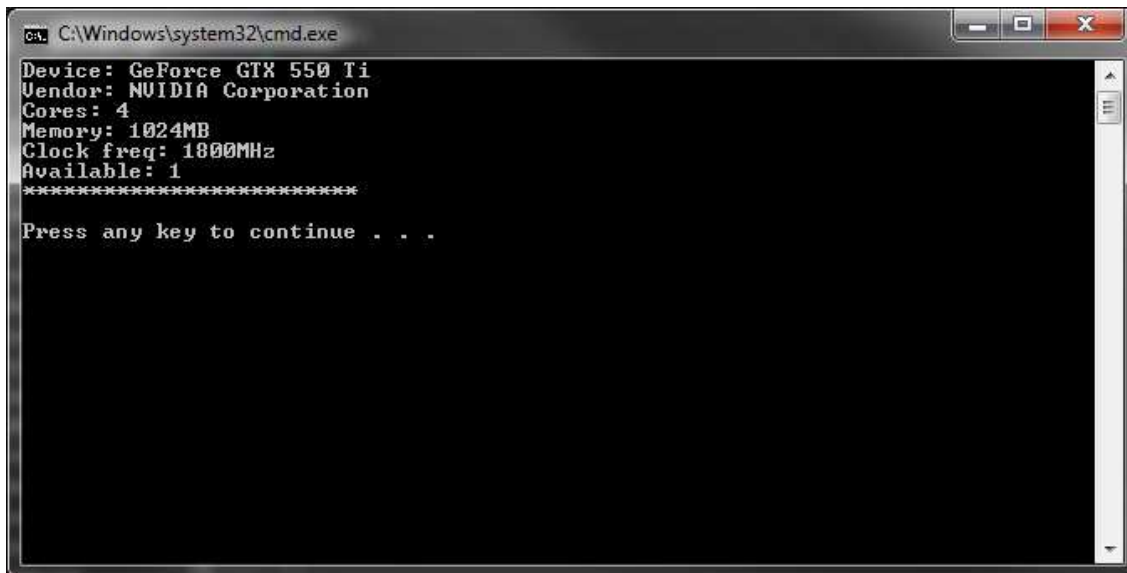


Figure 7.1: GPU Properties from OpenCL

GPU :

Device HD Graphics 4000

Cores 16

Memory 1752 MB

Clock freq 350 MHz

CPU :

Device i5 CPU

Cores 4

Memory 2047 MB

Clock 1900 MHz

So judging from the hardware specs above I have the following potential performance:

GPU desktop $4 \times 1800MHz = 7200MHz = 7.2GHz$

CPU laptop $4 \times 1900MHz = 7600MHz = 7.6GHz$

GPU laptop $16 \times 350MHz = 5600MHz = 5.6GHz$

My CPU looks more powerful but these are logical cores not physical. The CPU really only has 2 physical cores, giving 3.8 GHz. My desktop GPU also boasts 192 CUDA cores, which is theoretically 345.6 GHz. However, the graphics clock according to the specification is only 900 MHz, not 1800 MHz, so we have 172.8 Ghz.

On a standard desktop the difference between the potential processing power of the CPU and the GPU is far more dramatic. The GPU is usually many more cores at 800Mhz or more.

7.3 Loading an OpenCL Kernel

We are now going to look at how we have a basic method of setting up OpenCL and displaying the information we require, let us move onto performing some processing. This involves us loading what are called kernels. You can think of this a little bit like loading a shader, but we are doing less specialised programming and more general purpose (hence the name *General Purpose GPU programming*).

The kernel we are using is given in Listing 7.4. You should save this in a file called `kernel.cl`.

```

1 __kernel void vecadd(__global int *A, __global int *B, __global int
   *C)
2 {
3     // Get the work item's unique ID
4     int idx = get_global_id(0);
5     // Add corresponding locations of A and B and store in C
6     C[idx] = A[idx] + B[idx];
7 }

```

Listing 7.4: First OpenCL Kernel - Vector Addition

We will look at this code in a bit of detail. First, if our function is a kernel we use the keyword `__kernel` at the start of the declaration. Kernels do not return values, so our return value is `void`.

The parameters for our kernel all are declared as `__global`. This means that they are accessible to all the cores when the kernel executes it is global memory. We declare these as pointers as they will be blocks of memory that we will be accessing and writing to.

Our kernel is adding two vectors - or two arrays of a particular size (we will add two 2048 element vectors together). The `get_global_id` function allows us to get the index of the current executing thread. A thread can have various dimensions for the index - so we can get the `id` for 0, 1, 2, etc. We can also get the local `id` for work groups. As our kernel adds two 1D vectors, we only need to use the 0 dimension.

The final line of the kernel just stores the value. It is a standard line of code.

To load a kernel, we use Listing 7.5.

```

1 // Loads an OpenCL program
2 cl_program load_program(const string &filename, cl_context &context
   , cl_device_id &device, cl_int num_devices)
3 {
4     // Status of OpenCL calls
5     cl_int status;
6
7     // Create and compile program
8     // Read in kernel file
9     ifstream input(filename, ifstream::in);
10    stringstream buffer;
11    buffer << input.rdbuf();
12    // Get the character array of the file contents
13    auto file_contents = buffer.str();
14    auto char_contents = file_contents.c_str();
15
16    // Create program object
17    auto program = clCreateProgramWithSource(context, 1, &
        char_contents, nullptr, &status);
18    // Compile / build program
19    status = clBuildProgram(program, num_devices, &device, nullptr,
        nullptr, nullptr);

```

```

20
21 // Check if compiled
22 if (status != CL_SUCCESS)
23 {
24     // Error building - get log
25     size_t length;
26     clGetProgramBuildInfo(program, device, CL_PROGRAM_BUILD_LOG, 0,
27                             nullptr, &length);
28     char *log = new char[length];
29     clGetProgramBuildInfo(program, device, CL_PROGRAM_BUILD_LOG,
30                             length, log, &length);
31     // Print log
32     cout << log << endl;
33     delete[] log;
34 }
35
36 // Return program object
37 return program;
38 }

```

Listing 7.5: Loading an OpenCL Kernel

This method works much how we would load a shader. We read in the file contents (lines 9 to 14), and then create a `cl_program` object (line 17) and build it (line 19). If the build fails, we print out the build log (lines 22 to 32).

Now that we are loading a program, we need to choose which kernel to use. We can do this by updating the main function. Our updated version is in Listing 7.6.

```

1 int main()
2 {
3     // Status of OpenCL calls
4     cl_int status;
5
6     // Initialise OpenCL
7     vector<cl_platform_id> platforms;
8     vector<cl_device_id> devices;
9     cl_context context;
10    cl_command_queue cmd_queue;
11    initialise_opencl(platforms, devices, context, cmd_queue);
12
13    // Print info
14    print_opencl_info(devices);
15
16    // Load program
17    auto program = load_program("kernel.cl", context, devices[0],
18                                devices.size());
19
20    // Create the kernel
21    auto kernel = clCreateKernel(program, "vecadd", &status);
22
23    // Free OpenCL resources
24    clReleaseCommandQueue(cmd_queue);
25    clReleaseContext(context);
26    clReleaseKernel(kernel);
27    clReleaseProgram(program);
28
29    return 0;
30 }

```

Listing 7.6: OpenCL Main with Kernel Loading

We load the program on line 17, passing in the name of our file. We then select the kernel we want to use on line 20. Notice as well that we are now releasing our kernel and program (lines 25 and 26). Running this application should still print out your OpenCL device properties - however it should not print out an error log.

7.4 Passing Data to OpenCL Kernels

We are now ready to send data to our OpenCL kernel. This involves us creating memory buffers on the GPU and copying the data to the GPU.

The first thing we need to do is create the “host” memory - that is the memory that sits in main memory accessible by the CPU. For our application we will create two arrays of 2048 elements. Listing 7.7 should be added to the main function.

```

1 // Number of elements and size of buffer on GPU
2 const unsigned int elements = 2048;
3 const unsigned int data_size = sizeof(int) * elements;
4
5 // Host data - stored in main memory
6 array<int, elements> A;
7 array<int, elements> B;
8 array<int, elements> C;
9
10 // Initialise input data
11 for (unsigned int i = 0; i < elements; ++i)
12     A[i] = B[i] = i;

```

Listing 7.7: Creating Host Memory for OpenCL Application

Next we need to create our buffers on the OpenCL device. We do this in Listing 7.8.

```

1 // Create device buffers - stored on GPU
2 cl_mem buffer_A; // Input array on the device
3 cl_mem buffer_B; // Input array on the device
4 cl_mem buffer_C; // Output array on the device
5 // Allocate buffer size
6 buffer_A = clCreateBuffer(context, CL_MEM_READ_ONLY, data_size,
7     nullptr, &status);
8 buffer_B = clCreateBuffer(context, CL_MEM_READ_ONLY, data_size,
9     nullptr, &status);
10 buffer_C = clCreateBuffer(context, CL_MEM_WRITE_ONLY, data_size,
11     nullptr, &status);

```

Listing 7.8: Creating Device Memory for OpenCL Application

Notice that we have to tell OpenCL the type of buffer we are creating (read, write, etc.) and the size of the buffer. This is just creating a buffer on the GPU in our instance it is allocating memory. All we need to do is copy our host data to our device buffers. This is done in Listing 7.9.

```

1 // Copy host data to device data
2 status = clEnqueueWriteBuffer(cmd_queue, buffer_A, CL_FALSE, 0,
3     data_size, A.data(), 0, nullptr, nullptr);
4 status = clEnqueueWriteBuffer(cmd_queue, buffer_B, CL_FALSE, 0,
5     data_size, B.data(), 0, nullptr, nullptr);

```

Listing 7.9: Copying Host Memory to Device Memory

We use `clEnqueueWriteBuffer` to write our data to the relevant buffers. The third parameter is an interesting parameter as it tells OpenCL whether the application should wait for the write to complete. Typically we probably do not worry about a write completing - but we normally do for reading.

The final stage is setting the kernel arguments. Remember our arguments for the kernel are as follows:

A this is an input vector

B this is an input vector

C this is an output vector

We can set the kernel arguments as shown in Listing 7.10.

```
1 // Set the kernel arguments
2 status = clSetKernelArg(kernel, 0, sizeof(cl_mem), &buffer_A);
3 status |= clSetKernelArg(kernel, 1, sizeof(cl_mem), &buffer_B);
4 status |= clSetKernelArg(kernel, 2, sizeof(cl_mem), &buffer_C);
```

Listing 7.10: Setting OpenCL Kernel Arguments

We need the `cl_kernel` object, the parameter number (0 indexed), the size of the parameter, and a pointer to the parameter value.

We also need to make sure we free our buffers. We do this at the end of the main application as shown in Listing 7.11.

```
1 // Free OpenCL resources
2 clReleaseMemObject(buffer_A);
3 clReleaseMemObject(buffer_B);
4 clReleaseMemObject(buffer_C);
5 clReleaseCommandQueue(cmd_queue);
6 clReleaseContext(context);
7 clReleaseKernel(kernel);
8 clReleaseProgram(program);
```

Listing 7.11: Freeing OpenCL Memory Buffers

Running this application still will not do anything - we have not executed the kernel and got our results back.

7.5 Running and Getting Results from OpenCL Kernels

We are finally ready to run our OpenCL kernel. To do this we need to define a new piece of information - the dimensions of the work. In our instance we have a single dimension arrays with elements items. We can therefore set up our work size as shown in Listing 7.12.

```
1 // Configure the work dimensions - 1D of elements
2 array<size_t, 1> global_work_size = { elements };
```

Listing 7.12: Defining OpenCL Work Dimensions

We can now run our kernel. We do this in Listing 7.13.

```

1 // Enqueue the kernel for execution
2 status = clEnqueueNDRangeKernel(cmd_queue, kernel, 1, nullptr,
    global_work_size.data(), nullptr, 0, nullptr, nullptr);

```

Listing 7.13: Enqueuing a OpenCL Kernel for Execution

At the moment we are only interested in some of these parameters. These are:

`cmd_queue` the command queue we are using to enqueue the work

`kernel` the kernel we are executing

`1` the number of dimensions for the work

`global_work_size.data()` the number of elements per dimension

We will look at some of the other parameters as we work through the rest of the module. Finally we need to copy our data back from the kernel at the end. We do this as follows:

```

1 // Read the output buffer from the GPU to main memory
2 clEnqueueReadBuffer(cmd_queue, buffer_C, CL_TRUE, 0, data_size, C.
    data(), 0, nullptr, nullptr);

```

Listing 7.14: Reading Results Back from GPU Memory in OpenCL

Again we are only interested in a few parameters:

`cmd_queue` the command queue

`buffer_C` the buffer we are reading from

`CL_TRUE` we will wait for the read to complete

`0` the offset of the buffer to read from

`data_size` the amount of data to read from the buffer

`C.data()` pointer to the host memory to copy the device buffer to

Now all we want to do is validate that the data read back is correct. We can do this using the following code:

```

1 // Verify the output
2 auto result = true;
3 int i = 0;
4 // Iterate through each value in result array
5 for (auto &e : C)
6 {
7     // Check value
8     if (e != i + i)
9     {
10         result = false;
11         break;
12     }
13     ++i;
14 }
15
16 // Check if result is true and display accordingly
17 if (result)
18     cout << "Output is correct" << endl;

```

```
19| else
20|     cout << "Output is incorrect" << endl;
```

Listing 7.15: Verifying OpenCL Results

Running this application will provide the output shown in Figure 7.2.

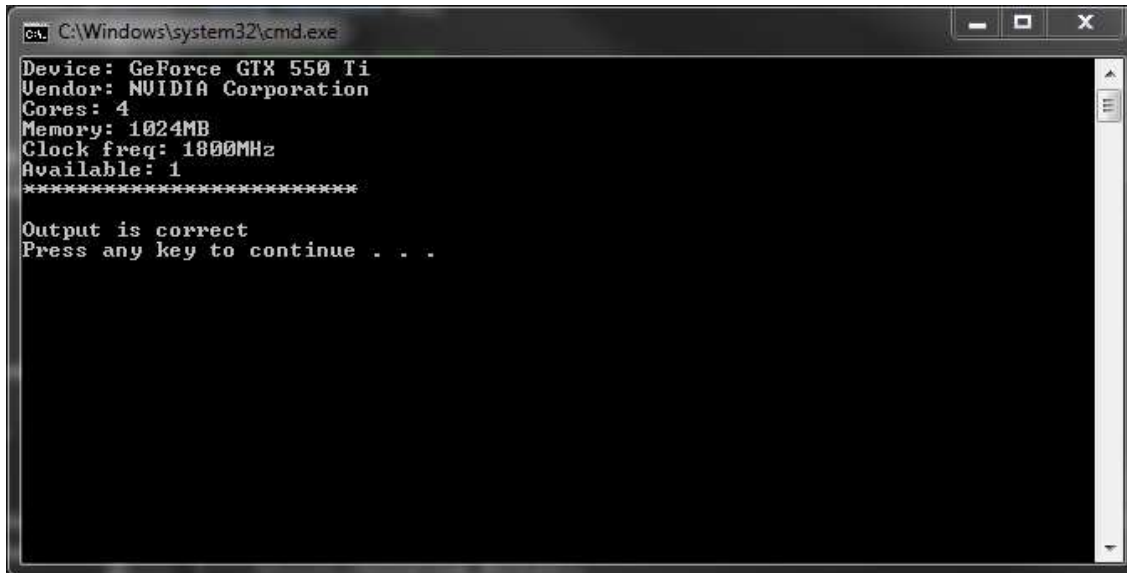


Figure 7.2: Output from OpenCL Application

We can break down our application as follows:

1. Initialise
2. Load kernel
3. Create host memory
4. Create device memory and copy host memory to device
5. Set kernel arguments
6. Run kernel remember to set the work dimensions
7. Copy device memory back to host to get the results
8. Clean up resources

This will be our quite standard approach to working with OpenCL. We might iterate through some of these stages (running kernels and setting / getting results), but typically the process is the same.

7.6 Matrix Multiplication

This is more of an exercise than a straight tutorial. The kernel we are using is given in Listing 7.16.

```
1 __kernel void simply_multiply(__global float *output_C, unsigned
  int width_A, unsigned int height_A, unsigned int width_B,
  unsigned int height_B, __global float *input_A, __global float *
  input_B)
2 {
3     // Get global position in Y direction
4     unsigned int row = get_global_id(1);
5     // Get global position in X direction
6     unsigned int col = get_global_id(0);
7
8     float sum = 0.0f;
9
10    // Calculate result of one element of matrix C
11    for (unsigned int i = 0; i < width_A; ++i)
12        sum += input_A[row * width_A + i] * input_B[i * width_B + col];
13
14    // Store result in matrix C
15    output_C[row * width_B + col] = sum;
16 }
```

Listing 7.16: OpenCL Matrix Multiplication Kernel

You have enough information to run this kernel. I would recommend using matrices that are square and setting all the values of the matrices to 1. The value of each element of the output matrix will then be the dimension of the square matrix (e.g. a 64×64 matrix will mean each element of the output is 64).

If you get really stuck, the Heterogeneous Computing with OpenCL book will help.

Unit 8

GPU Programming with CUDA

OpenCL is the equivalent of OpenGL for GPU programming. It runs on pretty much all hardware and has tools provided by a number of vendors. This generality can come at the cost of simplicity and performance. CUDA can be considered the DirectX of GPU programming. It is proprietary (it is provided by Nvidia to run on Nvidia hardware). Tools are from Nvidia. However, there is a simplicity and performance benefit because of this. You will find the general ideas are the same. Our setup is a bit different however.

8.1 Getting Started with CUDA

If you have the CUDA SDK installed, you should have Nvidia Nsight added to Visual Studio. This being the case, you will be able to create a new CUDA application by selecting it during the project creation (under NVIDIA in the templates). Visual Studio will provide an initial kernel. You should delete this code and start from an empty file.

First we need to write an application to initialise CUDA. This is shown in Listing 8.1.

```
1 #include <cuda_runtime.h>
2 #include <device_launch_parameters.h>
3
4 int main()
5 {
6     // Initialise CUDA - select device
7     cudaSetDevice(0);
8
9     return 0;
10 }
```

Listing 8.1: Initialising a CUDA Application

And that is it. Much simpler than OpenCL. We only need to select a device. If your machine only has one Nvidia device, this is just 0. You can run this application to test it just to ensure everything is set up OK. Let us now output some information from CUDA.

8.2 Getting CUDA Info

Getting information from CUDA is also fairly trivial. Listing 8.2 is our information operation.

```

1 void cuda_info()
2 {
3     // Get CUDA device
4     int device;
5     cudaGetDevice(&device);
6
7     // Get CUDA device properties
8     cudaDeviceProp properties;
9     cudaGetDeviceProperties(&properties, device);
10
11    // Display properties
12    cout << "Name: " << properties.name << endl;
13    cout << "CUDA Capability: " << properties.major << "." <<
        properties.minor << endl;
14    cout << "Cores: " << properties.multiProcessorCount << endl;
15    cout << "Memory: " << properties.totalGlobalMem / (1024 * 1024)
        << "MB" << endl;
16    cout << "Clock freq: " << properties.clockRate / 1000 << "MHz" <<
        endl;
17 }

```

Listing 8.2: Getting CUDA Information

You will need to call the operation `cuda_info` from the main application. Doing so will give you an output similar to Figure 8.1.



```

C:\Windows\system32\cmd.exe
Name: GeForce GTX 550 Ti
CUDA Capability: 2.1
Cores: 4
Memory: 1024MB
Clock freq: 1800MHz
Press any key to continue . . .

```

Figure 8.1: CUDA Device Information

As you can see, this is the same information achieved from calling OpenCL. So why do Nvidia state that they have 192 CUDA cores for the graphics card shown? Each multiprocessor (we have 4) has a number of streaming processors (processors that can execute an instruction for us). Each multiprocessor in the graphics card has in fact 48 of these. Depending on the CUDA capability of your graphics card, you will have a different number of streaming processors per multiprocessor. Table 8.1 illustrates the various capabilities.

A GTX 780 has Kepler architecture and has 12 multiprocessors. This means that the 780 will have 2304 cores in total, running at approximately 862 MHz. This means that we have close to 2 THz of performance.

Microarchitecture	CUDA Capability	SP per MP
Tesla	1.0 - 1.3	8
Fermi	2.0	32
Fermi	2.1	48
Kerpler	3.0 - 3.5	192
Maxwell	5.0	128

Table 8.1: CUDA Capabilities and Core Counts

8.3 CUDA Kernels

One of the main differences CUDA provides from OpenCL is that we are using a single file solution. That means that we will write our CUDA kernel in the same file as our main method. This keeps us closer to standard C++ development, and we do not need to load and compile external. Therefore, Listing 8.3 should be added to your main file.

```

1 __global__ void vecadd(const int *A, const int *B, int *C)
2 {
3     // Get block index
4     unsigned int block_idx = blockIdx.x;
5     // Get thread index
6     unsigned int thread_idx = threadIdx.x;
7     // Get the number of threads per block
8     unsigned int block_dim = blockDim.x;
9     // Get the thread's unique ID - (block_idx * block_dim) +
        thread_idx;
10    unsigned int idx = (block_idx * block_dim) + thread_idx;
11    // Add corresponding locations of A and B and store in C
12    C[idx] = A[idx] + B[idx];
13 }

```

Listing 8.3: CUDA Vector Addition Kernel

We will get to how we launch the kernel soon. First let us look at memory management between the CPU and GPU.

8.4 Passing Data to CUDA

As with OpenCL, we have to work between host memory (main memory) and device memory (GPU memory). If you remember with OpenCL we first declared and initialised some host memory. We will do the same this time. Listing 8.4 provides the code for your main function.

```

1 // Create host memory
2 auto data_size = sizeof(int) * ELEMENTS;
3 vector<int> A(ELEMENTS);    // Input array
4 vector<int> B(ELEMENTS);    // Input array
5 vector<int> C(ELEMENTS);    // Output array
6
7 // Initialise input data
8 for (unsigned int i = 0; i < ELEMENTS; ++i)
9     A[i] = B[i] = i;

```

Listing 8.4: Creating Host Memory for CUDA

We also need to initialise memory on our device. Listing 8.5 shows how we undertake this.

```

1 // Declare buffers
2 int *buffer_A, *buffer_B, *buffer_C;
3
4 // Initialise buffers
5 cudaMalloc((void**)&buffer_A, data_size);
6 cudaMalloc((void**)&buffer_B, data_size);
7 cudaMalloc((void**)&buffer_C, data_size);

```

Listing 8.5: Allocating Device Memory with CUDA

Notice that we don't need any special types with CUDA. We simply declare an `int` pointer as standard. The only difference is in how we allocate memory. You may be familiar with `malloc` from standard C. `cudaMalloc` undertakes the same functionality but for allocating memory on the GPU.

All we need to do now is copy the memory from the host to the device. We do this using the `cudaMemcpy` operation:

```

1 cudaMemcpy(dest, src, size, direction);

```

The `direction` value is used to tell CUDA which way the data is being copied (host to device, device to host, device to device). For our needs, we use the code shown in Listing 8.6.

```

1 // Write host data to device
2 cudaMemcpy(buffer_A, &A[0], data_size, cudaMemcpyHostToDevice);
3 cudaMemcpy(buffer_B, &B[0], data_size, cudaMemcpyHostToDevice);

```

Listing 8.6: Copying Host Memory to the Device in CUDA

8.5 Running and Getting Results from CUDA

Now we just need to run the kernel. We do this as shown in Listing 8.7.

```

1 // Run kernel with one thread for each element
2 // First value is number of blocks, second is threads per block.
   Max 1024 threads per block
3 vecadd<<<ELEMENTS / 1024, 1024>>>(buffer_A, buffer_B, buffer_C);
4
5 // Wait for kernel to complete
6 cudaDeviceSynchronize();

```

Listing 8.7: Running the CUDA Vector Addition Kernel

Notice that it looks very similar to running the operation normally. The only difference is that we are defining the number of blocks (`ELEMENTS / 1024`) and the number of threads per block (1024). The call to `cudaDeviceSynchronize` means that we wait for the kernel to complete executing before continuing.

To get our results back, we simply call `cudaMemcpy` again. Listing 8.8 provides the necessary call.

```

1 // Read output buffer back to the host
2 cudaMemcpy(&C[0], buffer_C, data_size, cudaMemcpyDeviceToHost);

```

Listing 8.8: Copying Device Memory to the Host in CUDA

You should write the code to check that the output is correct. It is the same as the code in the OpenCL example.

8.6 Freeing Resources

The last thing our application has to do is free any resources used by CUDA. As we have only allocated memory, this requires only a few calls. Listing 8.9 provides the necessary code.

```
1 // Clean up resources
2 cudaFree(buffer_A);
3 cudaFree(buffer_B);
4 cudaFree(buffer_C);
```

Listing 8.9: Freeing CUDA Memory

Running this application should give you the same output as that in the OpenCL version.

8.7 Matrix Multiplication

As with OpenCL, your task here is to write and test the application required to multiply two matrices using CUDA. The kernel code is as shown in Listing 8.10.

```
1 __global__ void simple_multiply(float *output_C, unsigned int
   width_A, unsigned int height_A, unsigned int width_B, unsigned
   int height_B, const float *input_A, const float *input_B)
2 {
3     // Get global position in Y direction
4     unsigned int row = (blockIdx.y * 1024) + threadIdx.y;
5     // Get global position in X direction
6     unsigned int col = (blockIdx.x * 1024) + threadIdx.x;
7
8     float sum = 0.0f;
9
10    // Calculate result of one element of matrix C
11    for (unsigned int i = 0; i < width_A; ++i)
12        sum += input_A[row * width_A + i] * input_B[i * width_B + col];
13
14    // Store result in matrix C
15    output_C[row * width_B + col] = sum;
16 }
```

Listing 8.10: Matrix Multiply in CUDA

Unit 9

GPU Programming Examples

In this unit we are going to look at some GPU samples. We will look at OpenCL and CUDA examples where possible. The general approaches are the same. We will start by looking at some of the problems we have already looked at, before moving onto some other ideas. Most of these examples will be only briefly discussed, as you should be able to implement the necessary applications by now. You should also be taking timings to compare performance with sequential, multithreaded, MPI, and other solutions.

9.1 Monte Carlo π

Monte Carlo π was our first problem examined, so you should have a good understanding of the principle by now. We will look at one OpenCL and a number of CUDA solutions. OpenCL can handle most of the CUDA approaches as well, so you can implement these too if you wish.

9.1.1 OpenCL Monte Carlo π

For this application you will have to generate some random values and pass them to the GPU. OpenCL comes with a type that can be used to represent a 2D position in space - `float2`. This is the type used in the kernel. For our main application, the type is `cl_float2`. The general approach we are taking is as follows:

1. Generate random points on CPU - one point for each thread
2. Allocate memory for points and a result value (0 or 1) for each thread
3. Copy points to GPU
4. Run kernel
5. Get results back from GPU
6. Sum results
7. Calculate π

You are only provided with the kernel for point point 4. It is up to you to implement the necessary main application. Listing 9.1 provides the kernel code.

```

1  __kernel void monte_carlo_pi(__global float2 *points, __global char
    *results)
2  {
3      // Get our work id
4      unsigned int id = get_global_id(0);
5
6      // Get the point to work on
7      float2 point = points[id];
8      // Calculate the length - built-in OpenCL function
9      float l = length(point);
10     // Result is either 1 or 0
11     if (l <= 1.0f)
12         results[id] = 1;
13     else
14         results[id] = 0;
15 }

```

Listing 9.1: OpenCL Version of Monte Carlo π

9.1.2 CUDA Monte Carlo π

For CUDA, we will look at a few different approaches which allow us to examine potential performance gains. We will implement the same solution as OpenCL first. Then we will look at a solution using a `for` loop within a kernel, generating random values using CUDA, and finally summing on the GPU rather than the CPU.

Standard Approach

Our standard approach follows the same method as we undertook for OpenCL. Therefore, the kernel is just provided. This is shown in Listing 9.2.

```

1  __global__ void monte_carlo_pi(const float2 *points, char *results)
2  {
3      // Calculate index
4      unsigned int idx = (blockIdx.x * blockDim.x) + threadIdx.x;
5
6      // Get the point to work on
7      float2 point = points[idx];
8      // Calculate the length - not built-in
9      float l = sqrtf((point.x * point.x) + (point.y * point.y));
10     // Check if in circle
11     if (l <= 1.0)
12         results[idx] = 1;
13     else
14         results[idx] = 0;
15 }

```

Listing 9.2: CUDA Version of Monte Carlo π

Notice that CUDA does not provide a `length` function like OpenCL, so we have to calculate the length manually.

Using a `for` Loop in the Kernel

Our next implementation is going to move towards our CPU implementation of Monte Carlo π . We do this by undertaking a number of iterations on each thread rather than a thread calculating a single point. For example, if we consider a solution

where we have 2^{24} points to check, we need to run 2^{24} threads and copy back 2^{24} bytes (16 MBytes) of data to main memory to sum the result.

If we allow each thread to calculate a number of iterations, we increase the amount of work a single thread does, while reducing the number of threads run and decreasing the memory usage. For example, if we have 2^{24} points to calculate and let each thread run 2^{16} iterations (so each thread calculates 2^{16} points), then we only run 2^8 threads, and only copy back $2^8 \times 4$ (we need an `int` now) bytes (1024 bytes or 0.001 MBytes). The kernel in this instance is shown in Listing 9.3.

```

1  __global__ void monte_carlo_pi(unsigned int iterations, float2 *
    points, int *results)
2  {
3      // Calculate index
4      unsigned int idx = (blockIdx.x * blockDim.x) + threadIdx.x;
5      // Calculate starting point
6      unsigned int start = idx * iterations;
7      // Calculate end point
8      unsigned int end = start + iterations;
9      // Set starting result to 0
10     results[idx] = 0;
11     // Loop for iterations
12     for (unsigned int i = start; i < end; ++i)
13     {
14         // Get the point to work on
15         float2 point = points[i];
16         // Calculate the length
17         float l = sqrtf((point.x * point.x) + (point.y * point.y));
18         // Check length and add to result accordingly
19         if (l <= 1.0f)
20             ++results[idx];
21     }
22 }
```

Listing 9.3: CUDA Version of Monte Carlo π Using a for Loop

The same approach is required as before, except you have to consider the amount of data required to copy back and the number of threads against the iterations per thread. You should experiment with different configurations to explore the different performance characteristics. Also try different thread to block ratios to further analyse performance. You will be surprised how some tweaks can alter performance.

Using CUDA to Generate Random Points

We can also get CUDA to generate our random values for us directly on the GPU. To do this we will need to use *cuRand* (<http://docs.nvidia.com/cuda/curand/>). You will need to read a little bit about how to use *cuRand*, but example code for generating random values (in your main application - not the kernel). Listing 9.4 provides the random value generation code. Note that this approach generates random values across 2-dimensions (since we have 2-dimensional values). Other approaches could also be used. Read more on *cuRand* to find out.

```

1  // Create random values on the GPU
2  // Create generator
3  curandGenerator_t rnd;
4  curandCreateGenerator(&rnd, CURAND_RNG_QUASI_SOBOL32);
5  curandSetQuasiRandomGeneratorDimensions(rnd, 2);
6  curandSetGeneratorOrdering(rnd, CURAND_ORDERING_QUASI_DEFAULT);
```

```

7
8 // Generate random numbers - point_buffer is an allocated device
  buffer
9 curandGenerateUniform(rnd, (float*)point_buffer, 2 * NUM_POINTS);
10
11 // Destroy generator
12 curandDestroyGenerator(rnd);

```

Listing 9.4: Generating 2-dimensional Random Numbers Using CUDA

You can use the same kernel as previously. All you are changing is how you are generating random values.

Summing on the Kernel

Our final approach to Monte Carlo π involves us performing the final stage of the computation (the calculation of π) on the GPU. To do this, we will use a new function within our kernel - `__syncthreads`. This function allows us to stop all threads within a block at a particular point in our code. *Note that only threads in the same block will synchronise.* This means that if you have more than one block, not all threads will synchronise. For our implementation to work then you can only have one block. Our kernel this time is given in Listing 9.5.

```

1 __global__ void monte_carlo_pi(unsigned int iterations, float2 *
  points, char *results, double *pi)
2 {
3   // Calculate index
4   unsigned int idx = (blockIdx.x * blockDim.x) + threadIdx.x;
5   // Calculate starting point
6   unsigned int start = idx * iterations;
7   // Calculate end point
8   unsigned int end = start + iterations;
9   // Loop for iterations
10  for (unsigned int i = start; i < end; ++i)
11  {
12    // Get the point to work on
13    float2 point = points[i];
14    // Calculate the length
15    float l = sqrtf((point.x * point.x) + (point.y * point.y));
16    // Check length and add to result accordingly
17    if (l <= 1.0f)
18      ++results[idx];
19  }
20  // Sync threads in block
21  __syncthreads();
22  // If idx = 0 perform the sum
23  if (idx == 0)
24  {
25    // Number of threads in block
26    unsigned int elements = blockDim.x;
27    // Sum result
28    unsigned int sum = 0;
29    for (unsigned int i = 0; i < elements; ++i)
30      sum += results[i];
31
32    // Calculate pi
33    pi[0] = (4.0 * (double)sum) / (double)elements;
34  }
35 }

```

 Listing 9.5: CUDA Version of Monte Carlo π With Summing on GPU

Note that you only have to copy back one value - the `float` containing π . Our host actually needs no buffers of data allocated if you are generating random values using CUDA as well. All data is just allocated on the GPU.

9.1.3 Exercises

1. Can you implement OpenCL versions of the different CUDA approaches? The random number generation approach you will likely need a library.
2. CUDA also allows you to generate random numbers within a kernel. Find out how to do this and reimplement the summing solution using this approach. Gather timings (taking into account random generation time for the original summing solution) and draw some conclusions.
3. In all cases we have calculated the length using standard Pythagoras. This can be slightly optimised very trivially. Try and do this and see the difference in results.

9.2 Mandelbrot Fractal

The Mandelbrot fractal we introduced when working with futures. We used a particular technique then called the *Escape Time Algorithm*. We can also implement this on the GPU. Let us look at examples in OpenCL and CUDA.

9.2.1 OpenCL Mandelbrot

Our kernel this time involves loops and branching statements - which might be an issue for performance. The kernel code is given in Listing 9.6.

```

1 // This kernel is adapted from Ben Hiller's code on Github
2 // https://github.com/benhiller/opencl-mandelbrot
3
4 // Convert the raw x coordinates [0, 1] into a scaled coordinate
5 // [0, 1] -> [-2, 1.25]
6 float mapX(float x)
7 {
8     return x * 3.25f - 2.0f;
9 }
10
11 // Same purpose as mapX
12 // [0, 1] -> [-1.25, 1.25]
13 float mapY(float y)
14 {
15     return y * 2.5f - 1.25f;
16 }
17
18 __kernel void mandelbrot(__global char *out)
19 {
20     int x_dim = get_global_id(0);
21     int y_dim = get_global_id(1);
22     size_t width = get_global_size(0);
23     size_t height = get_global_size(1);
  
```

```

24  int idx = width * y_dim + x_dim;
25
26  float x_origin = mapX((float)x_dim / (float)width);
27  float y_origin = mapY((float)y_dim / (float)height);
28
29  // Escape time algorithm. Follows the pseudocode from Wikipedia
   _very_ closely
30  float x = 0.0f;
31  float y = 0.0f;
32
33  int iteration = 0;
34
35  // This can be changed to be more or less precise
36  int max_iteration = 256;
37
38  // While loop - is this the best option???
39  while (x * x + y * y <= 4 && iteration < max_iteration)
40  {
41      float xtemp = x * x - y * y + x_origin;
42      y = 2 * x * y + y_origin;
43      x = xtemp;
44      ++iteration;
45  }
46
47  if (iteration == max_iteration)
48      // This coordinate did not escape, so is in the Mandelbrot set
49      out[idx] = 0;
50  else
51      // This coordinate did escape, so colour based on how quickly
   it escaped
52      out[idx] = iteration;
53 }

```

Listing 9.6: OpenCL Mandelbrot Kernel

Our main application just needs to declare our data that we read back into and use for our image. Listing 9.7 provides the core part of the main code file.

```

1  // Necessary values to run the kernel
2  const cl_uint dimension = 8192;
3  const cl_uint elements = dimension * dimension;
4
5  // Host data
6  vector<cl_char> results(elements);
7
8  // Create device buffer
9  cl_mem buffer_results = clCreateBuffer(context, CL_MEM_WRITE_ONLY
   , elements * sizeof(cl_char), nullptr, &status);
10
11  // Set the kernel arguments
12  clSetKernelArg(kernel, 0, sizeof(cl_mem), &buffer_results);
13
14  // Configure work-item structure
15  array<size_t, 2> globalWorkSize = { dimension, dimension };
16
17  // Enqueue the kernel for execution
18  status = clEnqueueNDRangeKernel(cmd_queue, kernel, 2, nullptr,
   globalWorkSize.data(), nullptr, 0, nullptr, nullptr);
19
20  // Read the output buffer back to the host

```

```

21 | clEnqueueReadBuffer(cmd_queue, buffer_results, CL_TRUE, 0,
    |     elements * sizeof(cl_char), &results[0], 0, nullptr, nullptr);

```

Listing 9.7: Running the Mandelbrot Kernel

Saving an Image Using FreeImage

You can save the data generated by the Mandelbrot kernel by using the code given in Listing 9.8.

```

1 | // Save the image
2 | // Initialise FreeImage
3 | FreeImage_Initialise();
4 | // Allocate the data
5 | FIBITMAP *bitmap = FreeImage_Allocate(dimension, dimension, 24);
6 | RGBQUAD colour;
7 | // Loop round the data
8 | for (int x = 0; x < dimension; ++x)
9 | {
10 |     for (int y = 0; y < dimension; ++y)
11 |     {
12 |         char col = results[(y * dimension) + x];
13 |         colour.rgbRed = colour.rgbGreen = colour.rgbBlue = col;
14 |         FreeImage_SetPixelColor(bitmap, x, y, &colour);
15 |     }
16 | }
17 | // Save
18 | FreeImage_Save(FIF_PNG, bitmap, "mandelbrot.png", 0);
19 | // Deinitialise
20 | FreeImage_DeInitialise();

```

Listing 9.8: Saving Mandelbrot Data Using FreeImage

You will have to download and set-up FreeImage to achieve this.

9.2.2 Exercise

You should be able to convert the OpenCL version of Mandelbrot to a CUDA version fairly trivially. Do this now to ensure you understand the basic concepts behind making a CUDA application.

9.3 Trapezoidal Rule

Our next example returns to the trapezoidal rule application we looked at with OpenMP. This time, we will use CUDA. Remember that the trapezoidal rule works by calling a particular function that we sample based on a value based on the thread ID. To do this, we need to introduce another idea for CUDA - device callable functions.

Up until now with CUDA we have defined functions using `__global__`. This defines code that runs on the GPU and is callable from our main application. For code that runs on the GPU and is callable from other GPU code we use `__device__`. The code in Listing 9.9 illustrates this.

```

1 | __device__ double func(double x)
2 | {
3 |     return sin(x);

```

```

4 }
5
6 __global__ void trap(double start, double end, unsigned int
    iterations, double *results)
7 {
8     // Get global thread ID
9     unsigned int idx = (blockIdx.x * blockDim.x) + threadIdx.x;
10    // Get number of threads
11    unsigned int thread_count = gridDim.x * blockDim.x;
12    // Calculate iteration slice size
13    double slice_size = (end - start) / iterations;
14    // Calculate number of iterations per thread
15    unsigned int iterations_thread = iterations / thread_count;
16    // Calculate this thread's start point
17    double local_start = start + ((idx * iterations_thread) *
        slice_size);
18    // Calculate this thread's end point
19    double local_end = local_start + iterations_thread * slice_size;
20    // Calculate initial result
21    results[idx] = (func(local_start) + func(local_end)) / 2.0;
22    // Declare x before the loop - stops it being allocated and
        destroyed each iteration
23    double x;
24    // Sum each iteration
25    for (unsigned int i = 0; i <= iterations_thread - 1; ++i)
26    {
27        // Calculate next slice to calculate
28        x = local_start + i * slice_size;
29        // Add to current result
30        results[idx] += func(x);
31    }
32    // Multiply the result by the slice size
33    results[idx] *= slice_size;
34 }

```

Listing 9.9: Trapezoidal Rule using CUDA

You will have to write the rest of the CUDA code yourself, but you should be able to do this easily enough.

9.3.1 Exercise

This time write the equivalent application in OpenCL. This again should be fairly trivial.

9.4 Image Rotation

Let us now look at a new example - image rotation. Image rotation involves us working on a large buffer of data that represents our image, creating another large buffer of the same size, and setting the values in this buffer based on the following calculation:

$$destination(x, y) = source(x' \times \cos \theta + y' \times \sin \theta + x', y' \times \cos \theta - x' \times \sin \theta + y')$$

where

$$x' = x - \frac{width}{2}$$

$$y' = y - \frac{height}{2}$$

θ is the angle of rotation

An example of the output from the application is shown in Figure 9.1.

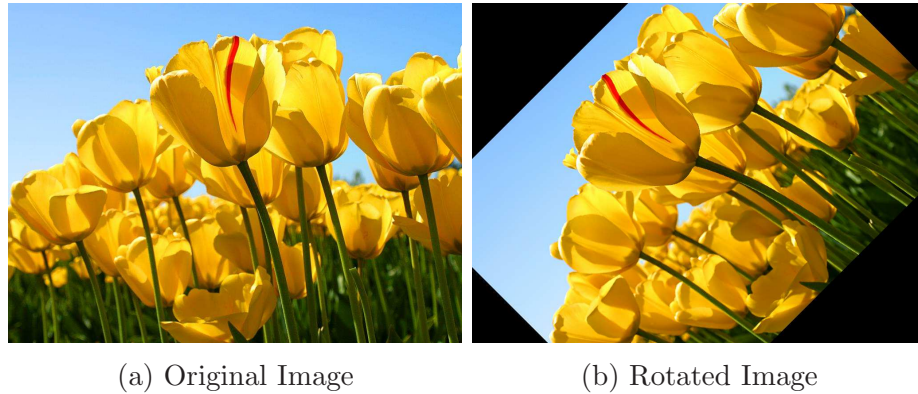


Figure 9.1: Rotating an Image

In OpenCL our kernel code is provided in Listing 9.10. It just calculates the pixel colour accordingly.

```

1  __kernel void image_rotate(__global int *dest_data, __global int *
    src_data, unsigned int width, unsigned int height, float
    sin_theta, float cos_theta)
2  {
3      // Get work item index - (x, y) coordinate
4      const int ix = get_global_id(0);
5      const int iy = get_global_id(1);
6
7      // Calculate location of data
8      float x0 = (float)width / 2.0f;
9      float y0 = (float)height / 2.0f;
10     float xOff = ix - x0;
11     float yOff = iy - y0;
12     int xpos = (int)(xOff * cos_theta + yOff * sin_theta + x0);
13     int ypos = (int)(yOff * cos_theta - xOff * sin_theta + y0);
14
15     // Bounds check - should we set some data?
16     if ((xpos >= 0) && (xpos < width) && (ypos >= 0) && (ypos <
        height))
17         dest_data[iy * width + ix] = src_data[ypos * width + xpos];
18 }

```

Listing 9.10: Image Rotation OpenCL Kernel

To use this kernel we need to load in the image. Here we use FreeImage as shown in Listing 9.11.

```

1  FreeImage_Initialise();
2

```



```

3  // Load an image
4  FREE_IMAGE_FORMAT format = FreeImage_GetFileType("Tulips.jpg");
5  FIBITMAP *image = FreeImage_Load(format, "Tulips.jpg", 0);
6  // Convert image to 32-bit - how kernel works
7  FIBITMAP *temp = image;
8  image = FreeImage_ConvertTo32Bits(image);
9  FreeImage_Unload(temp);
10
11 // Get the image data
12 int *bits = (int*)FreeImage_GetBits(image);
13 unsigned int width = FreeImage_GetWidth(image);
14 unsigned int height = FreeImage_GetHeight(image);
15 // Store in a vector - just makes our life easier
16 vector<int> image_data(bits, bits + (width * height));
17
18 // Create rest of host data
19 vector<int> result_data(width * height);

```

Listing 9.11: Loading and Image with FreeImage

Finally, the rest of our main code just sets up the various buffers and parameters for the kernel to execute. The code in Listing 9.12 provides this code.

```

1  // Create buffers
2  auto buffer_image = clCreateBuffer(context, CL_MEM_READ_ONLY,
3      sizeof(int) * image_data.size(), nullptr, &status);
4  auto buffer_result = clCreateBuffer(context, CL_MEM_WRITE_ONLY,
5      sizeof(int) * result_data.size(), nullptr, &status);
6
7  // Write host data to device
8  status = clEnqueueWriteBuffer(cmd_queue, buffer_image, CL_FALSE,
9      0, sizeof(int) * image_data.size(), &image_data[0], 0, nullptr,
10      nullptr);
11
12 // Set kernel arguments
13 float cos_theta = cos(PI / 4.0f);
14 float sin_theta = sin(PI / 4.0f);
15 status = clSetKernelArg(kernel, 0, sizeof(cl_mem), &buffer_result);
16 status = clSetKernelArg(kernel, 1, sizeof(cl_mem), &buffer_image);
17
18 status = clSetKernelArg(kernel, 2, sizeof(cl_uint), &width);
19 status = clSetKernelArg(kernel, 3, sizeof(cl_uint), &height);
20 status = clSetKernelArg(kernel, 4, sizeof(float), &cos_theta);
21 status = clSetKernelArg(kernel, 5, sizeof(float), &sin_theta);
22
23 // Configure the work item structure
24 array<size_t, 2> globalWorkSize = { width, height };
25
26 // Enqueue the kernel for execution
27 clEnqueueNDRangeKernel(cmd_queue, kernel, 2, nullptr,
28     globalWorkSize.data(), nullptr, 0, nullptr, nullptr);
29
30 // Read back result
31 clEnqueueReadBuffer(cmd_queue, buffer_result, CL_TRUE, 0, sizeof(
32     cl_int) * result_data.size(), &result_data[0], 0, nullptr,
33     nullptr);

```

Listing 9.12: Executing the Image Rotation Kernel

9.4.1 Exercises

1. We have used standard buffers with OpenCL here. However, OpenCL does support texture buffer objects. Investigate these and try and modify the application to utilise these concepts.
2. Implement the image rotation example using CUDA and textures. You will have to do a some research into this since we are using a number of new concepts.

9.5 Profiling and Debugging

There are a number of different tools you can use to analyse GPU performance. Here are a few links to get started:

- AMD CodeXL (OpenCL profiling on AMD hardware) [\[Link\]](#)
- Nvidia Nsight (CUDA and OpenCL profiling on Nvidia hardware) [\[Link\]](#)
- Intel Graphics Performance Analyzer (Intel profiling) [\[Link\]](#)

Unit 10

C++ AMP

This is the final unit of the practical part of the module. Here, we focus on another GPU programming technology - *C++ AMP*. C++ AMP is a technology developed by Microsoft to support GPU programming in standard C++ code. It is an open standard, although only Microsoft compilers really support it at present.

The main difference with C++ AMP is that we are only going to write C++ code. We will compile it as standard with no additional libraries and runtimes. We achieve this by using function objects and λ -expressions.

10.1 Getting C++ AMP Information

Let us start by looking at the header required for working with C++ AMP. This is the `amp.h` header:

```
1 #include <amp.h>
```

That is all we need to start working with C++ AMP. There are other headers we can add, but to get a basic application started we just need the header file.

Listing 10.1 provides code for getting information from C++ AMP. Notice that there isn't much we can get. The `accelerator` variable allows us to access details on the current device. Notice as well that we use `wcout` rather than `cout` to print details. `wcout` prints wide character (16-bit char) strings. The `accelerator` type and other C++ AMP operations and data types are part of the `concurrency` namespace.

```
1 void print_amp_info()
2 {
3     // Create accelerator
4     accelerator acc;
5     // Output properties - note the use of wcout (16-bit char strings)
6     wcout << "Device: " << acc.description << endl;
7     wcout << "Dedicated Memory: " << acc.dedicated_memory / 1024 << "
8         MB" << endl;
9     wcout << "Supports Shared Memory: " << acc.
10        supports_cpu_shared_memory << endl;
11 }
12 // This is all C++AMP provides
```

Listing 10.1: Displaying C++ AMP Information

We will use this operation in our main code. Let us now move onto a vector addition example.

10.2 Vector Addition Using C++ AMP

Using C++ AMP our vector addition example becomes very simple. This is due to our GPU code being written in our standard C++ code. We still need to go through the main steps when working with the GPU, but things are far simpler.

10.2.1 Declaring Host Memory

For our vector addition example declaring host memory is no different than in our other GPU examples. Listing 10.2 provides the general idea. Note that we have to state we are using the array type from the standard library (the `std::` part). This is because C++ AMP also declares an `array` type (just to confuse things).

```
1 // Create host memory
2 std::array<int, elements> A;
3 std::array<int, elements> B;
4 std::array<int, elements> C;
```

Listing 10.2: Declaring Host Memory Using `std::array`

10.2.2 Creating Device Memory

Device memory setup in C++ AMP is much easier. We just need to create an `array_view` object that contains our host memory. The copying and the allocation is handled in this data type. Listing 10.3 provides our device memory creation code.

```
1 // Create device buffers - stored on GPU
2 array_view<int, 1> buffer_A(elements, A);
3 array_view<int, 1> buffer_B(elements, B);
4 array_view<int, 1> buffer_C(elements, C);
```

Listing 10.3: Creating an `array_view` of Host Memory

The second value of the template (the number) provides the dimensions of the data. In our instance we only have one - a single vector. We can create different configurations as standard GPU code.

10.2.3 Vector Addition Function in C++ AMP

Our *kernel* code is shown in Listing 10.4. Notice the similarity to our other approaches, although now we a function object.

```
1 // Function we are going to run on the GPU
2 auto vecadd = [=](index<1> idx) restrict(amp)
3 {
4     // Get our index
5     unsigned int i = idx[0];
6     // Add the two vectors together
7     buffer_C[i] = buffer_A[i] + buffer_B[i];
8 };
```

Listing 10.4: `vecadd` Function in C++ AMP

There are a few parts that need explaining:

- `[=]` means that the function should automatically copy the required variables into the function scope. This means that we don't have to pass any parameters to call the function. They are automatically copied across.

- `index<1>` provides us with the id of the particular thread or work item. Here, we have only one dimension of work, but we could have more.
- `restrict(amp)` is an instruction to the compiler to ensure that the defined function meets the requirements of C++ AMP code.

10.2.4 Running the vecadd Function

To actually run the function we use the code shown in Listing 10.5.

```

1 // Execute the function
2 parallel_for_each(buffer_C.extent, vecadd);
3
4 // Wait for result on buffer_C
5 buffer_C.synchronize();

```

Listing 10.5: Running a C++ AMP Function

And that is it. We run the function, giving it the work dimensions (the `buffer_C.extent` part) and the function to run. We then wait on `buffer_C` to fill.

10.2.5 Complete Main for C++ AMP vecadd Application

The complete C++ AMP main code is below. It is short but performs the same operation as our previous two examples.

```

1 int main()
2 {
3     // Display AMP info
4     print_amp_info();
5
6     // Number of elements
7     const unsigned int elements = 2048;
8
9     // Create host memory
10    std::array<int, elements> A;
11    std::array<int, elements> B;
12    std::array<int, elements> C;
13
14    // Initialise input data
15    for (unsigned int i = 0; i < elements; ++i)
16        A[i] = B[i] = i;
17
18    // Create device buffers - stored on GPU
19    array_view<int, 1> buffer_A(elements, A);
20    array_view<int, 1> buffer_B(elements, B);
21    array_view<int, 1> buffer_C(elements, C);
22
23    // Function we are going to run on the GPU
24    auto vecadd = [=](index<1> idx) restrict(amp)
25    {
26        // Get our index
27        unsigned int i = idx[0];
28        // Add the two vectors together
29        buffer_C[i] = buffer_A[i] + buffer_B[i];
30    };
31
32    // Execute the function
33    parallel_for_each(buffer_C.extent, vecadd);

```

```

34
35 // Wait for result on buffer_C
36 buffer_C.synchronize();
37
38 // Verify the output
39 auto result = true;
40 int i = 0;
41 // Iterate through each value in result array
42 for (auto &e : C)
43 {
44     // Check value
45     if (e != i + i)
46     {
47         result = false;
48         break;
49     }
50     ++i;
51 }
52
53 // Check if result is true and display accordingly
54 if (result)
55     cout << "Output is correct" << endl;
56 else
57     cout << "Output is incorrect" << endl;
58
59 return 0;
60 }

```

Listing 10.6: Complete C++ AMP vecadd Example

10.3 Matrix Multiplication Using C++ AMP

```

1 void multiply_matrix(array_view<float, 2> &output_C, const
   array_view<float, 2> &input_A, const array_view<float, 2> &
   input_B)
2 {
3     // Function to run
4     auto simple_multiply = [=](index<2> idx) restrict(amp)
5     {
6         // Get row and column
7         unsigned int row = idx[0];
8         unsigned int col = idx[1];
9
10        float sum = 0.0f;
11
12        // Calculate result of one element of matrix C
13        for (int i = 0; i < input_A.extent[0]; ++i)
14            sum += input_A(row, i) * input_B(i, col);
15
16        // Store result in matrix C
17        output_C(row, col) = sum;
18    };
19
20    // Run the function...
21 }

```

10.4 Monte Carlo π Using C++ AMP

```

1 // A struct representing a 2-dimensional point
2 struct point
3 {
4     // Align memory at 4 bytes
5     // Not necessary here (float is 4 bytes) but good practice
6     __declspec(align(4)) float x;
7     __declspec(align(4)) float y;
8 };

```

```

1 // Number of elements to create
2 const unsigned int elements = pow(2, 24);
3
4 // Host data - create on heap
5 point *points = new point[elements];
6 int *results = new int[elements];
7
8 // Initialise points
9 default_random_engine e(duration_cast<milliseconds>(
10     system_clock::now().time_since_epoch()).count());
11 uniform_real_distribution<float> distribution(0.0f, 1.0f);
12 for (unsigned int i = 0; i < elements; ++i)
13 {
14     points[i].x = distribution(e);
15     points[i].y = distribution(e);
16 }
17
18 // Create device buffers - stored on GPU
19 array_view<point, 1> point_buffer(elements, points);
20 array_view<int, 1> result_buffer(elements, results);
21
22 // Monte Carlo pi function on the GPU
23 auto monte_carlo = [=](index<1> idx) restrict(amp)
24 {
25     // Get point to work on
26     point p = point_buffer[idx[0]];
27     // Calculate the squared length
28     float l = p.x * p.x + p.y * p.y;
29     // Result is either 1 or 0
30     if (l <= 1.0f)
31         result_buffer[idx[0]] = 1;
32     else
33         result_buffer[idx[0]] = 0;
34 };
35
36 // Execute the function
37 parallel_for_each(result_buffer.extent, monte_carlo);
38
39 // Wait for result on result buffer
40 result_buffer.synchronize();

```

10.5 Mandelbrot Using C++ AMP

```

1 // Dimensions
2 const unsigned int dimension = 4096;
3 const unsigned int elements = dimension * dimension;

```

```

4
5 // Host data
6 int *results = new int[elements];
7
8 // Device data
9 array_view<int, 2> result_buffer(dimension, dimension, results)
10 ;
11 auto mandelbrot = [=](index<2> idx) restrict(amp)
12 {
13     // Get x and y coordinate
14     int x_dim = idx[0];
15     int y_dim = idx[1];
16     // Calculate index
17     int index = dimension * y_dim + x_dim;
18
19     // Calculate x and y origin
20     float x_origin = ((float)x_dim / (float)dimension) * 3.25f
21                     - 2.0f;
22     float y_origin = ((float)y_dim / (float)dimension) * 2.5f -
23                     1.25f;
24
25     // Escape time algorithm
26     float x = 0.0f;
27     float y = 0.0f;
28     int iteration = 0;
29     int max_iteration = 256;
30
31     while (x * x + y * y <= 4 && iteration < max_iteration)
32     {
33         float xtemp = x * x - y * y + x_origin;
34         y = 2 * x * y + y_origin;
35         x = xtemp;
36         ++iteration;
37     }
38
39     if (iteration == max_iteration)
40         result_buffer[idx] = 0;
41     else
42         result_buffer[idx] = iteration;
43 };

```

10.6 Trapezoidal Rule Using C++ AMP

```

1 // Start and end values
2 double start = 0.0;
3 double end = PI;
4 // Number of trapezoids to generate
5 unsigned int trapezoids = static_cast<unsigned int>(pow(2, 24))
6 ;
7
8 // Create host memory
9 double *results = new double[NUM_THREADS];
10
11 // Declare buffers
12 array_view<double, 1> results_buffer(NUM_THREADS, results);
13
14 // Use precise_math package from C++ AMP

```



```

14 // There is also fast_math - less accurate
15 auto func = [](double value) restrict(amp) { return
    precise_math::cos(value); };
16
17 auto trap = [=](index<1> idx) restrict(amp)
18 {
19     // Calculate iteration slice size
20     double slice_size = (end - start) / trapezoids;
21     // Calculate number of iterations per thread
22     unsigned int iterations_thread = trapezoids / NUM_THREADS;
23     // Calculate this thread's start point
24     double local_start = start + ((idx[0] * iterations_thread)
        * slice_size);
25     // Calculate this thread's end point
26     double local_end = local_start + iterations_thread *
        slice_size;
27     // Calculate initial result
28     results_buffer[idx] = (func(local_start) + func(local_end))
        / 2.0;
29     // Declare x before the loop - stops it being allocated and
        destroyed each iteration
30     double x;
31     // Sum each iteration
32     for (unsigned int i = 0; i <= iterations_thread - 1; ++i)
33     {
34         // Calculate next slice to calculate
35         x = local_start + i * slice_size;
36         // Add to current result
37         results_buffer[idx] += func(x);
38     }
39     // Multiply the result by the slice size
40     results_buffer[idx] *= slice_size;
41 };

```

10.7 Image Rotation Using C++ AMP

10.8 Profiling and Debugging

10.9 Exercises

