Parallel Programming Report

This report will be divided into two sections. The first section will comprise lab reports that are completed each week. The second section will be a development report of the project I will be undertaking.

Table of Contents

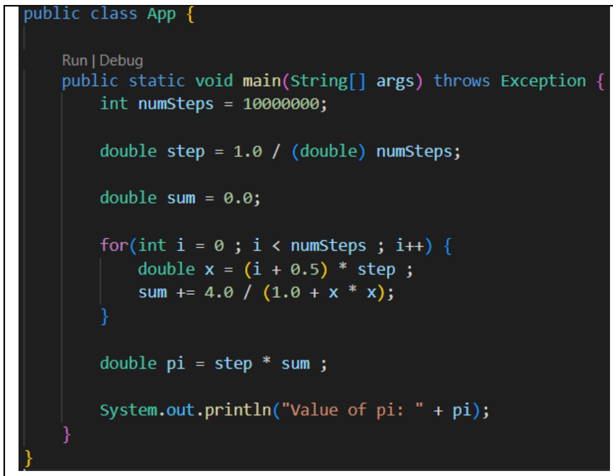
1 Lab Reports

1.1 Week 1 – Java Threads Reprise

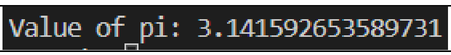
This week's lab is to act as refresher on using Java threads to speed up simple calculations using parallelism.

1.1.1 Calculating Pi

As this is the first week, not enough of the content has been covered therefore this lab session will be an introduction to parallel programming and a refresher to Java threads and how they speed up simple calculations through parallelism.

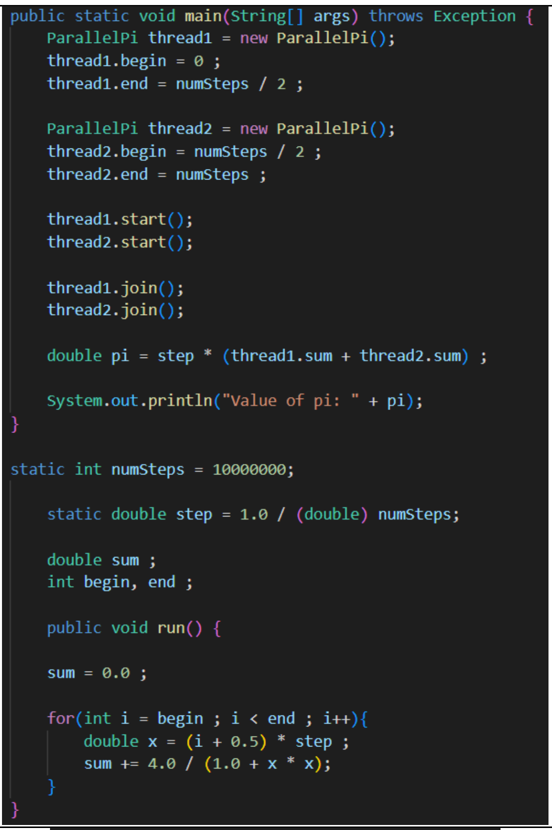


```java
public class App {

    Run | Debug
    public static void main(String[] args) throws Exception {
        int numSteps = 10000000;

        double step = 1.0 / (double) numSteps;

        double sum = 0.0;

        for(int i = 0 ; i < numSteps ; i++) {
            double x = (i + 0.5) * step ;
            sum += 4.0 / (1.0 + x * x);
        }

        double pi = step * sum ;

        System.out.println("Value of pi: " + pi);
    }
}
```

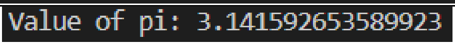


```
Value of pi: 3.141592653589731
```

Code for Calculating Pi Sequentially + Result



```java
public static void main(String[] args) throws Exception {
    ParallelPi thread1 = new ParallelPi();
    thread1.begin = 0 ;
    thread1.end = numSteps / 2 ;

    ParallelPi thread2 = new ParallelPi();
    thread2.begin = numSteps / 2 ;
    thread2.end = numSteps ;

    thread1.start();
    thread2.start();

    thread1.join();
    thread2.join();

    double pi = step * (thread1.sum + thread2.sum) ;

    System.out.println("Value of pi: " + pi);
}

static int numSteps = 10000000;

    static double step = 1.0 / (double) numSteps;

    double sum ;
    int begin, end ;

    public void run() {

    sum = 0.0 ;

    for(int i = begin ; i < end ; i++){
        double x = (i + 0.5) * step ;
        sum += 4.0 / (1.0 + x * x);
    }
}
```



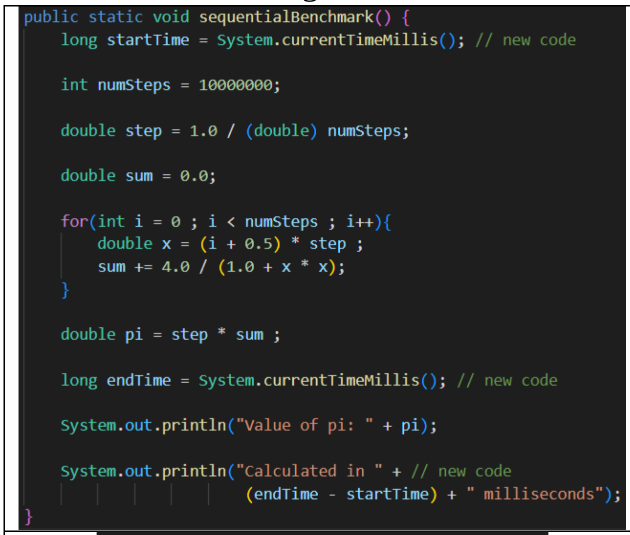
```
Value of pi: 3.141592653589923
```

Code for Calculating Pi through Parallelism + Result

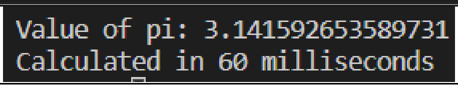
Figure 1.1: Calculation of Pi

The first tasks were to run the code for calculating pi and compare the results of the two. As seen in Figure 1.1 the result for pi comes out differently when calculating it sequentially and through parallelism. The result from sequential calculation is slightly lower than the result from calculating pi through parallelism.

1.1.2 Benchmarking

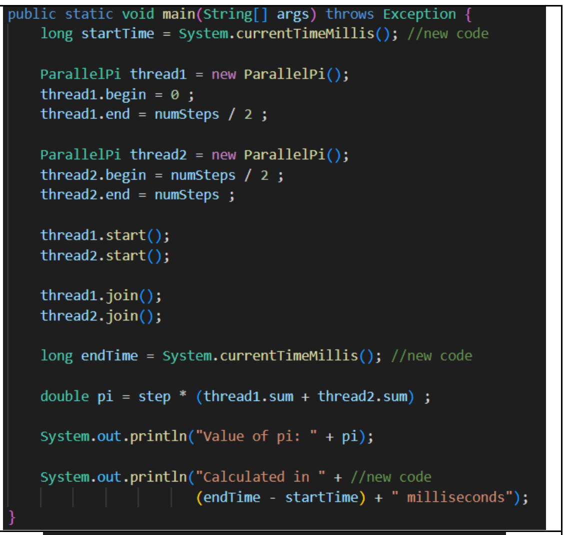


```java
public static void sequentialBenchmark() {
    long startTime = System.currentTimeMillis(); // new code

    int numSteps = 10000000;

    double step = 1.0 / (double) numSteps;

    double sum = 0.0;

    for(int i = 0 ; i < numSteps ; i++){
        double x = (i + 0.5) * step ;
        sum += 4.0 / (1.0 + x * x);
    }

    double pi = step * sum ;

    long endTime = System.currentTimeMillis(); // new code

    System.out.println("Value of pi: " + pi);

    System.out.println("Calculated in " + // new code
                    (endTime - startTime) + " milliseconds");
}
```

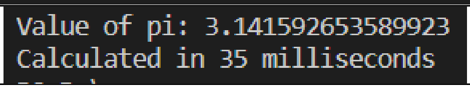


```
Value of pi: 3.141592653589731
Calculated in 60 milliseconds
```

Calculating Benchmark for Sequential Pi Code



```java
public static void main(String[] args) throws Exception {
    long startTime = System.currentTimeMillis(); //new code

    ParallelPi thread1 = new ParallelPi();
    thread1.begin = 0 ;
    thread1.end = numSteps / 2 ;

    ParallelPi thread2 = new ParallelPi();
    thread2.begin = numSteps / 2 ;
    thread2.end = numSteps ;

    thread1.start();
    thread2.start();

    thread1.join();
    thread2.join();

    long endTime = System.currentTimeMillis(); //new code

    double pi = step * (thread1.sum + thread2.sum) ;

    System.out.println("Value of pi: " + pi);

    System.out.println("Calculated in " + //new code
                    (endTime - startTime) + " milliseconds");
}
```



```
Value of pi: 3.141592653589923
Calculated in 35 milliseconds
```

Calculating Benchmark for Pi through Parallelism

Figure 1.2: Benchmarking the Calculation of Pi

After calculating pi, I then added some new lines of code to the main function of each calculation method. The new code is to allow for a measurement of how long it take for the program to calculate pi through both sequential calculation and through parallelism. As seen in Figure 1.2 calculating pi through parallelism is shown to take almost half the time than calculating pi sequentially.

$$\boxed{parallel\ speedup = \frac{sequential\ time}{parallel\ time}}$$

Figure 1.3: Parallel Speedup Formula

The parallel speedup that I calculated the parallel speedup to be 1.71428571429. This was calculated by having the sequential time be 60 and the parallel time to 35; both taken from Figure 1.2 and by using the equation from Figure 1.3.

numSteps Value	Sequential Time (ms)	Parallel Time (ms)	Parallel Speedup
1,000,000,000	10425	2931	3.55680655067
10,000,000	60	35	1. 71428571429
1,000,000	28	10	1.33333333…

Figure 1.4: Benchmark for multiple numSteps values

Figure 1.4 above shows parallel speedup times when the numSteps values have changed. As shown above, when the program is executed through parallelism, it is shown to be significantly faster when done using parallelism than when the program is executed sequentially.

1.1.3 Exercises

1.1.3.1 Exercise 1 – Quad-Core Speed

This exercise is meant to showcase the speed of calculating pi through parallelism using a quad-core rather than a dual-core. The program was executed on a laptop that has 6 cores. As shown in Figure 1.5 below, the times recorded are significantly faster than using dual-core; results found in Figure 1.4.

numSteps Value	Parallel Time using Quad-Core (ms)
1,000,000,000	1452
10,000,000	26
1,000,000	12

Figure 1.5: Calculating Parallel Time using Quad-Core Threading

1.1.3.2 Exercise 2 – Use of System.nanotime()

	Parallel Time	
numSteps Value	Time Taken (milliseconds / ms)	Time Taken (nanoseconds)
Dual-Core		
1,000,000,000	2931	2923542500
10,000,000	35	35608600
1,000,000	10	8218600
Quad-Core		
1,000,000,000	1452	1538538500
10,000,000	26	24528200
1,000,000	12	9995700

Figure 1.6: Calculating the Parallel Time in Nanoseconds

	Parallel Time	
numSteps value	Time Taken (nanoseconds)	Nanoseconds converted to milliseconds
Dual-Core		
1,000,000,000	2923542500	2923.5425
10,000,000	35608600	35.6086
1,000,000	8218600	8.2186
Quad-Core		
1,000,000,000	1538538500	1538.5385
10,000,000	24528200	24.5282
1,000,000	9995700	9.9957

Figure 1.7: Converting the Time in Nanoseconds to Milliseconds

As shown in both Figure 1.6 and Figure 1.7 above, it can be shown that when calculating the time in nanoseconds that it will be more accurate as when they are converted in milliseconds, the times are now shown with decimals allowing for a more accurate time when compared to the times calculated in just milliseconds when the program is ran.

1.2 Week 2 – Threads and Data

This week's objective is to speed up the execution of the calculation of the Mandelbrot set using multiple threads on multicore processor.

1.2.1 Calculating the Mandelbrot Set

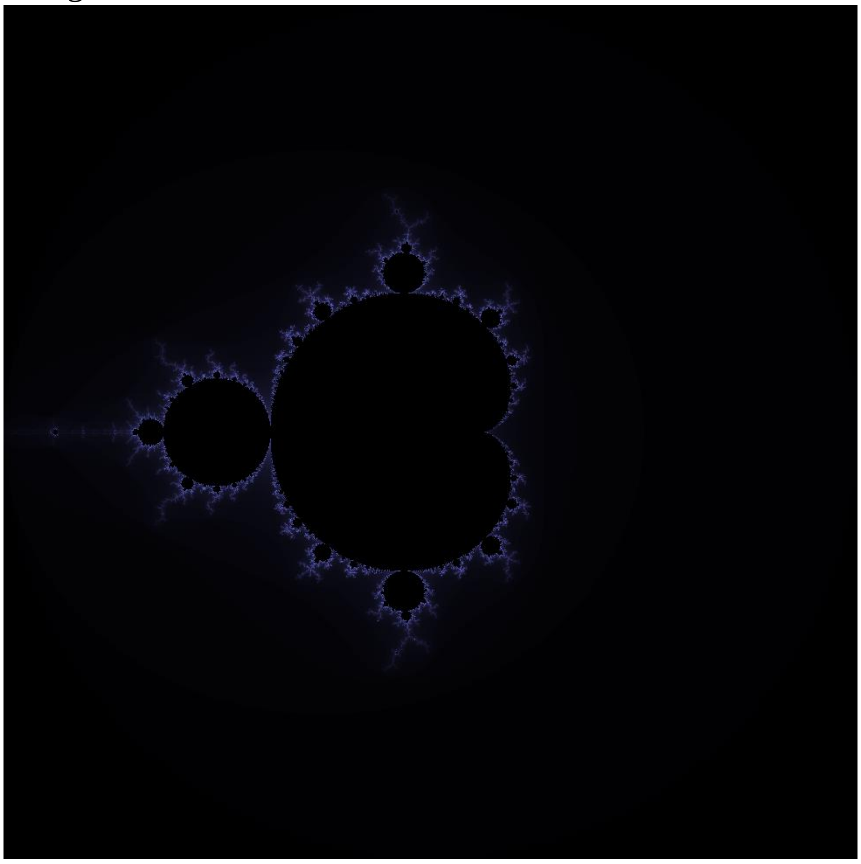


Figure 1.8: Mandelbrot Set Image

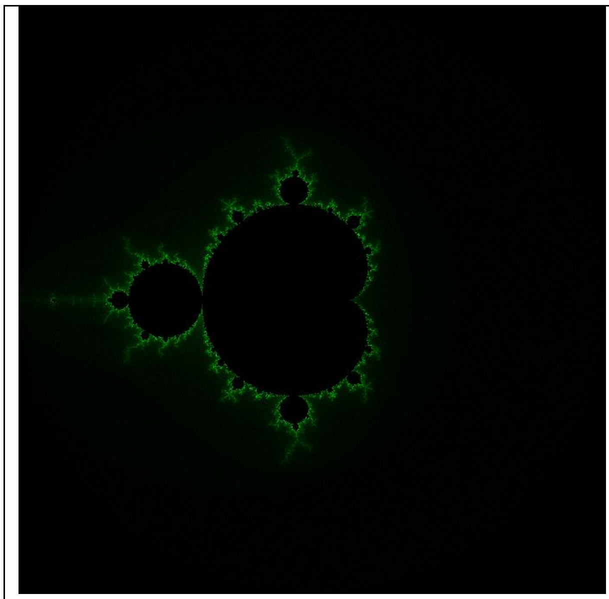
Using the code provided in the lab sheet, the program produces the images shown in Figure 8.

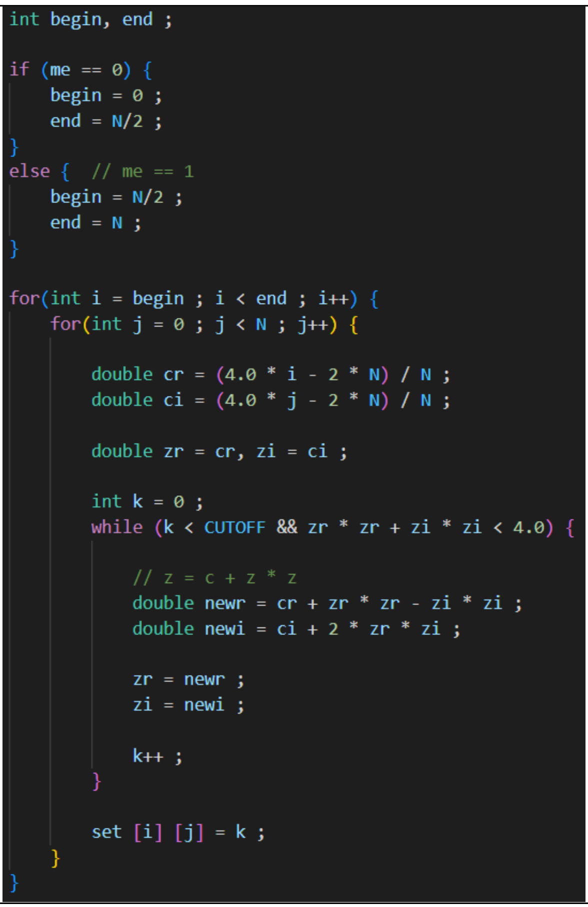
Runs	Time Taken (milliseconds / ms)
1	615
2	760
3	613
4	763
5	759
6	753
7	612
8	758
9	749
10	763

Figure 1.9: Sequential Mandelbrot Benchmark Results

1.2.2 Calculating the Mandelbrot Set through Parallelism

The image in Figure 1.10 below was calculated through parallelism using the code that was given in the lab worksheet.





```
int begin, end ;

if (me == 0) {
    begin = 0 ;
    end = N/2 ;
}
else {  // me == 1
    begin = N/2 ;
    end = N ;
}

for(int i = begin ; i < end ; i++) {
    for(int j = 0 ; j < N ; j++) {

        double cr = (4.0 * i - 2 * N) / N ;
        double ci = (4.0 * j - 2 * N) / N ;

        double zr = cr, zi = ci ;

        int k = 0 ;
        while (k < CUTOFF && zr * zr + zi * zi < 4.0) {

            // z = c + z * z
            double newr = cr + zr * zr - zi * zi ;
            double newi = ci + 2 * zr * zi ;

            zr = newr ;
            zi = newi ;

            k++ ;
        }

        set [i] [j] = k ;
    }
}
```

Figure 1.10: Mandelbrot Set Image, Inserted run() code

Runs	Time Taken (milliseconds / ms)
1	586
2	505
3	503
4	514
5	487
6	482
7	477
8	482
9	472
10	482

Figure 1.11: Parallel Mandelbrot Benchmark Results

1.2.3 Exercises

1.2.3.1 Exercise 1 – Division of i Range

Cores	Time Taken (milliseconds / ms)
Single-Core	612
Dual-Core	472

Figure 1.12: Time Taken for Single and Dual-Core (i Range)

Using Figure 1.12, which takes the best benchmark results from Figure 1.8 and Figure 1.9 and by adapting the parallel speedup formula from Figure 1.3; the calculated parallel speedup is 1.29661017.

1.2.3.2 Exercise 2 – Division of j Range

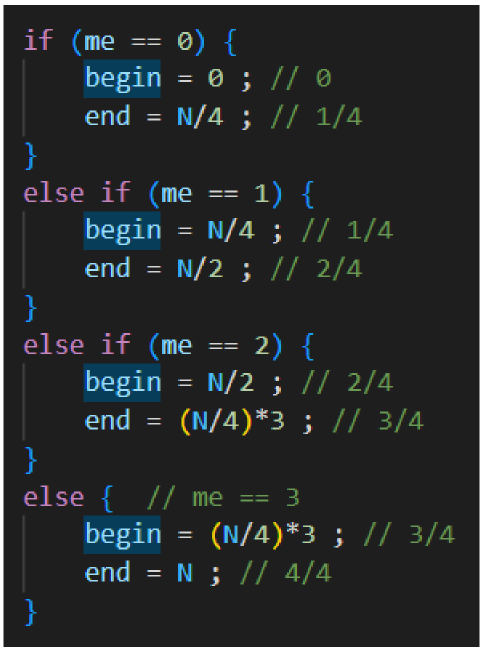
Cores	Time Taken (milliseconds / ms)
Single-Core	612
Dual-Core	442

Figure 1.13: Time Taken for Single and Dual-Core (j Range)

Using Figure 1.13 and by adapting the parallel speedup formula from Figure 1.3; the calculated parallel speedup is 1.38461538. The reason for the faster time when dividing the j range for dual-core threading is due to split in shape. When the split is done horizontally, the load balancing is equal whereas if dividing the i range which splits it vertically, the load balance becomes unequal.

1.2.3.3 Exercise 3 – Quad-Core Threading of the Mandelbrot Set

In this exercise, the Mandelbrot set program will run using quad-core threading and will be compared the dual-core threading result from the previous two exercises.



```
if (me == 0) {
    begin = 0 ; // 0
    end = N/4 ; // 1/4
}
else if (me == 1) {
    begin = N/4 ; // 1/4
    end = N/2 ; // 2/4
}
else if (me == 2) {
    begin = N/2 ; // 2/4
    end = (N/4)*3 ; // 3/4
}
else {  // me == 3
    begin = (N/4)*3 ; // 3/4
    end = N ; // 4/4
}
```

Figure 1.14: Quad-Core Threading

Cores	Time Taken (milliseconds / ms)	
Single-Core	749	
Dual-Core	Division of i Range	Division of j Range
	586	442
Quad-Core	104	

Figure 1.15: Parallel Time for the Mandelbrot Set

Cores	Parallel Speedup	
Dual-Core	Division of i Range	Division of j Range
	1.278157	1.69457014
Quad-Core	7.20192308	

Figure 1.16: Parallel Speedup for Dual-Core and Quad-Core Threading

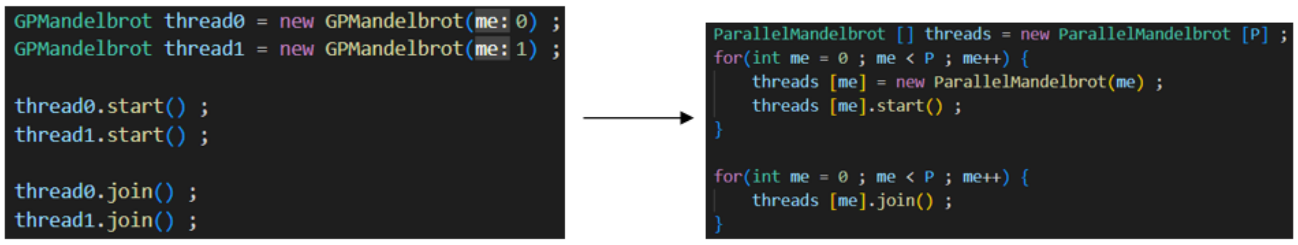
As shown in Figure 1.16: Parallel Speedup for Dual-Core and Quad-Core Threading, the quad-core threading is many times faster than the dual core threading of both i and j range division.

1.3 Week 3 – Workload Decompositions & a "Simulation"

This week's objective us to complete the work on the parallel Mandelbrot set from Week 2. This will be done by incorporating ideas on index space decomposition formats. This week will also introduce "simulation"-type program, Conway's Game of Life.

1.3.1 Generalising the Parallel Mandelbrot Set

Below shows the changes made to the code given in the lab worksheet.



```
GPMandelbrot thread0 = new GPMandelbrot(me: 0) ;
GPMandelbrot thread1 = new GPMandelbrot(me: 1) ;

thread0.start() ;
thread1.start() ;

thread0.join() ;
thread1.join() ;
```

```
ParallelMandelbrot [] threads = new ParallelMandelbrot [P] ;
for(int me = 0 ; me < P ; me++) {
    threads [me] = new ParallelMandelbrot(me) ;
    threads [me].start() ;
}

for(int me = 0 ; me < P ; me++) {
    threads [me].join() ;
}
```

Figure 1.17: Code Changes

For the new way to initialise the threads, two loops over *me* are used this may be due to how the loops are utilised. In the above example, the first loop create and starts all the threads and the second loop then joins them to together which may result in a quicker time whereas if there was only one loop then through each loop it would have start a thread and join them together immediately after.

Threads	Time Taken (milliseconds / ms)
1	614
2	307
4	165
8	103

Figure 1.18: Time Taken for Generalised Parallelisation

Threads	Parallel Speedup
2	2
4	3.7212121…
8	5.96116505

Figure 1.19: Parallel Speedup for the Generalised Parallelisation Times

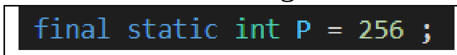
Both Figure 1.18 and Figure 1.19 above show the time taken for when each program was run depending on the number threads it utilises and the parallel speedup for each program execution. As seen in the parallel speedup, the number is shown to be as expected with the exception of when the program is run with 8 threads; this is because the device that program was ran on has a hex-core processor instead of an octa-core processor therefore the maximum number of threads possible would be 6.

1.3.2 Game of Life Program – Sequential Experiments

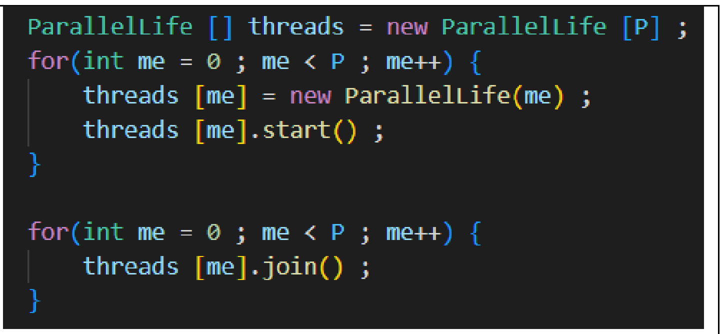
When experimenting with the board size in the sequential game of life program, given in the lab sheet, it showed the bigger the board size processing power was needed to display the game of life visuals.

1.3.3 Exercise

1.3.3.1 Parallelizing the Life Program



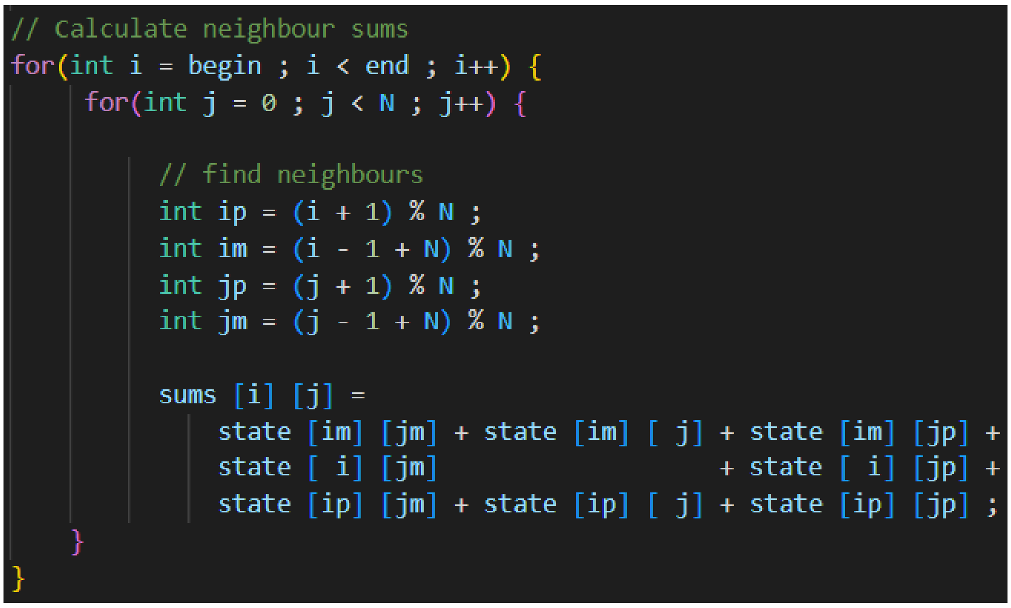
```java
final static int P = 256 ;
```



```java
ParallelLife [] threads = new ParallelLife [P] ;
for(int me = 0 ; me < P ; me++) {
    threads [me] = new ParallelLife(me) ;
    threads [me].start() ;
}

for(int me = 0 ; me < P ; me++) {
    threads [me].join() ;
}
```

Figure 1.20: Code for Thread Initialisation

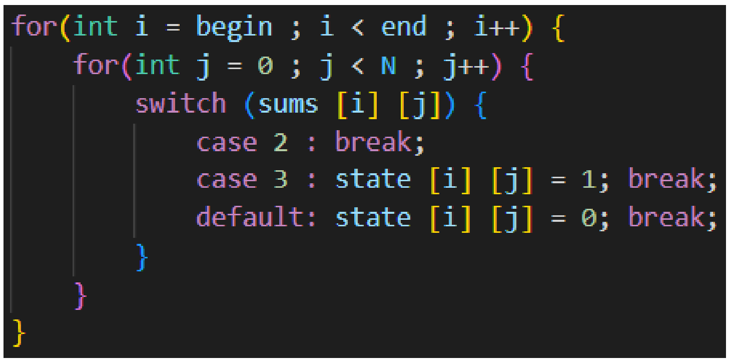
Figure 1.20 shows on the left, the variable that was created to store the number of threads that will be used when running the program. Whilst on the right show a new way to initialise the threads, which can be shown in Figure 1.17



```java
// Calculate neighbour sums
for(int i = begin ; i < end ; i++) {
    for(int j = 0 ; j < N ; j++) {

        // find neighbours
        int ip = (i + 1) % N ;
        int im = (i - 1 + N) % N ;
        int jp = (j + 1) % N ;
        int jm = (j - 1 + N) % N ;

        sums [i] [j] =
            state [im] [jm] + state [im] [ j] + state [im] [jp] +
            state [ i] [jm]                   + state [ i] [jp] +
            state [ip] [jm] + state [ip] [ j] + state [ip] [jp] ;
    }
}
```

Figure 1.21: FOR Loop for neighbours

Figure 1.2121 show the code for calculating the neighbours for each cell in the program.



```java
for(int i = begin ; i < end ; i++) {
    for(int j = 0 ; j < N ; j++) {
        switch (sums [i] [j]) {
            case 2 : break;
            case 3 : state [i] [j] = 1; break;
            default: state [i] [j] = 0; break;
        }
    }
}
```

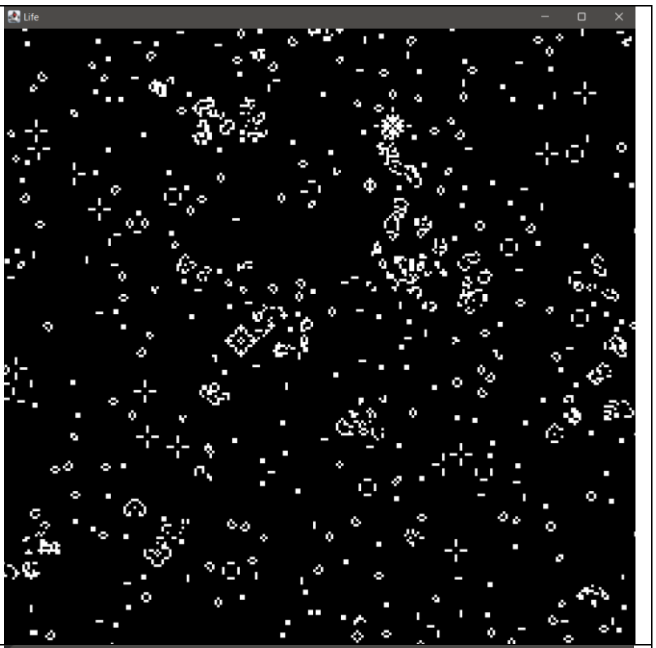
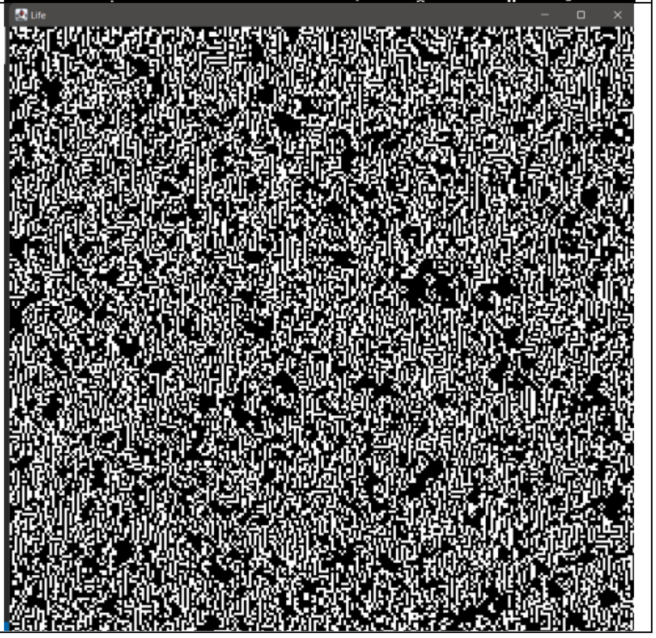
Figure 1.22: Updating Board Value State

Figure 1.22 shows the code for updating the state of the board value.

1.4 Week 4 – Parallel Programs with Interacting Threads

This week's objective is to learn and understand how to use barrier synchronisation across threads in Java.

1.4.1 Parallel Game of Life

Keeping both synch() in	 Life
Commenting both synch() out	 Life

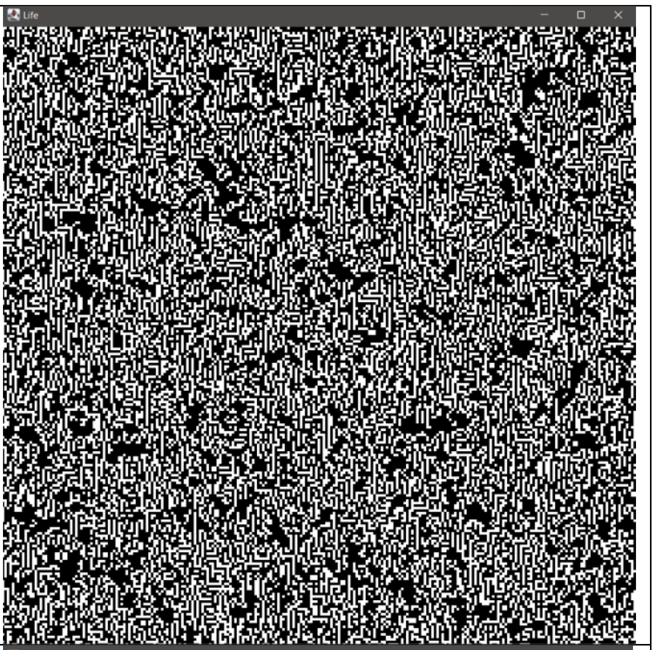
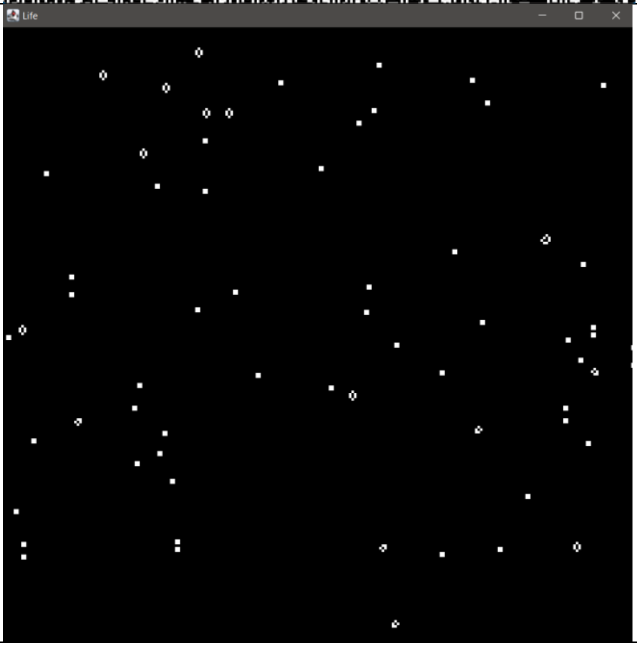
Commenting only the first synch() out	
Commenting only the second synch() out	

Figure 1.23: How synch() Affects the Program

As shown in Figure 1.23, it is essential to the program to have both synch(). When both synch() are commented out or when the first synch() is commented out then program will work but in a more packed manner. When the second synch() is commented out the program works but in a more spread out manner.

1.4.2 Solving the Laplace Equation

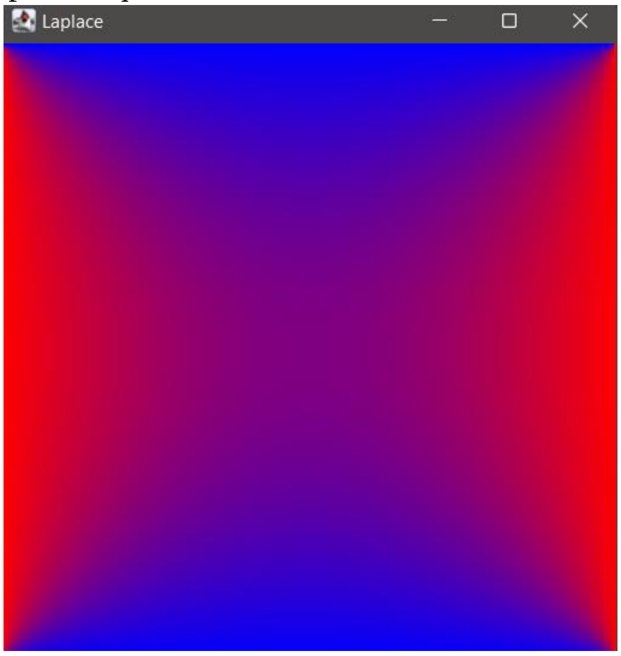


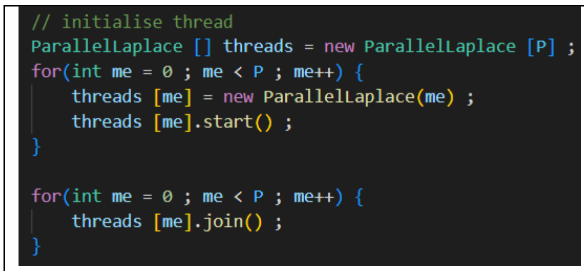
Figure 1.24: End Result of Running the Laplace Program

Figure 1.24 was achieved using the code provided in the lab sheet.

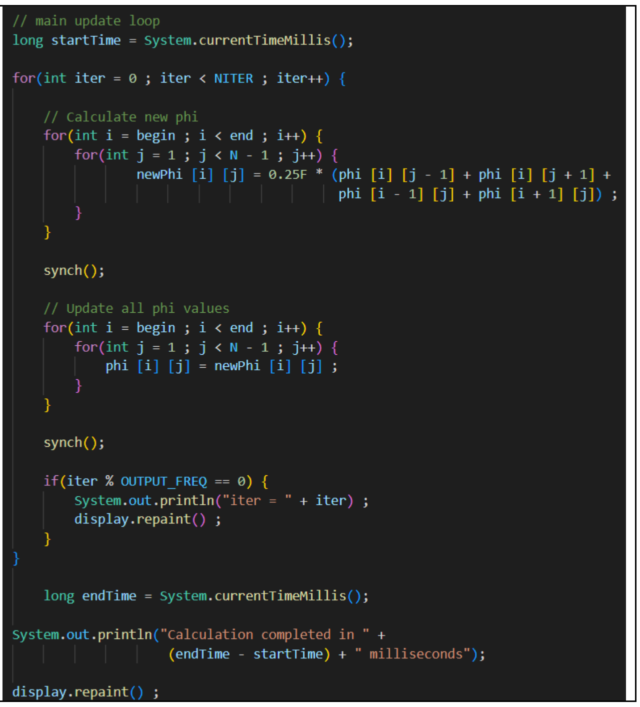
1.4.3 Exercises

1.4.3.1 Exercise 1 – Parallelizing the Laplace Equation

The Laplace program that will be used for the following exercises was created by imitating the parallel game of life program used in Section 1.4.1.



```java
// initialise thread
ParallelLaplace [] threads = new ParallelLaplace [P] ;
for(int me = 0 ; me < P ; me++) {
    threads [me] = new ParallelLaplace(me) ;
    threads [me].start() ;
}

for(int me = 0 ; me < P ; me++) {
    threads [me].join() ;
}
```



```java
// main update loop
long startTime = System.currentTimeMillis();

for(int iter = 0 ; iter < NITER ; iter++) {

    // Calculate new phi
    for(int i = begin ; i < end ; i++) {
        for(int j = 1 ; j < N - 1 ; j++) {
            newPhi [i] [j] = 0.25F * (phi [i] [j - 1] + phi [i] [j + 1] +
                                      phi [i - 1] [j] + phi [i + 1] [j]) ;
        }
    }

    synch();

    // Update all phi values
    for(int i = begin ; i < end ; i++) {
        for(int j = 1 ; j < N - 1 ; j++) {
            phi [i] [j] = newPhi [i] [j] ;
        }
    }

    synch();

    if(iter % OUTPUT_FREQ == 0) {
        System.out.println("iter = " + iter) ;
        display.repaint() ;
    }
}

    long endTime = System.currentTimeMillis();

System.out.println("Calculation completed in " +
                        (endTime - startTime) + " milliseconds");

display.repaint() ;
```

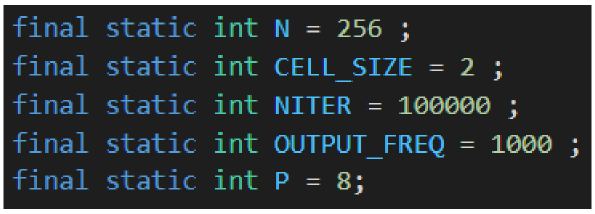
Figure 1.25: Code for Parallelized Laplace Equation

Figure 1.25 shows the code that was created to initialize the thread for parallelized code and how a new method, run(), was created by moving the main update loop from main() to run() and modifying it to fit the problem.

1.4.3.2 Exercise 2 – Benchmarking the Parallelized Laplace Equation

Threads	Time Taken (milliseconds /ms)	Parallel Speedup
1	6569	
2	4968	1.32226248
4	6090	1.07865353
8	10380	0.63285164

Figure 1.26: Benchmark Results



```
final static int N = 256 ;
final static int CELL_SIZE = 2 ;
final static int NITER = 100000 ;
final static int OUTPUT_FREQ = 1000 ;
final static int P = 8;
```

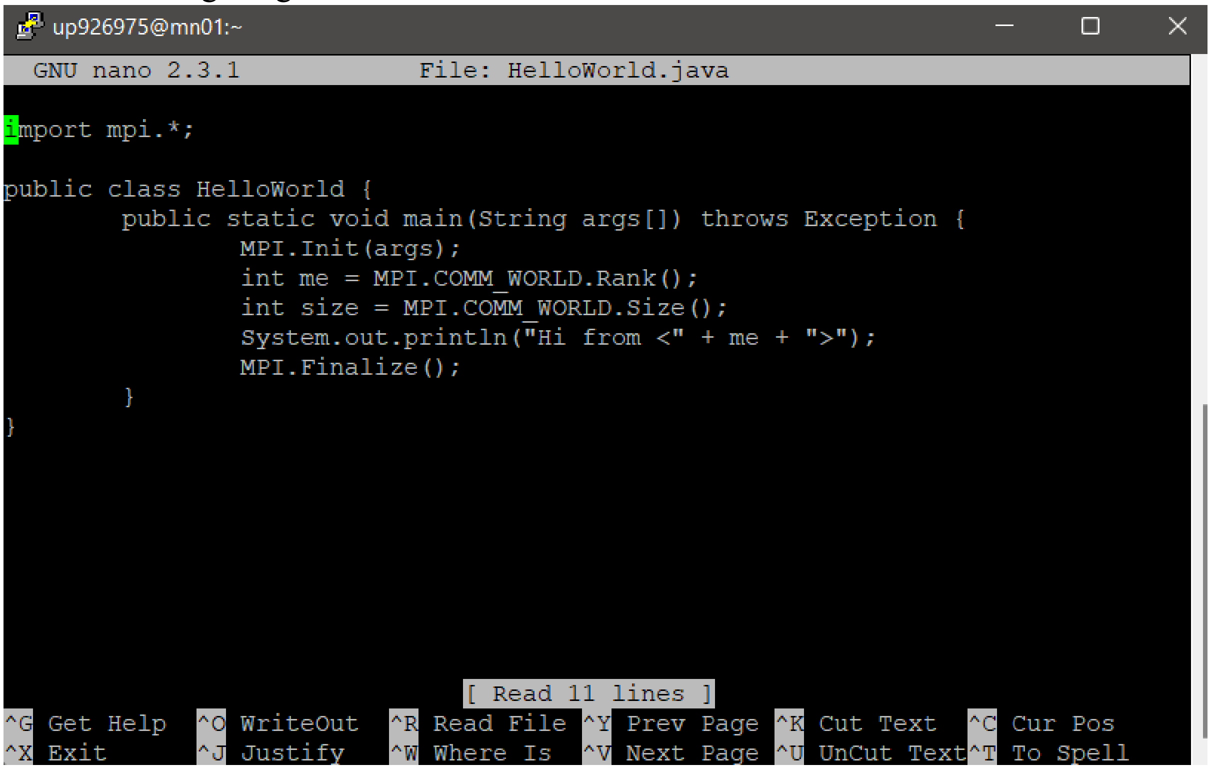
Figure 1.27: Variable Settings used for the Benchmark Results

As seen in Figure 1.26, there was a parallel speedup when using 2 threads and 4 threads however there was a parallel slowdown when using 8 threads. This may be due to the setting used for program, shown in Figure 1.27.

1.5 Week 5 – Running MPJ Programs

This week's lab objective is to use MPJ Express to gain experience of MPI-style parallel programming on small clusters of workstations. This week will also show the use of using several computers to work together on parts of a single problem

1.5.1 Running Programs in "Multicore" Mode

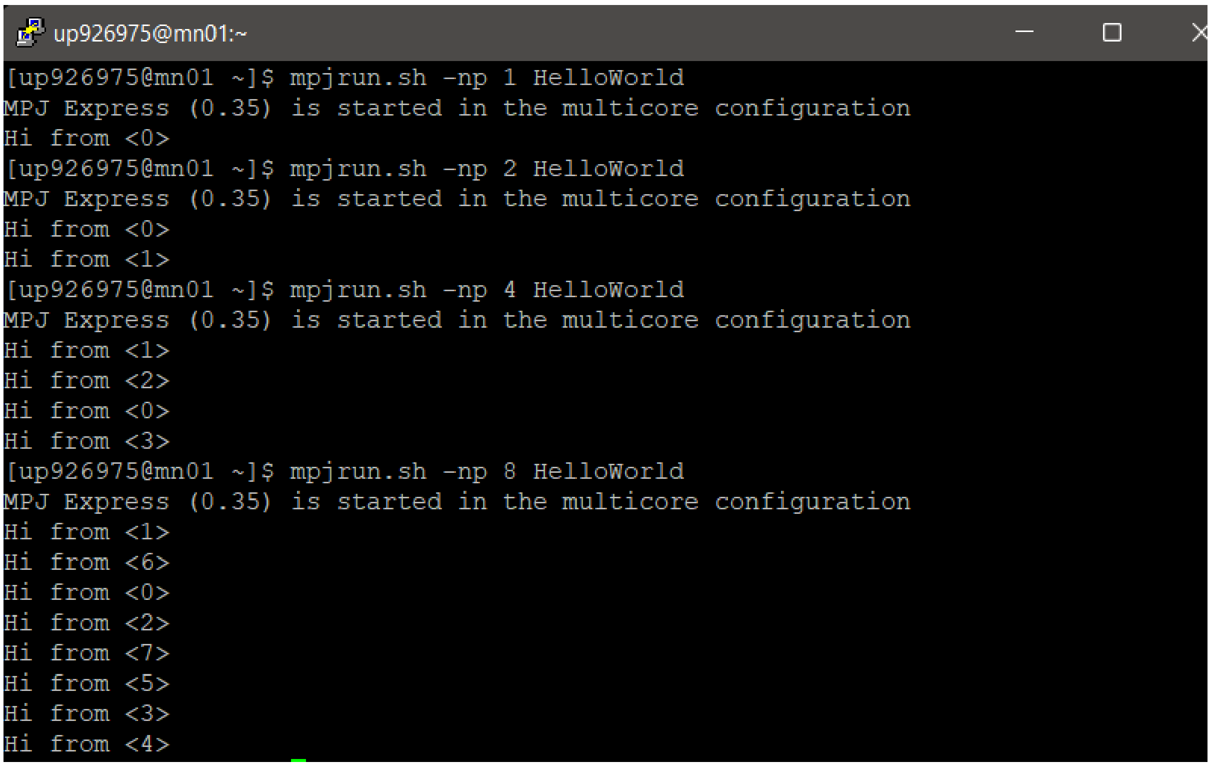


```
up926975@mn01:~
  GNU nano 2.3.1             File: HelloWorld.java

import mpi.*;

public class HelloWorld {
        public static void main(String args[]) throws Exception {
                MPI.Init(args);
                int me = MPI.COMM_WORLD.Rank();
                int size = MPI.COMM_WORLD.Size();
                System.out.println("Hi from <" + me + ">");
                MPI.Finalize();
        }
}

                               [ Read 11 lines ]
^G Get Help  ^O WriteOut  ^R Read File ^Y Prev Page ^K Cut Text  ^C Cur Pos
^X Exit      ^J Justify   ^W Where Is  ^V Next Page ^U UnCut Text^T To Spell
```

Figure 1.28: Creation of HelloWorld.java

Figure 1.28 shows the creation of the HelloWorld.java file which was created when inputting the command *nano HelloWorld.java* and inputting code from the lab sheet.

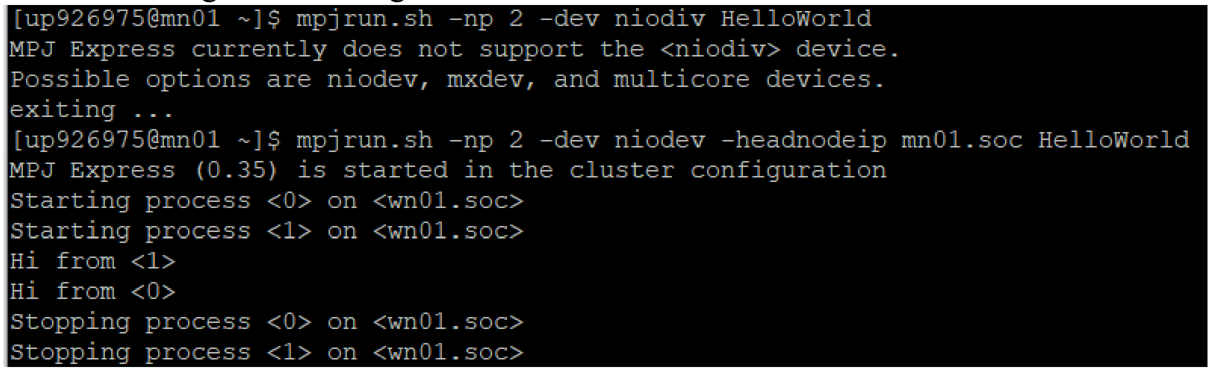
Using the HelloWorld.java file that was created, the program is then run in multicore mode using the mpjrun.sh command and the results for that are shown in Figure 1.29



```
[up926975@mn01 ~]$ mpjrun.sh -np 1 HelloWorld
MPJ Express (0.35) is started in the multicore configuration
Hi from <0>
[up926975@mn01 ~]$ mpjrun.sh -np 2 HelloWorld
MPJ Express (0.35) is started in the multicore configuration
Hi from <0>
Hi from <1>
[up926975@mn01 ~]$ mpjrun.sh -np 4 HelloWorld
MPJ Express (0.35) is started in the multicore configuration
Hi from <1>
Hi from <2>
Hi from <0>
Hi from <3>
[up926975@mn01 ~]$ mpjrun.sh -np 8 HelloWorld
MPJ Express (0.35) is started in the multicore configuration
Hi from <1>
Hi from <6>
Hi from <0>
Hi from <2>
Hi from <7>
Hi from <5>
Hi from <3>
Hi from <4>
```

Figure 1.29: Running HelloWorld.java in Multicore Mode using 1, 2, 4 & 8 Threads

1.5.2 Running an MPJ Program in “Cluster” Mode



```
[up926975@mn01 ~]$ mpjrun.sh -np 2 -dev niodiv HelloWorld
MPJ Express currently does not support the <niodiv> device.
Possible options are niodev, mxdev, and multicore devices.
exiting ...
[up926975@mn01 ~]$ mpjrun.sh -np 2 -dev niodev -headnodeip mn01.soc HelloWorld
MPJ Express (0.35) is started in the cluster configuration
Starting process <0> on <wn01.soc>
Starting process <1> on <wn01.soc>
Hi from <1>
Hi from <0>
Stopping process <0> on <wn01.soc>
Stopping process <1> on <wn01.soc>
```

Figure 1.30: MPJ Program in “Cluster” Mode

In the figure above, it is shown that the first command does not work, this is due to there being no default node being set hence why in the second command the head node must be specified, in this case it is mn01.soc, which will be predominantly used throughout the rest of the lab sheets.

1.5.3 Distributed Calculation of Pi

Number of Threads	Time Taken (milliseconds /ms)
1	665
2	553
4	502
8	170

Figure 1.31: Time Taken for Calculating Pi in “Cluster” Mode

Number of Threads	Parallel Speedup
2	1.20253165
4	1.3247012
8	3.91176471

Figure 1.32: Parallel Speedup for Calculating Pi in Cluster Mode

The figures above show that in cluster mode, running Pi program using multiple threads produces a parallel speedup. When running the program in cluster mode, two worker nodes were used for the results which were wn01.soc and wn03.soc.

1.5.4 Exercises

1.5.4.1 Parallel Speedup in "Multicore" Mode

This exercise was to show the benchmark result that are recorded when the program for calculating pi is ran in multicore mode.

Number of Threads	Time Taken (milliseconds /ms)	Parallel Speedup
1	665	
2	368	1.8070652
4	206	3.22815534
8	130	5.11538462
12	90	7.38888889

Figure 1.33: Benchmark Results for Calculation of Pi in Multicore Mode

1.5.4.2 Parallel Speedup in "Cluster" Mode

For the time taken and parallel speedup for single thread, dual thread, quad thread and octa thread refer to Figure 1.3131 and Figure 1.3232

Number of Threads	Time Taken (milliseconds /ms)	Parallel Speedup
12	151	4.40397351

Figure 1.34: Benchmark Results for Calculation of Pi using 12 Threads

1.6 Week 6 – MPJ Communication

The objective for this week's lab work focus on one particular complex distributed-memory parallel program, an MPJ parallel version of the Laplace equation solver.

1.6.1 A Distributed Memory Solver

Using the code given from the lab, a MPJLalplace.java file was created which will be ran on MobaXterm instead of PuTTY.

Threads	Time Taken (milliseconds / ms)	Parallel Speedup
1	12674	
2	55441	0.22860338
4	59730	0.21218818

Figure 1.35: Benchmark Results of MPJLaplace.java ran at the University

Threads	Time Taken (milliseconds /ms)	Parallel Speedup
1	12674	
2	8376	1.51313276
4	9001	1.40806577
8	13962	0.90774961

Figure 1.36: Benchmark Results of MPJLaplace.java ran at Home

Both figures above show the benchmark results of running the MPJ Laplace program both at the university and my place of residence. The benchmark results at the university showed a very lard parallel slowdown time whereas the results at my place of residence shows a significant increase parallel speedup increase. This may be due to connectivity issues with the cluster servers, or it may be due to large number of users using the cluster nodes at the same time during the test.

Threads	Time Taken (milliseconds / ms)	Parallel Speedup
1	10140	
2	10216	0.9896545
4	9581	1.05834464
6	9308	1.08938547

Figure 1.37: Benchmark Results of MPJLaplace.java without the Creation of Graphics

1.6.2 Running the MPJLalplace.java Across Multiple Nodes

In the previous test, the MPJLaplace.java program was only run through a single node, wn01.soc. In these tests the program will be ran through 2 nodes, wn01.soc and wn03.soc.

The program was tested when the graphic was formed and when it wasn't formed, although it is shown that a parallel speedup was not produced, the program running without the graphics produced a slightly faster time.

Threads	Time Taken (milliseconds / ms)	Parallel Speedup
1	9134	
2	42634	0.21424215
4	39108	0.23355835
8	37419	0.24410059

Figure 1.38: Benchmark Results of MPJLaplace.java with the Creation of Graphics

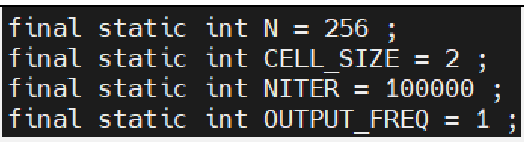
Threads	Time Taken (milliseconds / ms)	Parallel Speedup
1	11490	
2	43091	0.26664501

4	37662	0.30508205
8	35755	0.32135366

Figure 1.39: Benchmark Results of MPJLaplace.java without the Creation of Graphics

1.6.3 Replace of the Edge Swap Code

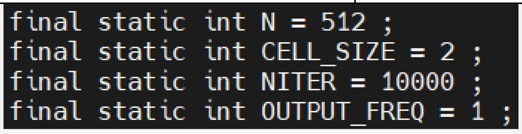
Test 1 Variables:



```
final static int N = 256 ;
final static int CELL_SIZE = 2 ;
final static int NITER = 100000 ;
final static int OUTPUT_FREQ = 1 ;
```

Threads	Time Taken (milliseconds / ms)	Parallel Speedup
1	8469	
2	51074	0.16581822
4	41674	0.20322023
8	39227	0.21589721
12	37803	0.22402984
16	36443	0.23239031

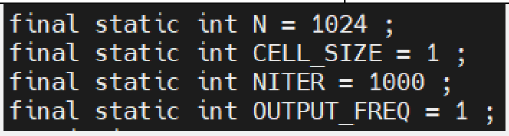
Test 2 Variables:



```
final static int N = 512 ;
final static int CELL_SIZE = 2 ;
final static int NITER = 10000 ;
final static int OUTPUT_FREQ = 1 ;
```

Threads	Time Taken (milliseconds / ms)	Parallel Speedup
1	5005	
2	93707	0.05341116
4	100585	0.04975891
8	119748	0.04179611
12	68012	0.07358995
16	68336	0.07324104

Test 3 Variables:



```
final static int N = 1024 ;
final static int CELL_SIZE = 1 ;
final static int NITER = 1000 ;
final static int OUTPUT_FREQ = 1 ;
```

Threads	Time Taken (milliseconds / ms)	Parallel Speedup
1	3590	
2	35295	0.10171412
4	33596	0.10685796
8	36568	0.09817327
12	39280	0.09139511
16	42842	0.08379627

Figure 1.40: Benchmark Results for Edge Swap Code Replacement

The benchmark results in the Figure 1.40 show the results of what happens when the edge swap code was replaced with what was given in the lab sheet. The results above show that there was not a lot of difference from before the change in Figure 1.38 and Figure 1.39. As the shows that there was still a significant parallel slowdown.

1.7 Week 7 – An MPJ Task Farm

This week's objective is to understand how MPJ task farms work.

1.7.1 A Slow Mandelbrot

Threads	Time Taken (milliseconds /ms)	Parallel Speedup
1	28418	
2	28420	0.99992963
4	28289	1.00456008
8	28282	1.00480871

Figure 1.41: Benchmark Results of Slow Mandelbrot in Multicore Mode

In Figure 1.41, a parallel speedup is shown when using 4 and 8 threads, however it is a very small difference, it is also the same with using 2 threads, although a parallel slowdown is calculated it only by 2 seconds which can be negligible to the overall process.

Threads	Time Taken (milliseconds / ms)	Parallel Speedup
1	26572	
2	26582	0.99962381
4	27203	0.97680403
8	27815	0.95531188

Figure 1.42: Benchmark Results of Slow Mandelbrot in Cluster Mode

However, Figure 1.42 shows that there was a parallel slowdown on all tests that were conducted. This again may be due to connectivity issues or overhead issues.

1.7.2 An MPJ Task Farm – Slow Mandelbrot

Threads	Time Taken (milliseconds / ms)	Parallel Speedup
1	28547	
2	14418	1.97995561
4	7389	3.86344566
8	3914	7.29356157
12	2848	10.0235253

Figure 1.43: Benchmark Results in Multicore Mode using an MPJ Task Farm

Threads	Time Taken (milliseconds / ms)	Parallel Speedup
1	26641	
2	13634	1.95401203
4	7223	3.68835664
8	3912	6.81007157
12	2730	9.7586081

Figure 1.44: Benchmark Results in Cluster Mode using an MPJ Task Farm

2 Mini Project

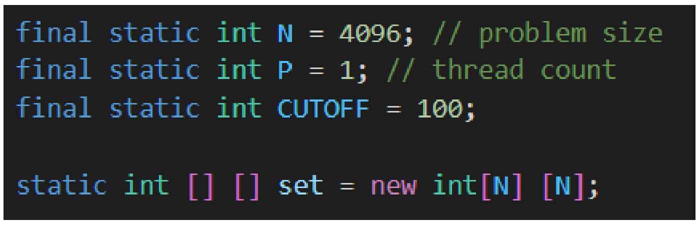
The mini project that will be undertaken is a ray tracing program that was provided by the lecturer.

2.1 Approach 1 – Generalised Parallel Programming

The first approach that was undertaken was by using the method of generalised parallelisation.

2.1.1 Code Modifications/Additions

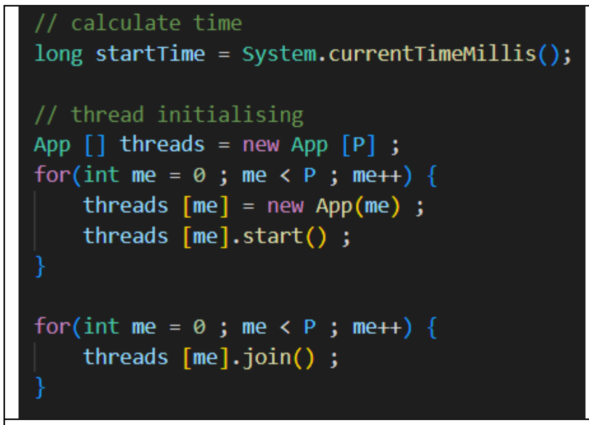
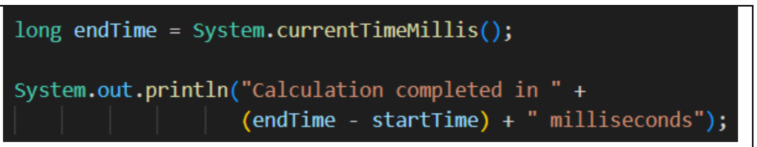
For the code, the program used in Generalising the Parallel Mandelbrot Set was used as foundation to better understand how to generalise the problem and then addition to code were made based on the project.



```
final static int N = 4096; // problem size
final static int P = 1; // thread count
final static int CUTOFF = 100;

static int [] [] set = new int[N] [N];
```

Figure 2.1: Global Variables

The variables above were crucial to allow the program to work in the concept of generalized parallel programming. The variable 'N' would represent the size of the problem and 'P' would represent the number of threads that would be used for the program.

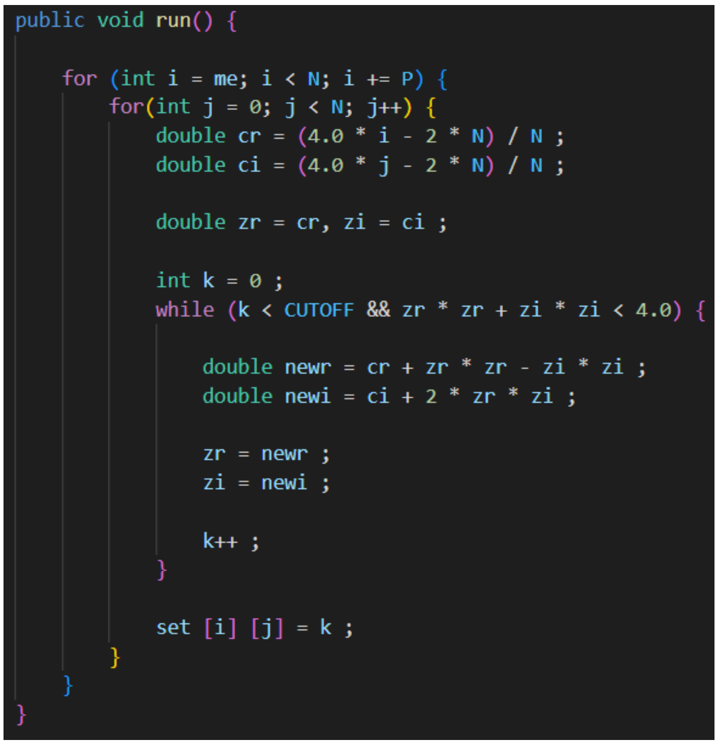
	
Before Scene Setup	After Scene Setup

```
// calculate time
long startTime = System.currentTimeMillis();

// thread initialising
App [] threads = new App [P] ;
for(int me = 0 ; me < P ; me++) {
    threads [me] = new App(me) ;
    threads [me].start() ;
}

for(int me = 0 ; me < P ; me++) {
    threads [me].join() ;
}
```

```
long endTime = System.currentTimeMillis();

System.out.println("Calculation completed in " +
                    (endTime - startTime) + " milliseconds");
```

Figure 2.2: Code Added in main() Method

Within the main() method, before setting up the scene, the code on the left is to initialize the thread and record the start time for benchmark results. The code on the right is written after setting up the scene in main() and is used to record end time output the time taken for the program to finish in milliseconds.



```
public void run() {

    for (int i = me; i < N; i += P) {
        for(int j = 0; j < N; j++) {
            double cr = (4.0 * i - 2 * N) / N ;
            double ci = (4.0 * j - 2 * N) / N ;

            double zr = cr, zi = ci ;

            int k = 0 ;
            while (k < CUTOFF && zr * zr + zi * zi < 4.0) {

                double newr = cr + zr * zr - zi * zi ;
                double newi = ci + 2 * zr * zi ;

                zr = newr ;
                zi = newi ;

                k++ ;
            }

            set [i] [j] = k ;
        }
    }
}
```

Figure 2.3: run() Method using Cyclic Decomposition

The above code is the run() method which was added to the given code. This will allow for code to run a generalized manner using the concept of cyclic decomposition.

2.1.2 Benchmark Results

Thread Count	Time Taken (ms)	Parallel Speedup
Problem Size = 4096		
1	815	
2	541	1.5064695
4	378	2.15608466
8	330	2.46969697
12	359	2.27019499
16	334	2.44011976
Problem Size = 2048		
1	379	
2	334	1.13473054
4	285	1.32982456
8	293	1.29351536
12	304	1.24671053
16	279	1.35842294
Problem Size = 1024		
1	297	
2	261	1.13793103
4	270	1.1
8	256	1.16015625
12	255	1.16470588
16	256	1.16015625

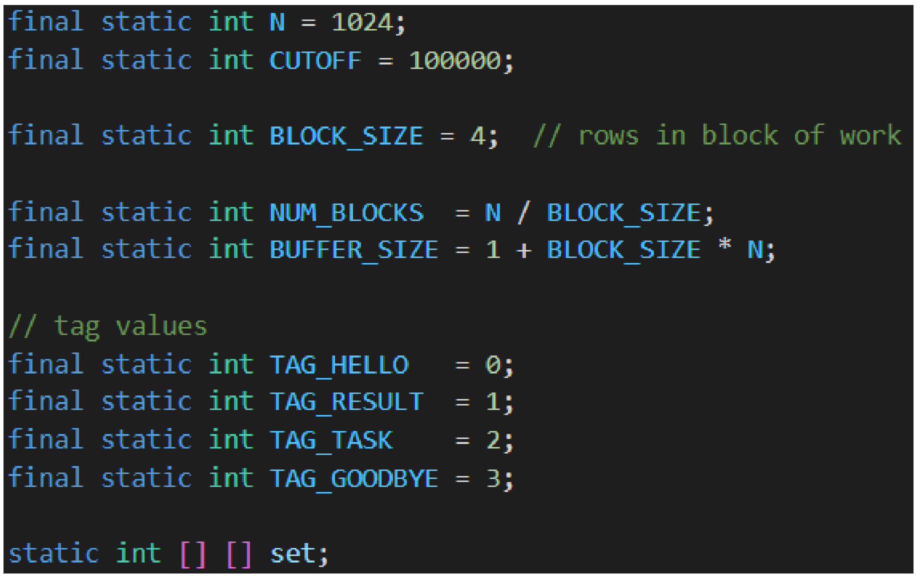
Figure 2.4: Benchmark Results

The results in Figure 2.4 show how with varying problem sizes and varying thread counts the parallel speedup changes. The usual assumption is the as the problem size decreases the speed at which program finishes is faster. The results above prove that however it also shows that with a lower problem size, the time taken for the program to finish is not that much faster as the threads increases. This shows that even though a faster time is shown with a lower problem size, the efficiency during parallelising decreases. Therefore, the best way to run the program would be by finding a balance between the problem size and the number of threads.

2.2 Approach 2 – MPJ Task Farm

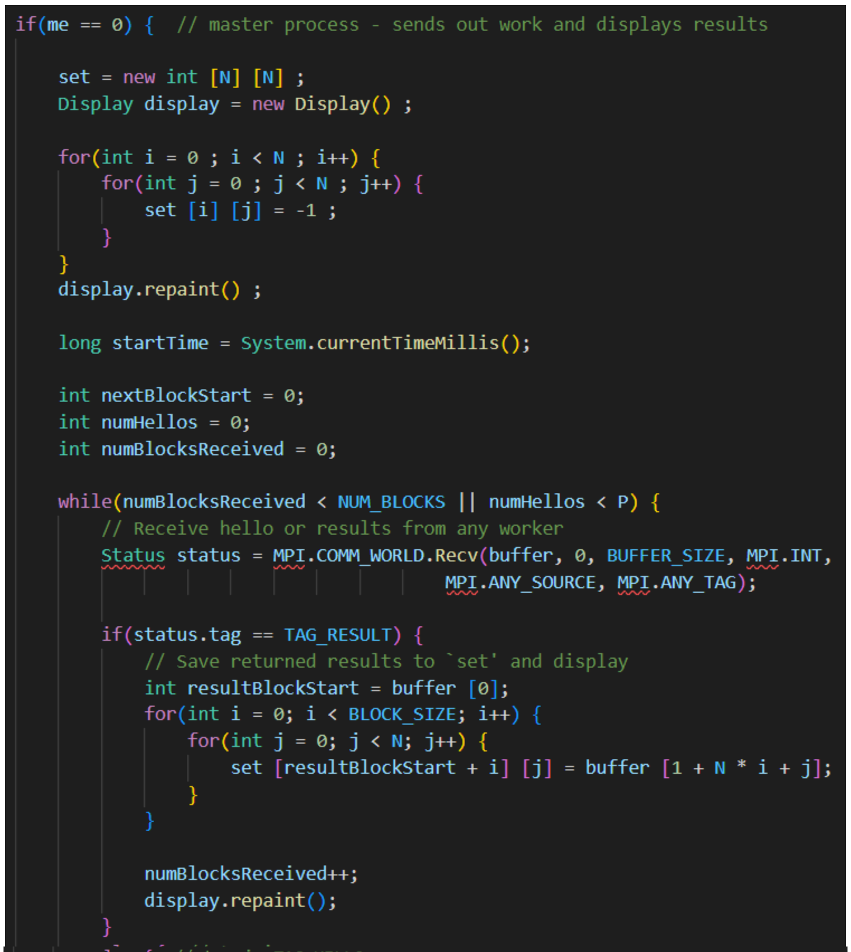
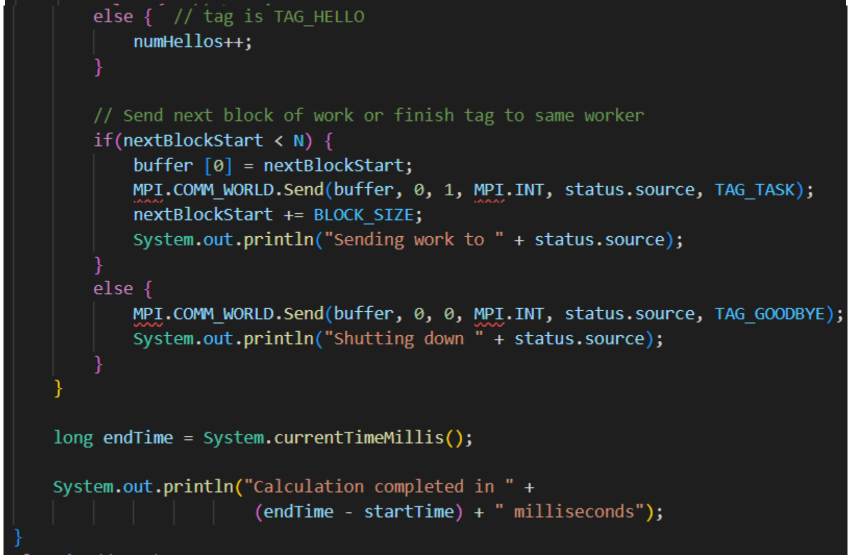
The plan for this approach is to use and modify the concept of MPJ Task Farm from Week 7.

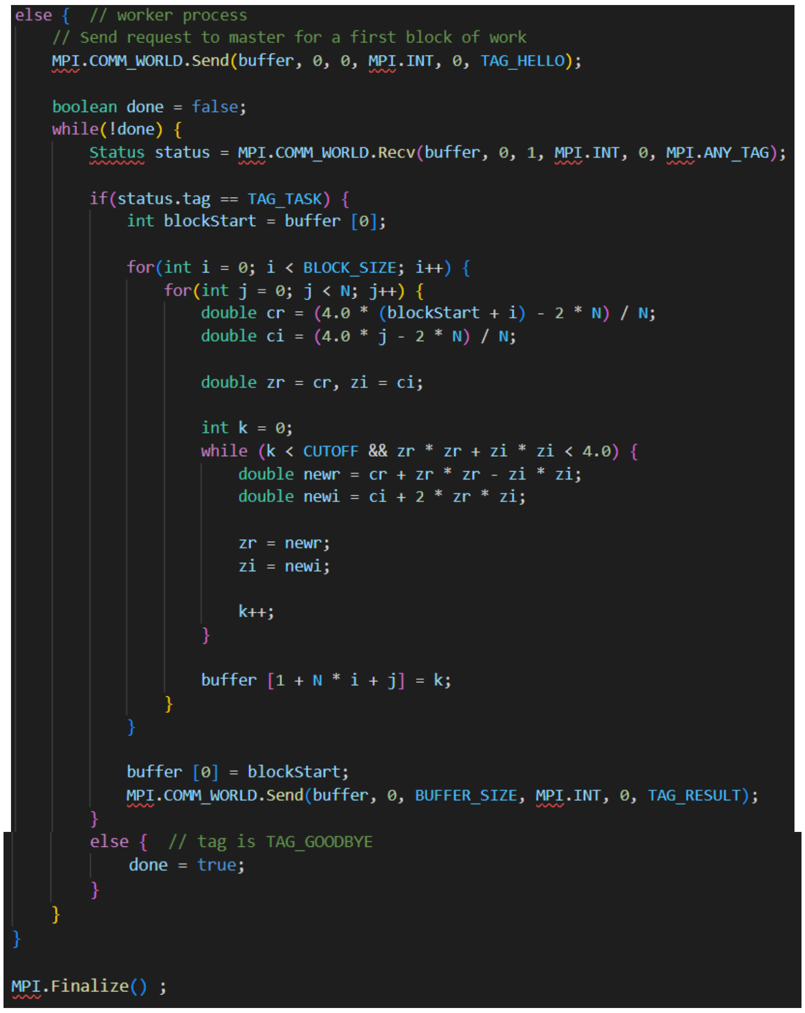
2.2.1 Code Modifications



```
final static int N = 1024;
final static int CUTOFF = 100000;

final static int BLOCK_SIZE = 4;  // rows in block of work

final static int NUM_BLOCKS  = N / BLOCK_SIZE;
final static int BUFFER_SIZE = 1 + BLOCK_SIZE * N;

// tag values
final static int TAG_HELLO   = 0;
final static int TAG_RESULT  = 1;
final static int TAG_TASK    = 2;
final static int TAG_GOODBYE = 3;

static int [] [] set;
```

Figure 2.5: New Global Variables



```java
if(me == 0) {  // master process - sends out work and displays results

    set = new int [N] [N] ;
    Display display = new Display() ;

    for(int i = 0 ; i < N ; i++) {
        for(int j = 0 ; j < N ; j++) {
            set [i] [j] = -1 ;
        }
    }
    display.repaint() ;

    long startTime = System.currentTimeMillis();

    int nextBlockStart = 0;
    int numHellos = 0;
    int numBlocksReceived = 0;

    while(numBlocksReceived < NUM_BLOCKS || numHellos < P) {
        // Receive hello or results from any worker
        Status status = MPI.COMM_WORLD.Recv(buffer, 0, BUFFER_SIZE, MPI.INT,
                                            MPI.ANY_SOURCE, MPI.ANY_TAG);

        if(status.tag == TAG_RESULT) {
            // Save returned results to `set' and display
            int resultBlockStart = buffer [0];
            for(int i = 0; i < BLOCK_SIZE; i++) {
                for(int j = 0; j < N; j++) {
                    set [resultBlockStart + i] [j] = buffer [1 + N * i + j];
                }
            }

            numBlocksReceived++;
            display.repaint();
        }
        else {  // tag is TAG_HELLO
            numHellos++;
        }

        // Send next block of work or finish tag to same worker
        if(nextBlockStart < N) {
            buffer [0] = nextBlockStart;
            MPI.COMM_WORLD.Send(buffer, 0, 1, MPI.INT, status.source, TAG_TASK);
            nextBlockStart += BLOCK_SIZE;
            System.out.println("Sending work to " + status.source);
        }
        else {
            MPI.COMM_WORLD.Send(buffer, 0, 0, MPI.INT, status.source, TAG_GOODBYE);
            System.out.println("Shutting down " + status.source);
        }
    }

    long endTime = System.currentTimeMillis();

    System.out.println("Calculation completed in " +
                       (endTime - startTime) + " milliseconds");
}
```



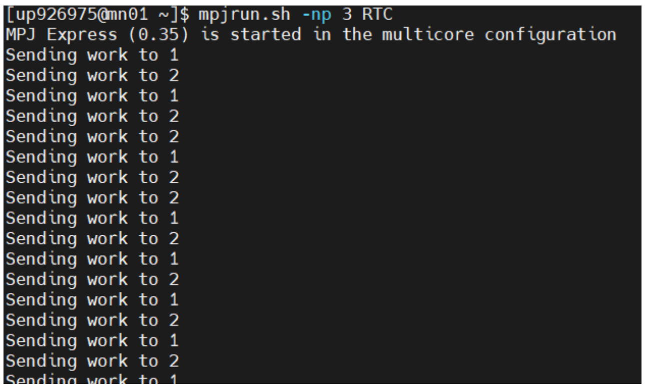
```
else {  // worker process
    // Send request to master for a first block of work
    MPI.COMM_WORLD.Send(buffer, 0, 0, MPI.INT, 0, TAG_HELLO);

    boolean done = false;
    while(!done) {
        Status status = MPI.COMM_WORLD.Recv(buffer, 0, 1, MPI.INT, 0, MPI.ANY_TAG);

        if(status.tag == TAG_TASK) {
            int blockStart = buffer [0];

            for(int i = 0; i < BLOCK_SIZE; i++) {
                for(int j = 0; j < N; j++) {
                    double cr = (4.0 * (blockStart + i) - 2 * N) / N;
                    double ci = (4.0 * j - 2 * N) / N;

                    double zr = cr, zi = ci;

                    int k = 0;
                    while (k < CUTOFF && zr * zr + zi * zi < 4.0) {
                        double newr = cr + zr * zr - zi * zi;
                        double newi = ci + 2 * zr * zi;

                        zr = newr;
                        zi = newi;

                        k++;
                    }

                    buffer [1 + N * i + j] = k;
                }
            }

            buffer [0] = blockStart;
            MPI.COMM_WORLD.Send(buffer, 0, BUFFER_SIZE, MPI.INT, 0, TAG_RESULT);
        }
        else {  // tag is TAG_GOODBYE
            done = true;
        }
    }
}

MPI.Finalize() ;
```

Figure 2.6: Code Additions in main() Method

The code above show the new code that was added to main() method which was based off the MPJ task farm code from 1.7 Week 7.

2.2.2 Issues

Using the code above, several issues arise from running it in both multicore mode and cluster mode. When running in multicore mode, the program would open up multiple displays as shown in below.



```
[up926975@mn01 ~]$ mpjrun.sh -np 3 RTC
MPJ Express (0.35) is started in the multicore configuration
Sending work to 1
Sending work to 2
Sending work to 1
Sending work to 2
Sending work to 2
Sending work to 1
Sending work to 2
Sending work to 2
Sending work to 1
Sending work to 2
Sending work to 1
Sending work to 2
Sending work to 1
Sending work to 2
Sending work to 1
Sending work to 2
```

Figure 2.7: Multicore Test

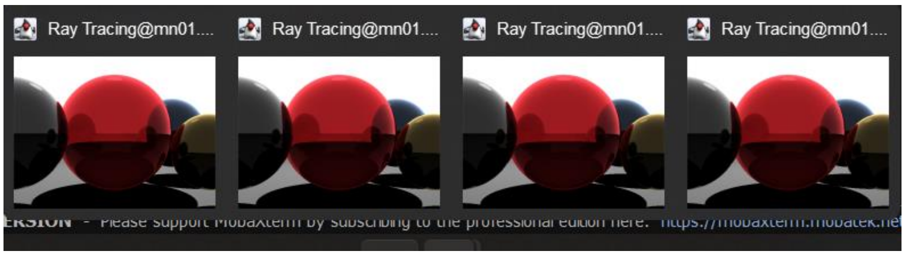
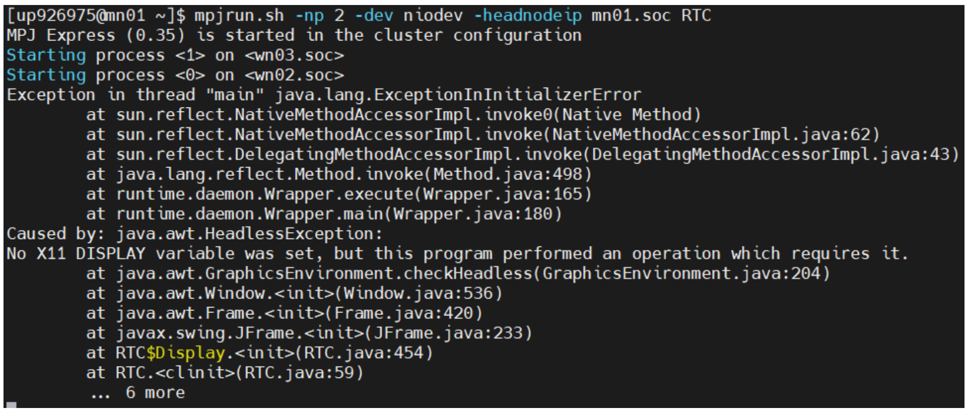


Figure 2.8: Multicore Test Images

Along with this, the benchmark results do not print when the visuals for the code finish.

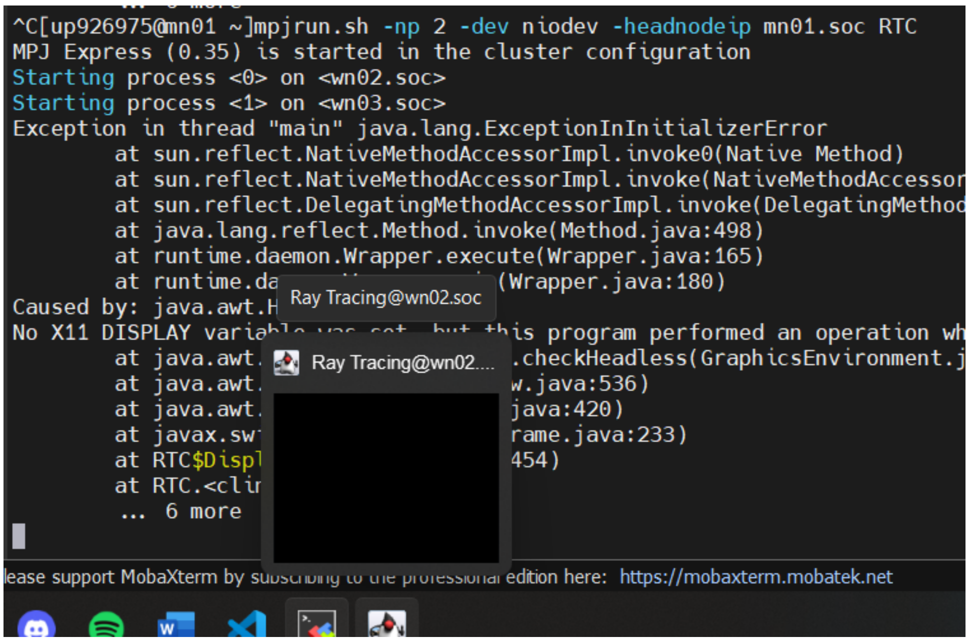
When trying to run on cluster mode, multiple issues occur. Running the code using a single thread would cause to images to be produced however not run time would be printed to the user. Running the code using two or more threads causes the issues below to be shown. One informing that there is an issue in the man() method regarding threading whilst the other issue is regarding about display issues.



```
[up926975@mn01 ~]$ mpjrun.sh -np 2 -dev niodev -headnodeip mn01.soc RTC
MPJ Express (0.35) is started in the cluster configuration
Starting process <1> on <wn03.soc>
Starting process <0> on <wn02.soc>
Exception in thread "main" java.lang.ExceptionInInitializerError
        at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
        at sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:62)
        at sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:43)
        at java.lang.reflect.Method.invoke(Method.java:498)
        at runtime.daemon.Wrapper.execute(Wrapper.java:165)
        at runtime.daemon.Wrapper.main(Wrapper.java:180)
Caused by: java.awt.HeadlessException:
No X11 DISPLAY variable was set, but this program performed an operation which requires it.
        at java.awt.GraphicsEnvironment.checkHeadless(GraphicsEnvironment.java:204)
        at java.awt.Window.<init>(Window.java:536)
        at java.awt.Frame.<init>(Frame.java:420)
        at javax.swing.JFrame.<init>(JFrame.java:233)
        at RTC$Display.<init>(RTC.java:454)
        at RTC.<clinit>(RTC.java:59)
        ... 6 more
```

Figure 2.9: Cluster Mode Test Issues

Below shows the issue regarding the displays as talked about before.



```
^C[up926975@mn01 ~]mpjrun.sh -np 2 -dev niodev -headnodeip mn01.soc RTC
MPJ Express (0.35) is started in the cluster configuration
Starting process <0> on <wn02.soc>
Starting process <1> on <wn03.soc>
Exception in thread "main" java.lang.ExceptionInInitializerError
        at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
        at sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessor
        at sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethod
        at java.lang.reflect.Method.invoke(Method.java:498)
        at runtime.daemon.Wrapper.execute(Wrapper.java:165)
        at runtime.da                 (Wrapper.java:180)
Caused by: java.awt.H
No X11 DISPLAY variab                  this program performed an operation wh
        at java.awt.                  .checkHeadless(GraphicsEnvironment.j
        at java.awt.                  w.java:536)
        at java.awt.                  java:420)
        at javax.sw                   rame.java:233)
        at RTC$Displ                  454)
        at RTC.<cli
        ... 6 more
```

Ray Tracing@wn02.soc
Ray Tracing@wn02....
lease support MobaXterm by subscribing to the professional edition here: https://mobaxterm.mobatek.net

Figure 2.10: Cluster Mode Test Issues Display

3 Conclusion

In conclusion, throughout this entire period of learning, I have come to better understand what parallel programming means, even if it is in a more broader sense. From threading in the Java programming language, where it possible to handwrite each thread that will be used or to generalise it so that the user only needs to change how many threads are used rather than rewriting the entire code, to the concept of multicore parallel programming and cluster parallel programming. The concept of using MPI and MPJ programming were also covered which was also coded in Java. Each lab week covered a different topic regarding different problems such as pi, Conway's Game of Life and the Mandelbrot Set. During the first few lab weeks, it taught us how parallel programming in Java regarding the concepts that were mentioned earlier along new topics like cyclic decomposition and barriers.

In the later weeks, the lab sheets taught us how to code in Java regarding MPI and MPJ. This also allowed us to learn about multicore programming using a single server provided to us by the university. It also allowed us to learn and utilise the many worker nodes available to us, which allowed us to use cluster programming across multiple nodes.

What was analysed showed that even when a program has become parallelised, it does not always mean that it will run faster. There are many reasons for the program to not run faster, it may not have a good connection to server, the problem size may be too small which could cause the program to not work as efficiently as it would when the problem size was larger, or it may be hardware issue.

Using the knowledge learnt throughout each lab book, the project, a ray tracing program, was done to utilise both generalised parallel programming and MPJ task farm. The generalised ray tracing program showed that program got faster even as the problem sized increased, this was probably due to the programs efficiency increasing. The MPJ task farm however did not work as intended, this may was due to my lack of knowledge and understanding in the topic area and my ability to better adapt the code. For future work it would best to go over MPJ task farms and learn to how better adapt the code so that it can parallelise the ray tracing program.