UP877962

Parallel Programming Report

Introduction

This report details the activities carried out over the course of the semester in the parallel programming module. The report is split into two sections with part 1 containing the lab book recordings and write up of the work and activities carried out throughout the unit with tables used to display the results from the various benchmarks, for the purposes of readability the fastest times in each category of the tables have been highlighted in bold, and the best achieved speed up recorded.

The second part of the report details the work carried out in developing parallel implementations of a given problem that can be solved sequentially in Java and then testing and benchmarking these implementations, the problem chosen to be parallelised was that of creating a prime number sieve which is a method of determining if a given number is a prime or not, in the case of the implementation created it checks that numbers within a range of 1 and N numbers and returns a count of how many are found and can also optionally print these numbers however for performance reasons this is not used. There were some issues when using the supplied cluster and these have also been documented.

Part 1 - Lab Book

For the purposes of benchmarking each program was run 4 time and all scores recorded, the best time was then used to benchmark between various versions of the sequential program including benchmarking parallel versions against one another The parallel speed up formula of sequential time/parallel time was used to show these. Due to using a computer with an i5 6200U with only 2 cores and 4 threads there were limitations with how much testing could be carried out i.e., using 8 threads.

Week 1

The initial version of the program without accurate timings was run however all gave a run time of 0 seconds and Pi correct and the same to 12 decimal places so were not incorporated into a table.

The values of Pi achieved for this part are recorded below.

Sequential: Value of pi: 3.141592653589731
Parallel: Value of pi: 3.141592653589923

The following table (Table 1.1) displays the recorded times with the Pi programs updated to display the execution time in milliseconds, with the best times used a parallel speed up of 1.97 is achieved which is very close to the figure of 2.00 given in the tutorials as a “good” speed up. The values of Pi achieved are as follows.

Sequential PI:
Value of pi: 3.141592653589731
Parallel Pi
Value of pi: 3.141592653589923

*Table 1.1***.** Parallel/sequential Pi Second run with millisecond timing.

Run	Sequential time	Parallel time	Best parallel speed up
1	110ms	50ms	
2	**93ms**	54ms	
3	97ms	**47ms**	
4	98ms	53ms	
			93/47 = 1.97

The table below (Table 1.2) displays the results when the number of steps is reduced to 1 million, it can be clearly seen that the reduction in the complexity of the problem has reduced the benefits of parallelising the code. The values of PI are as follows.

Sequential.
Value of pi: 3.1415926535897643
Parallel:
Value of pi: 3.1415926535898993

Table 1.2. Parallel/sequential Pi Reduction to 1 million steps.

Run	Sequential time	Parallel time	Best parallel speed up
1	16ms	**15ms**	
2	16ms	15ms	
3	14ms	16ms	
4	**13ms**	16ms	
			13/15 = 0.86

The following table (Table 1.3) displays the results when the number of steps is increased to 1 billion, it is clear from the parallel speed up from 2.07 that increasing the complexity of the problem improves parallel results, the Pi results are as follows.

Sequential:
Value of pi: 3.1415926535899708

Parallel:
Value of pi: 3.141592653589901

Table 1.3, Parallel/sequential Pi increased to 1 billion steps.

Run	Sequential time	Parallel time	Best parallel speed up
1	10080ms	**4388ms**	
2	9154ms	4957ms	
3	**9114ms**	4685ms	
4	9120ms	4620ms	
			9114/4388 = 2.07

Sequential:
Value of pi: 3.1415926535899708

Parallel:
Value of pi: 3.141592653589901

Explaining the results:

The results show the known fact that parallel computing is more useful when using larger numbers/ amounts of data etc, the difference with 1 million as the base counter was negligible however was much larger when the base counter was increased to 1 billion. The value of pi was accurate each time to at 12 decimal spaces.

Exercise 1

.
The table (Table 1.4) below displays the results once the parallel version is modified to use 4 threads instead of 2, again it is clear to see from these results that the more complex version with 10 million steps performed 300% better when using parallel speed up to determine performance. There were issues with the implementation when increasing the numsteps value to 1 billion in that the calculation of PI was doubled each time, it appears this may be down to the ordering in which the thread start/join operations and is something that will be revisited again if time allows.

10 million steps
Value of pi: 3.1415926535896697
1 million steps
Value of pi: 3.1415926535898757

Table 1.4. 4 threaded Pi implementations.

Run	10 million steps	1 million steps	1 billion steps	Parallel Speed up 10 million	Parallel Speed up 1 million
1	59ms	**13ms**	n/a		
2	**31ms**	16ms	n/a		
3	44ms	16ms	n/a		
4	49ms	14ms	n/a		
5 sequential time (best)	**93ms**	**13ms**	n/a		
				93/31 = 3.0	13/13 = 1.0

Exercise 2

Nano time

Nano-time is far more accurate and is the most precise system java system timer however it is far more expensive to call and as such can have an adverse effect on program running times. Nano time is also not recommended for multithreaded programming due to the difference in nanoseconds being positive or negative, nano-time is unaffected by external influences such as the user changing system time.

Week 2

Exercise 1, parallel speed up threads split on I.

The timings recorded for the provided sequential and parallel Mandelbrot program are displayed in the table below (Table 2.1), the best speed up achieved was 1.21.

Exercise 2, threads split on J.

The timings recorded for sequential and parallel Mandelbrot program with the workload split on the J variable (horizontal axis) are displayed in the table below, the best speed up achieved was 1.87 which is ~30% faster than the version where the workload is split vertically. This is due to the image being processed symmetrically allowing the two threads to have balanced loads and process the image more efficiently.

Table 2.1. Mandelbrot set, sequential and parallel versions splitting the workload horizontally and vertically.

Run	Sequential mandelbrot	Parallel Mandelbrot 2 threads (divided by I)	Parallel Mandelbrot 2 threads (divided by J)	Parallel Speed up (divided by I)	Parallel Speed up (divided by J)
1	1063ms	1015ms	553sm		
2	1154ms	781ms	**500ms**		
3	**937ms**	**771ms**	859ms		
4	1536ms	780ms	547ms		
				937/771 = 1.21	937/500 = 1.87

Exercise 3, 4 thread Mandelbrot and comparison with 2 threads from above

The table below (Table 2.2) displays the speed ups between the sequential and 4 thread version and the 2 thread and 4 thread version, reasonable improvements have been achieved by doubling the number of cores, however a parallel speed up of more than 2 was not achieved.

Table 2.2. Comparison of best times achieved for the Mandelbrot set program execution.

Run	Parallel Mandelbrot 2 threads (divided by I)	Parallel Mandelbrot 2 threads (divided by J)	Parallel Mandelbrot 4 threads (divided by I)	Parallel Mandelbrot 4 threads (divided by J)	Parallel Speed up from 2 threads to 4 (divided by I)	Parallel Speed up from 2 threads to 4 (divided by J)
1	1015ms	553sm	641ms	**469ms**		
2	781ms	**500ms**	**614ms**	499ms		
3	**771ms**	859ms	659ms	796ms		
4	780ms	547ms	655ms	563ms		
					771/614 = 1.25	500/469 = 1.06

Exercise 3, 2 block-wise decomposition (division by I excluded)

The table below (Table 2.3) describes the times achieved when a block-wise implementation is used It is clear from the results shown that using a block wise decomposition in this situation is substantially faster with a parallel speed up of 3.19 achieved form the base sequential version and a parallel speed up of 1.6 for the improvement from the fastest previous version created for exercise 2.

Table 2.3. Comparison between previous implementations and a block-wise implementation.

run	Sequential Mandelbrot	4 thread block-wise decomposition.	Parallel Mandelbrot 4 threads (divided by I)	Parallel Mandelbrot 4 threads (divided by J)	Parallel speed up from sequential to 4 thread block-wise	Parallel Speed up from 4 threads divided by j to 4 thread block-wise
1	1063ms	**293ms**	641ms	**469ms**		
2	1154ms	297ms	**615ms**	499ms		
3	**937ms**	300ms	659ms	796ms		
4	1536ms	301ms	655ms	563ms		
					937/293 = 3.19	469/293 = 1.60

Week 3

- In answer to the question proposed, the loops cannot be merged as this would result in the threads starting and joining instantly before being used, this would likely lead to a deadlock situation.

The table below (Table 3.1) contains all the timings recorded for the generalised Mandelbrot implementations, the cyclic implementation was significantly faster with there being a parallel speed up of 1.87 achieved in the two threaded cyclic decomposition in comparison to a speed up of 1.25 (937/746) achieved by the two threaded block-wise decomposition, also the 4 threaded cyclic decomposition achieved a speed up of 3.23 in comparison to the block wise decomposition achieving a speed up of 1.87 (937/499). The most interesting difference is the performance gained by increasing thread count with the block-wise decomposition achieving barely any speed up while the cyclic decomposition almost doubled in speed with a doubled thread count showing that for this problem a cyclic decomposition is preferable for scalable performance gains.

Table 3.1. Generalised Mandelbrot comparisons.

run	Sequential Mandelbrot	Generalised Mandelbrot 2 threads Block-wise	Generalised Mandelbrot 4 threads Block-wise	Generalised Mandelbrot 2 threads Cyclic	Generalised Mandelbrot 4 threads Cyclic	Parallel speed up 2 threads cyclic	Parallel speed up 4 threads cyclic
1	1063ms	**746ms**	841ms	806ms	500ms		
2	1154ms	797ms	811ms	**499ms**	**290ms**		
3	**937ms**	947ms	786ms	789ms	460ms		
4	1536ms	802ms	**778ms**	500ms	445ms		
						937/499 = 1.87	937/290 = 3.23

Exercises

The sequential game of life program was successfully converted into a generalised parallel version that the compiler accepted however it clearly did not run correctly. the images below (Figure 3.1) display the relevant segments of code and an image of the output (Figure 3.2) when the number of threads is set to 256. Without the inclusion of barriers, a race condition is occurring meaning the results are not accurate.

```
public class ParallelLife extends Thread {

    final static int N = 256 ;
    final static int P = 256 ;  // Number of threads
    final static int CELL_SIZE = 4 ;
    final static int DELAY = 0 ;

    static int [][] state = new int [N][N] ;
    static int [][] sums  = new int [N][N] ;

    static Display display = new Display() ;

    public static void main(String args []) throws Exception {

        // Define initial state of Life board

        for(int i = 0 ; i < N ; i++) {
            for(int j = 0 ; j < N ; j++) {
                state [i] [j] = Math.random() > 0.5 ? 1 : 0 ;
            }
        }

        display.repaint() ;
        pause() ;

        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() ;
        }
    }

    int me ;

    ParallelLife(int me) {
        this.me = me ;
    }

    final static int B = N / P ;  // block size

    public void run() {
```

```
public void run() {

    int begin = me * B ;
    int end = begin + B ;

    // Main update loop.

    int iter = 0 ;
    while(true) {

        if(me == 0)
            System.out.println("iter = " + iter++) ;

        // 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] ;
            }
        }



        // Update state of board values.

        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 3.1. The code sections that were modified.

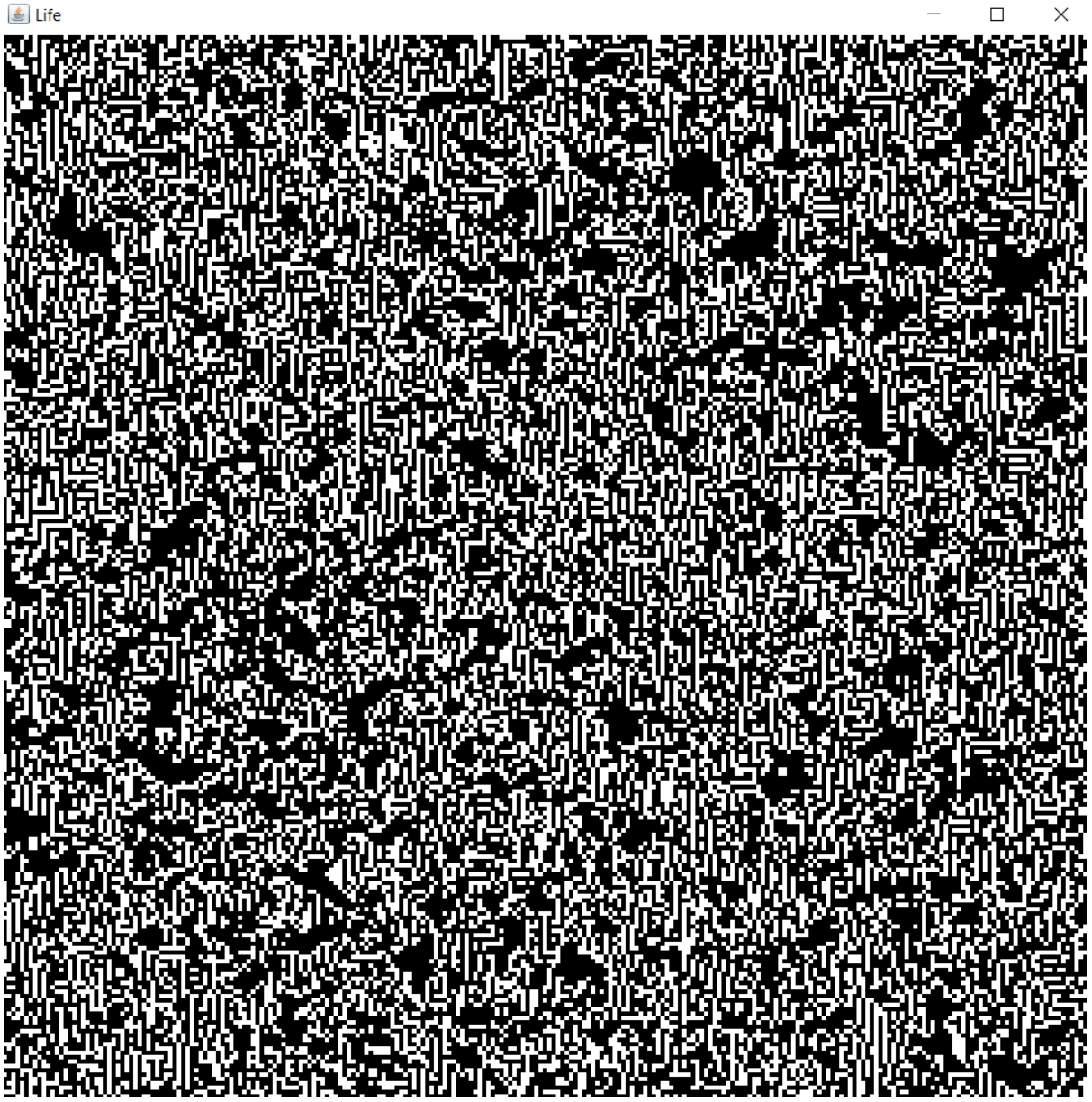

Figure 3.2. An image of the output when the number of threads requested is 256.

Week 4

Both synchronisation steps are required to give a true representation of the algorithm as the process of finding neighbours and updating the board are two distinct processes that cannot have any overlap, if the neighbours on one thread are found and then instantly updated before other threads have found their neighbours then the updates on those threads will not necessarily be the same if the threads had all waited on completion of the finding neighbours' step. Using both barrier calls prevent a race condition occurring.

Exercise 1

The figures below (Figure 4.1 – Figure 4.2) show the code used for this question; the display class was omitted as not modified in any way.

```
package parallellaplace;

/**
 *
 * @author pompe
 */
import java.awt.* ;
import javax.swing.* ;
import java.util.concurrent.CyclicBarrier ;

 public class ParallelLaplace extends Thread {

    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 = 4 ;  // Number of threads

    static float [][] phi = new float [N][N] ;
    static float [][] newPhi = new float [N][N] ;

    static Display display = new Display() ;

    static CyclicBarrier barrier = new CyclicBarrier(P) ;

    public static void main(String args []) throws Exception {

        // Make voltage non-zero on left and right edges

        for(int j = 0 ; j < N ; j++) {
            phi [0] [j] = 1.0F ;
            phi [N-1] [j] = 1.0F ;
        }



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

int me ;

ParallelLaplace(int me) {
    this.me = me ;
}

final static int B = N / P ;  // block size

public void run() {

    int begin = me * B ;
  int end = begin + B ;

  if(me == 0)
      begin = 1 ;

  if(me == P-1)
      end = N - 1 ;




   display.repaint() ;

  // 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]) ;
```

Figure 4.1, Code used for this section.

```
                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 & me == 0) {
            System.out.println("iter = " + iter) ;
            display.repaint() ;
        }
    }

    long endTime = System.currentTimeMillis();

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

    display.repaint() ;
}
```

Figure 4.2. Code used in this section (continued).

Exercise 2

The tables below (Table 4.1 – Table 4.2) show the timings recorded for the Laplace equation for a sequential implementation and a 2 and 4 threaded version with all versions executed with and without synchronisation barriers in the code, it was not possible to achieve a speed up greater than 2.00 with the barriers included however a best speed up of 2.12 was achieved without them being included demonstrating the costs involved in their usage.

Table 4.1, Laplace equation with barriers included.

run	Sequential Laplace	Parallel laplace 2 threads (with barriers)	Parallel laplace 4 threads (with barriers)	Parallel Speed up 2 threads	Parallel Speed up 4 threads
1	9879ms	8874ms	9692ms		
2	10023ms	8751ms	**9572ms**		
3	**9668ms**	**7784ms**	9612ms		
4	10052ms	7999ms	9625ms		
				9668/7784 = 1.24	9668/9572 = 1.01

Table 4.2, Laplace equation without barriers.

run	Sequential Laplace	Parallel laplace 2 threads (without barriers)	Parallel laplace 4 threads (without barriers)	Parallel Speed up 2 threads	Parallel Speed up 4 threads
1	9879ms	6390	4644		
2	10023ms	6541	4640		
3	**9668ms**	**6275**	4654		
4	10052ms	6853	**4546**		
				9668/6275 =1.54	9668/4546 = 2.12

Explaining the results:

This behaviour was unexpected as the 4 threaded version with barriers had very similar times to the sequential version while on removal of the barriers the speed up was dramatic, this may be an issue with my implementation, but it seems to be that the overheads of using barriers many times within nested loops may outweigh the potential benefits of using them or parallelising the program if they may not be needed such as could be the case for this equation. The fact that the implementation with barriers slowed down to almost the sequential speed with a speed up of just 1.01 indicates that if the thread count had been increased anymore the sequential version would likely have become faster removing the benefits of making the code parallel.

Unfortunately, no Linux machine was available for use when carrying out the exercises, however OpenMP was researched and its purpose as an API for scalable shared memory applications is understood.

Week 5

In response to the question an error is thrown if there are more processes requested than machines activated for the hello world program running in cluster mode. The image below (Figure 5.1) displays the implementation used.

```
[up877962@mn01 ~]$ cat 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() ;
      }
  }
[up877962@mn01 ~]$
```

Figure 5.1. The Pi program used.

The table below (Table 5.1) shows the timings recorded for both the sequential Pi program and the MPI version, there was a slight speed up, however nowhere near 2.00.

Table 5.1. N = 100million sequential/cluster mode Pi timings.

run	Sequential PI	MPJ Pi 2 processes	Parallel speed up
1	693ms	596ms	
2	**684ms**	**551ms**	
3	687ms	560ms	
4	688ms	558ms	
			684/551 = 1.24

Exercise 1

The table below (Table 5.2) shows the timings recorded for this question, as the number of cores goes above 8 the growth in speed increase begins to level off, this is likely to the low complexity of the problem and cores are not being used to their potential. The best parallel speed up recorded was an impressive 6.0 when utilising all the cores.

Table 5.2. Multicore mode N = 100 million

run	Sequential Pi	1 core	2 cores	4 cores	8 cores	10 cores	12 cores	Best Parallel speed up
1	**684ms**	665ms	378ms	**218ms**	**142ms**	**121ms**	117ms	
2	n/a	**659ms**	**357ms**	220ms	168ms	144ms	108ms	
3	n/a	666ms	403ms	239ms	161ms	128ms	**114ms**	
4	n/a	667ms	422ms	229ms	125ms	133ms	142ms	
								684/114 = 6.0

Exercise 2

For this question 5 machines were used, the cluster seemed to always have at least one other user online and there was a desire not to impact others if possible. This version when initially using one process per machine did not achieve as good a parallel speed up as the multicore version when comparable numbers of processes are used with the best parallel speed up achieved being 1.93. (Table 5.3) It appears message passing is clearly having a detrimental effect in places here and so the number of processes was increased to 6,8, and 10 with a parallel speed up of 4.04 achieved when using 10 processes across 5 machines (Table 5.4). Now that more work is carried out on each machine the difference is noticeable and shows how much message passing was costing in terms of best speed up times.

Table 5.3. Cluster mode N = 100 million 5 additional machines were used here wn01-wn05

run	Sequential Pi	1 machine	2 machines	3 machines	4 machines	5 machines	Best parallel speed up
1	**684ms**	479	646	679	632	780	
2	n/a	479	622	562	641	746	
3	n/a	480	562	632	649	580	
4	n/a	482	649	520	**354**	710	
							684/354 =1.93

Table 5.4. Increasing process count above 5 to utilise multiple cores on different machines (5 used)

run	Sequential Pi	6 processes	8 processes	10 processes	Best parallel speed up
1	**684**	208	233	172	
2	n/a	817	497	180	
3	n/a	526	490	**169**	
4	n/a	837	285	190	
					684/169 =4.04

Week 6

The table below (Table 6.1) shows the timings recorded when running MPJ Laplace in multicore mode in comparison to the times gained from the previous exercise in week 4, surprisingly once the process count reached 8 the program was slower than the sequential version with a parallel speed up of 0.82 and so increasing the core count was abandoned. The best speed up achieved was 1.51 when using 2 processes, this is likely due to it being an MPJ implementation and not optimised for multicore mode. The estimated cost of messages passed is niter^3 + niter/2 for the display messages.

Table 6.1. MPI Laplace equation, multicore mode

run	Sequential laplace times from week 4	P = 1	P = 2	P = 4	P = 8	Best parallel speed up
1	9879ms	8915ms	6772ms	6957ms	12163ms	
2	10023ms	9092ms	7599ms	**6560ms**	**11735ms**	
3	**9668ms**	9031ms	**6396ms**	7652ms	13022ms	
4	10052ms	**8835ms**	6834ms	6835ms	12212ms	
						9668/6396 = 1.51

The table below (Table 6.2) displays the results when the program is run in 2 node cluster mode configuration and the timings for when the edge swap code is removed, this was incredibly slow in comparison to the sequential version with only the implementation without the edge swap code achieving a positive speed up of 2.04. and in fact, there was a 90% reduction in the time taken to compute the problem demonstrating the overhead involved in this part of the equation. The updated edge swap code was slower than the original which was unexpected.

Table 6.2. the timings for the Laplace equation

run	Sequential best time	2 node cluster mode, 2 daemons	Edge swap code removed	Updated edge swap code	Best Parallel speed up
1	**9668ms**	43190ms	4749ms	**50345ms**	
2		**42859ms**	4765ms	51538ms	
3		42927ms	4801ms	51262ms	
4		43206ms	**4728ms**	51737ms	
					2.04

The table below (Table 6.3) shows the time comparisons of the Laplace equation being run with various numbers of processes with 2 daemons used for each, there is ~10% speed up for every 4 processes added, but gain slowly tapered off after utilising 12 processes, the gains can be explained by the individual machines taking on larger portions of the code which helps balance the load and decreases the number of messages passed. However largely useless when considered against the sequential version being considered and even when using 2 processes as a baseline the best speed up is just 1.20.

Table 6.3. Cluster mode Laplace equation with multiple processes

run	Sequential best time	2 node cluster mode, 2 daemons	Cluster mode 4 processes	Cluster mode 8 processes	Cluster mode 12 processes	Cluster mode 20 processes	Best parallel speed up (from sequential)	Best speed up From 2 node cluster mode
1	**9668ms**	43190ms	**42277ms**	39665ms	37969ms	35900ms		
2		**42859ms**	42375ms	39606ms	37922ms	**35697ms**		
3		42927ms	42433ms	**39288ms**	**37498ms**	35983ms		
4		43206ms	42382ms	39445ms	37749ms	35855ms		
							n/a	42859/35697 = 1.20

The following tables (Table 6.4 - 6.5) displays the timings when the problem complexity is increased by increasing the value of N, the number of iterations were reduced but it can be seen by comparing the tables that increasing the complexity of the problem dramatically reduces the processing time with a best new speed up of 3.77 achieved. The tables also indicate that doubling the problem complexity in this case doubled the parallel speed up. As speedups are slowing 20 processes not were recorded.

Table 6.4. Laplace equation, N=512 niter = 10000

run	Sequential time based on week 4	Cluster mode 2 processes	Cluster mode 4 processes	Cluster mode 8 processes	Cluster mode 12 processes	Cluster mode 20 processes	Best parallel speed up
1	**9668ms**	8032ms	**6518ms**	5907ms	5286ms	n/a	
2		**7922ms**	6608ms	5818ms	**5161ms**	n/a	
3		7930ms	7703ms	**5720ms**	5289ms	n/a	
4		7934ms	7498ms	5816ms	5163ms	n/a	
							9668/5161 = 1.87

Table 6.5. N=1024 cell size 1, niter = 1000

run	Sequential time based on week 4	Cluster mode 2 processes	Cluster mode 4 processes	Cluster mode 8 processes	Cluster mode 12 processes	Best parallel speed up
1	**9668ms**	3033ms	2435ms	2540ms	**2560ms**	
2		**2906ms**	2517ms	2560ms	3175ms	
3		3180ms	2556ms	**2508ms**	2941ms	
4		3122ms	**2249ms**	2708ms	3480ms	
						9668/2560 = 3.77

Explaining the results:

The benefits of adding additional processes were reached around the 8-process mark. These were far faster times considering the problem size was larger, this is since parallel programming is more suited to larger problems with each individual process taking on a more even share of the overall problem and reducing the overheads from excessive message passing.

Week 7

The table below (Table 7.1) details the best recorded times for the slow Mandelbrot implementation when run on the cluster in multicore mode, increasing the core count had a dramatic effect here and despite not being able to utilise 12 cores a best speed up of 6.44 was achieved.

Table 7.1. Multicore comparisons against sequential Mandelbrot run on mn01.

run	Sequential slow Mandelbrot Run on mno1	2 core slow Mandelbrot	4 core slow Mandelbrot	8 core slow Mandelbrot	Best speed up
1	**27128ms**	26235ms	9461ms	4212ms	
2	27773ms	27150ms	9469ms	**4210ms**	
3	27708ms	27138ms	9554ms	4248ms	
4	27708ms	**26138ms**	**9444ms**	4280ms	
					27128/4210 = 6.44

The table below (Table 7.2) details the recorded times for the Mandelbrot task farm exercise, it can be determined from these timings that the percentage of decrease in times taken begins to taper off after 4 additional machines are used indicating that the effectiveness is unlikely to increase much more and that the multicore implementation may be more appropriate for this problem.

Table 7.2. MPJ Mandelbrot task farm

run	Sequential slow Mandelbrot Run on mno1	2 additional machines (1 workers)	3 additional machines (2 workers)	4 additional machines (3 workers)	5 additional machines (4 workers)	6 additional machines (5 workers)	Best speed up
1	**27128ms**	**26655ms**	13716ms	**9290ms**	7355ms	n/a	
2	27773ms	26800ms	13825ms	9429ms	7297ms	n/a	
3	27708ms	26689ms	**13631ms**	9348ms	**7142ms**	n/a	
4	27708ms	**26783ms**	14239ms	9291ms	7308ms	n/a	
							27128/7142 = 3.79

Explaining the results:

It was not possible to increase the number of nodes above 5 additional workers as the cluster began to throw error messages after the number of workers was increased above 5 despite 9 machines being available and as such are not included. Unfortunately, the block-wise implementation was not carried out however this would have likely given better results than processing strips as the program does not have to traverse the length/height of the N*N space to process each block therefore shortening the number of calculations.

Part 2 - Development Report

The section of the report details the work undertaken during the development project, the project chosen was the set exercise of parallelising a prime number sieve using the method explained, after a working sequential program was written the next task involved benchmarking a parallelised version using both a cyclic and blockwide decomposition, these tests were run on the local machine using a an upper bound of 10000 numbers. The implementation of both versions followed techniques learnt throughout the course of the unit to parallelise the problem with the material from week two, three and four particularly useful. the figures below (Figure 1.1 – Figure 1.3) display the major components of the implementation.

```
public class MultiCorePrimeSieve extends Thread{
    final static int S = 1;
    final static int N = 200000000;
    final static int P = 4;

    final static ArrayList<ArrayList<Integer>> primes = new ArrayList<>();
    final static int total = 0;

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


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

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

        ArrayList<Integer> primeList = new ArrayList<>();
        primes.forEach((list) -> {
            primeList.addAll(list);
        });

        long endTime = System.currentTimeMillis();

        System.out.println("Found "+ primeList.size() +" primes across "+
        //System.out.println(primes);
    }

    int me;
    public MultiCorePrimeSieve(int me) {
        this.me = me;
    }
```

Figure 1.1. The main code used for the multicore implementation.

```
//blockwise decomposition, works best here

public void run(){

    int begin = me * (N / P);
    int end = begin + (N / P);

    ArrayList<Integer> prime = new ArrayList<>();

    for(int i = begin; i < end; i++){
        if(isPrime(i)){
            prime.add(i);
        }
    }

    primes.add(prime);
}
```

Figure 1.2. The java code used to test a block-wise decomposition.

```
//cyclic decomposition

public void run(){
    //blockwise decomposition works best here, do coMparison

    ArrayList<Integer> prime = new ArrayList<>();

    for(int i = me; i < N; i+=P){
        if(isPrime(i)){
            prime.add(i);
        }
    }

    primes.add(prime);
}
```

Figure 1.3. A cyclic decomposition.

The table below (Table 1.1) describes the recordings when these two decomposition methods were compared, for this problem a block wise decomposition works better and therefore this method will be used for further benchmarks between the different versions.

Table 1.1. A comparison of the test runs between the different decompositions.

run	Cyclic decomposition 4 threads	Block-wise decomposition 4 threads	Best time
1	312ms	359ms	
2	263ms	234ms	
3	333ms	**212ms**	
4	258ms	219ms	
			212ms

The following table details the results of the multicore implementation (Table 1.2), and the fastest achieved speed up achieved when run on the head node (mn01) of the cluster, the sequential times are used as a comparison to calculate the best achieved parallel speed up. For the purposes of testing on the cluster and to get a better distinction between times the complexity of the problem was increased, and the program now looks for primes up to an upper bound of 20 million. This implementation worked well on this problem with a parallel speed up of 7.69 achieved when all 12 cores on the head node were used.

Table 1.2. Generalised parallel prime sieve run on mn01 in comparison to sequential times and different numbers of cores.

run	Sequential time	Generalised Primes, 2 threads	Generalised Primes, 4 threads	Generalised Primes, 8 threads	Generalised Primes, 12 threads	Best speed up
1	**16039ms**	**10382ms**	5943ms	3567ms	**2083ms**	
2	16115ms	11453ms	**5915ms**	**3538ms**	2160ms	
3	16052ms	10451ms	5922ms	3560ms	2435ms	
4	16128ms	10422ms	5918ms	3557ms	2417ms	
						16039/2083 = 7.69

To create the mpj implementation the material from weeks 4, 5 and 6 was used in combination with the previous tutorials work, the first thread is used to manage the storage of prime numbers and deals with the print statement while the rest are used to determine if a number is prime. The figure. Below (Figure 1.4) detail the key part of the code used for creating this implementation.

```
if(me > 0){
    int[] sendBuf = new int[]{temporaryStorage.size()};
    MPI.COMM_WORLD.Send(sendBuf, 0, 1, MPI.INT, 0, 1);

    for(int i = 0; i < temporaryStorage.size(); i++){
        int[] send = new int[]{temporaryStorage.get(i)};
        MPI.COMM_WORLD.Send(send, 0, 1, MPI.INT, 0, 0);
    }
} else {
    for(int i = 0; i < temporaryStorage.size(); i++){
        primes.add(temporaryStorage.get(i));
    }

    int[] recvBuf = new int[1];
    for (int src = 1; src < P; src++) {
        MPI.COMM_WORLD.Recv(recvBuf, 0, 1, MPI.INT, src, 1);

        for (int ind = 1; ind <= recvBuf[0]; ind++) {
            int[] recv = new int[1];
            MPI.COMM_WORLD.Recv(recv, 0, 1, MPI.INT, src, 0);
            primes.add(recv[0]);
        }
    }
}
```

Figure 1.4. The main part of the code containing the mpj message passing logic.

The next table (Table 1.3) describes the times achieved when the program was modified to work with mpj express running in multicore mode, it is clear from the timings that the addition of the mpj code is having a substantial negative affect on the performance of the program and that the gains are quickly diminishing with the addition of more cores and a best achieved speed up of just 1.52 in comparison to the 7.69 achieved by the generalised multicore version, showing it is around 80% less efficient.

Table 1.3. The timings recorded with the mpj implementation running in multi core mode.

run	Sequential time	MPJ multicore, 2 threads	MPJ multicore, 4 threads	MPJ multicore, 8 threads	MPJ multicore, 12 threads	Best speed up
1	**16039ms**	16662ms	12729ms	**10504ms**	11425ms	
2	16115ms	15497ms	13298ms	10509ms	**10921ms**	
3	16052ms	17043ms	**12063ms**	10606ms	10931ms	
4	16128ms	**16157ms**	12119ms	11789ms	11608ms	
				x		16039/10504 = 1.52

The table below (Table 1.4) describes the attempts to run the mpj implementation in the mpj express cluster mode, there were issues with using more than 5 nodes at one time that meant it was not possible to test it beyond this number. Figures have also been included below (Figure 1.5 – Figure 1.7) to show the attempts to start the mpj daemon process on the targeted machines and the resulting failure responses. Despite the difficulties with testing, this implementation performs in a very similar manner to the multicore implementation up to the 4 thread/node point indicating that there is very little benefit to running a clustered implementation when using a low number of additional processes or on smaller problems.

Table 1.4. The timings achieved when running the mpj implementation.

run	Sequential time	MPJ cluster, 2 nodes	MPJ cluster, 4 nodes	MPJ cluster, 5 nodes	Best speed up
1	**16039ms**	17553ms	12087ms	11060ms	
2	16115ms	**16026ms**	11752ms	11187ms	
3	16052ms	16150ms	**11197ms**	10793ms	
4	16128ms	17298ms	11940ms	**10729ms**	
				x	16039/10729 = 1.49

There was also an attempt to create an mpj task farm implementation however it was not possible to get a functioning program and time constraints meant that it had to be abandoned so all observations for this part of the report will be based around the available data gained. The non mpj multicore version was far more efficient when dealing with the problem than the similar mpj version showing the costs involved in the additional message passing code and even when the problem was run in cluster mode with the limited number of clusters available the gains had already begun to diminish by the time the 5th node was introduced with only a 4.2% increase in speed from the 4th node, this was likely down to the relatively small size of the problem meaning the overheads of the message passing were outweighing the benefits of adding additional machines but with only 5 available it was not worth rerunning all experiments at this late stage, although in theory a task farm implementation should be able to achieve good speed ups with the chosen problem due to the ability to distribute the workload across multiple cores on each individual machine offsetting the costs of the mpj message passing.

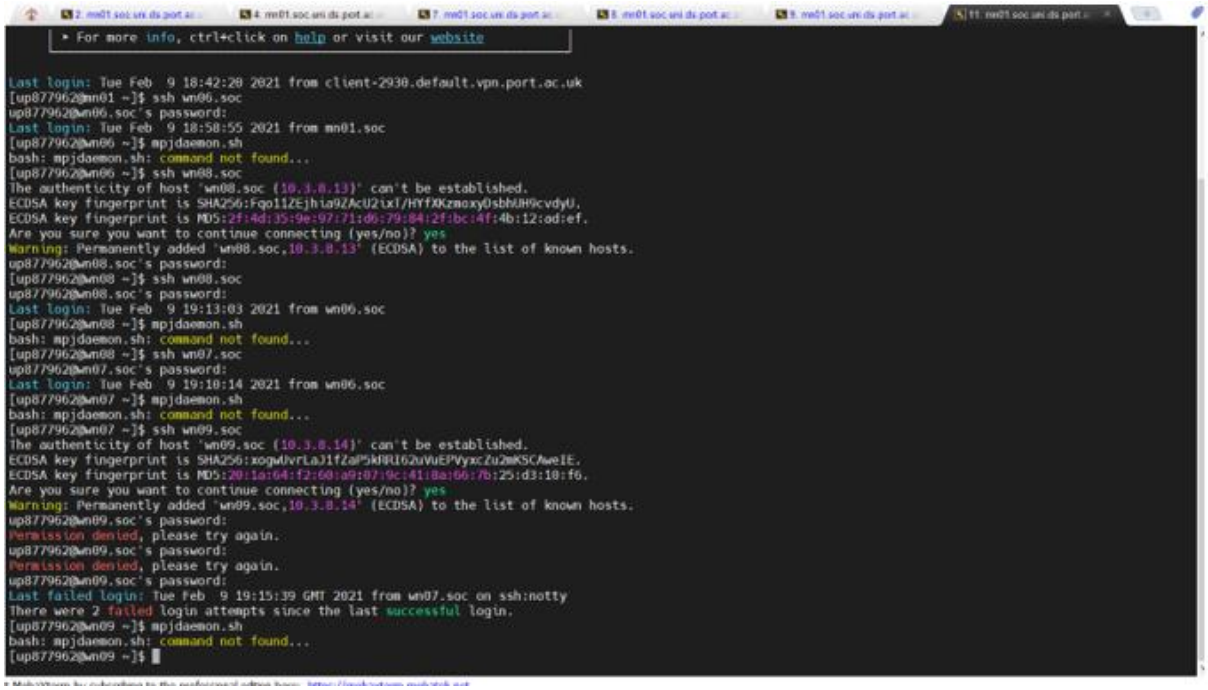

Figure1.5. 1st failed attempt at running more than 5 daemons.

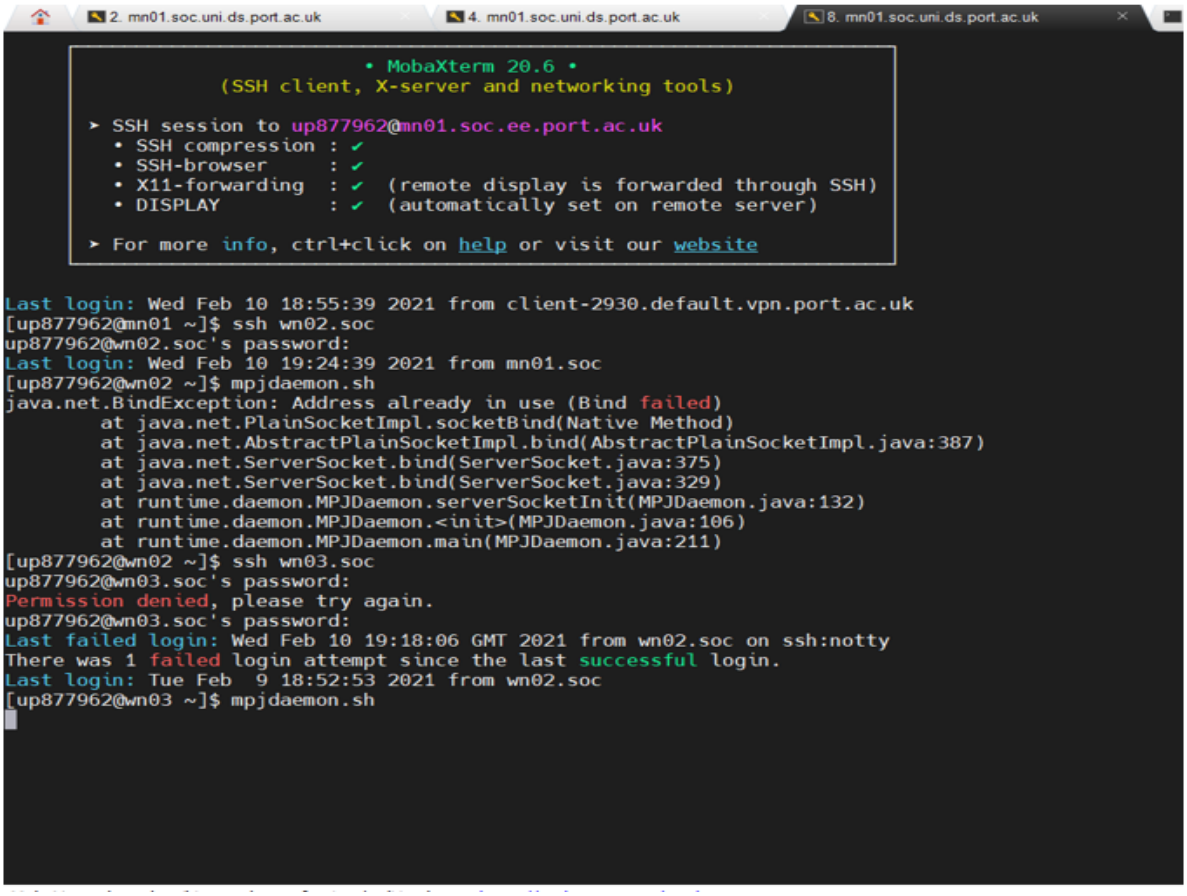

Figure1.6. 2nd experienced error with the cluster.

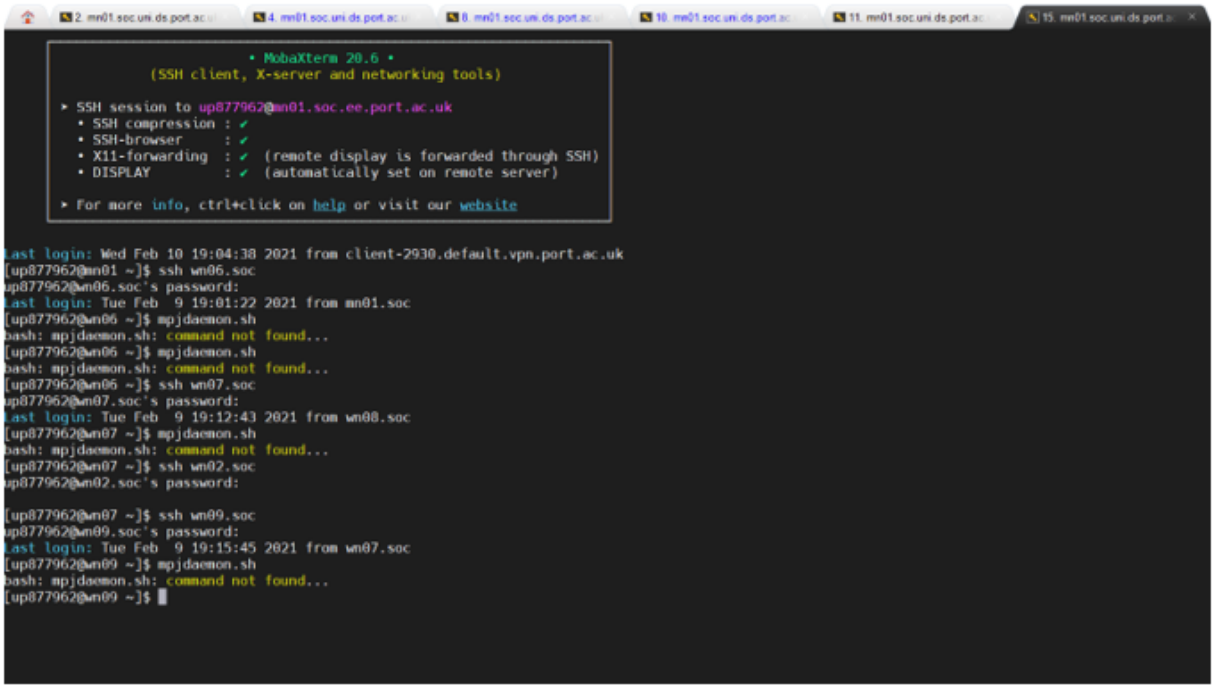

Figure 1.7. 2nd failed attempt at running more than 5 nodes.

Conclusion

In conclusion when used with a problem of an appropriate size parallel programming can give significant improvements over similar sequential implementations with some good parallel speed ups achieved throughout the process of compiling this report particularly when using a multicore implementation on a single machine, however the problem must be significant enough to justify the extra effort involved in parallelising sequential code.

When using distributed parallel programming across many machines the cost of message passing must also be considered as these overheads can quickly diminish gains achieved and in some cases may outweigh the benefits of making the program parallel in the first place, however spreading more work across individual machines cores on distributed systems such as when using a task farm gives the best benefit and a hybrid approach of allowing individual machines to process data in parallel while feeding data into a wider distributed parallel system should give the best all around performance while avoiding unnecessary bottlenecks and overheads from using excessive message passing.