Parallel Programming Coursework

UP781439

5th February 2021

School of Computing
University of Portsmouth

Contents

Introduction

The first half of this document will outline the work completed during the lab sessions for the Parallel Programming unit. This will include brief discussions of all the experiments that were conducted as well as the speedups and efficiencies obtained. All lab work was conducted on my own PC due to coronavirus and the CPU is a six-core Intel Core i5.

The second half of this document will describe my individual development project. For the project I chose to parallelise a prime number finder as well as attempting to parallelise Eratosthenes Sieve. The prime number finder finds all the prime numbers up to and including a given integer value. This has been accomplished with a distributed memory MPJ Express implementation on the Universities' Hadoop Cluster.

The aims and objectives as I see them for this project were to understand how to decompose sequential problems and then to calculate parallel speedups and efficiencies and how to accurately benchmark parallel programs.. Furthermore, I have gained a much deeper understanding into the different kinds of parallelism, shared and distributed memory, and how these are implemented.

Lab Book

The labs started by downloading and installing Java JDK and Netbeans IDE so that we could use the Thread library of the Java language. This allows programmers to initialise instances of Thread objects that can be executed concurrently, which is almost the same as parallelism except that 2 Java threads are not executing their processes at exactly the same time. The task for all weeks is to benchmark sequential versus parallel versions of an algorithm. The results of this shall be displayed as a table showing ten runs and the fastest of the ten highlighted in green. The fastest run is the value that has been used to calculate speedups and efficiencies throughout.

Week 1 - Java Threads Reprise

The parallelisation tasks of the first few weeks started off with embarrassingly parallel problems. This means that it is very easy to break the problem up into smaller subtasks that can be completed by any number of threads. To decompose the sequential problem you can simply split the range of numbers to be computed into several ranges to be executed by individual threads. The sequential algorithm performance is displayed in Figure 1, the fastest it achieved was 48 milliseconds.

Run	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
Time (ms)	48	49	49	49	48	49	49	49	49	49

Fig. 1 Sequential Pi running times

The parallel version of this algorithm was initially built to work with two threads but was altered to be a 'quad-core' version and use four threads. Figure 2 shows the results of running the parallel Pi approximation using two threads. The fastest time it obtained was 22 milliseconds meaning that it achieved a parallel speedup of 2.18.

Run	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
Time (ms)	22	22	22	22	23	22	22	23	22	23

Fig. 2 Parallel Pi two threads running times

The program was made to run on four threads by simply creating two extra threads and dividing the range of operations to be executed by four instead of by two. The results in Figure 3 show that with four threads we achieved a parallel speedup of 3**.** This is even better and shows that with four threads we were able to get to an approximation of Pi three times faster than the sequential algorithm as able to.

Run	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
Time (ms)	18	18	18	18	18	18	17	18	17	16

Fig. 3 Parallel Pi four threads running times

Week 2 - Threads and Data

The sequential Mandelbrot Set calculation involved filling an N x N 2 dimensional array where N = 4096, with values of the mandelbrot set that are calculated for those index positions. The array values are calculated row at a time, from left to right. The execution times are displayed in Figure 4.

Run	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
Time (ms)	566	921	563	563	563	560	566	563	560	564

Fig. 4 Sequential Mandelbrot calculation times

The fastest time that was achieved was 560 milliseconds and this shall be used as the baseline sequential speeds in benchmarking the parallel versions. The first parallel version of this algorithm was run with four threads and decomposed horizontally, along the outer loop. The fastest time over the ten runs was 385 milliseconds (Figure 5) meaning that this achieved a parallel speedup of 1.45.

Run	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
Time (ms)	400	391	516	519	519	530	385	387	394	519

Fig. 5 Four thread, block-wise Mandelbrot calculation times

Next I decomposed the problem vertically, along the inner loop. This achieved better results, getting a fastest time of 281 milliseconds (Figure 6), giving this method a speedup of 1.992. This achieved better results because the shape of the mandelbrot set is such that splitting the workload horizontally is not optimal. Splitting this problem vertically created a better spread of the workload.

Run	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
Time (ms)	283	425	426	423	421	425	281	284	282	285

Fig. 6 Four thread, vertical decomposition calculation times

Finally the problem was split along both axes, the outer and inner loop. Dividing the mandelbrot set into 4 squares showed to be the most efficient way to calculate all of the values with 210 milliseconds (Figure 7). This meant a parallel speedup of 2.66 showing this was the best way to decompose this problem.

Run	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
Time (ms)	330	248	329	329	330	210	328	327	331	329

Fig. 7 Four thread, 2 axis calculation times

Week 3 - A Simulation

Mandelbrot Generalisation

Initially, we generalised a parallel implementation of calculating the mandelbrot set to work with any number of threads, P, that are factors of the problem size, N. This is achieved by looping through a range, zero to P, and creating, starting and waiting threads. Using this method, block decomposition and four threads we achieved 395 milliseconds (Figure 8) fastest time and 1.41 parallel speedup. It was run again in the same way with six and eight threads and did not reach a parallel speedup of two until eight threads.

Run	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
Time (ms)	523	517	521	395	514	517	518	517	523	516

Fig. 8 Four threads, generalised Mandelbrot Set Calculation speeds

Parallelising Life

The second task in Week 3 was parallelising a simulation of Conway's Game of Life. Initially using Bryans' sequential algorithm for the Game of Life I was able to run this and it worked correctly (Figure 8).

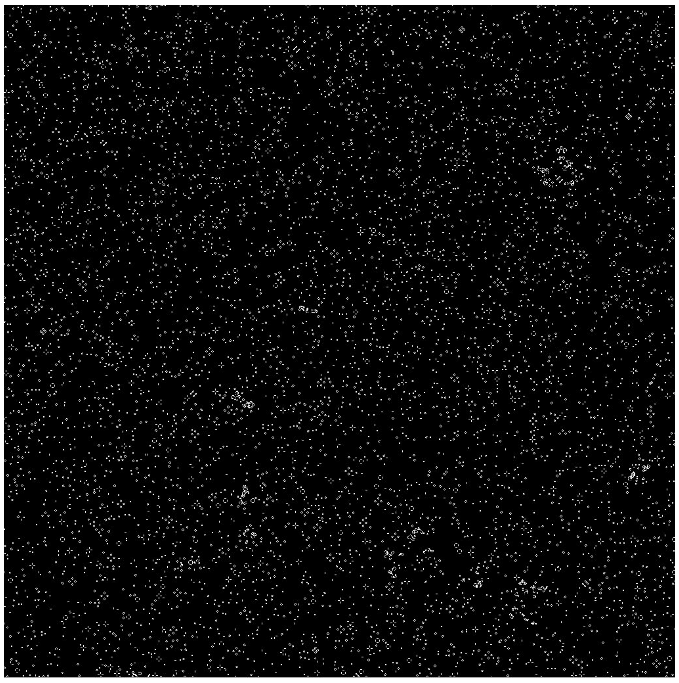


Fig. 8 Sequential Game of Life after ~400 generations

Using what I had just learned from the generalisation of mandelbrot calculation, I altered the sequential Life to implement a parallel, generalised algorithm to play Life. The main computation loop is decomposed horizontally, along the i-loop. Figure 9 displays the parallel version after 4000 iterations for comparison with Figure 8.

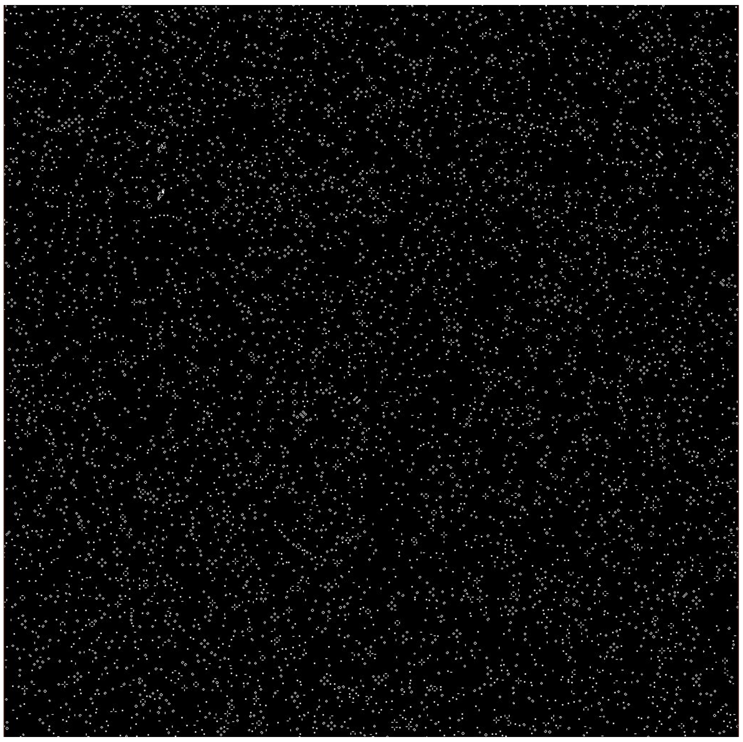


Fig. 9 Parallel Game of Life after ~4000 generations

Finally to get a base idea of how the parallelism befitted this computation, timers were put before and after the neighbour counter and board updater loops to calculate exactly how long it takes thread[0] to do those computations per iteration. Average speed for four threads = 13ms and six threads = 8ms.

Week 4 - Interacting Threads

This week we parallelised finding an approximate solution to the Laplace Equation. The sequential version worked by doing sufficient iterations of averaging all values until the rule works across the board. They repeat this until an acceptably accurate solution to the problem is found, after 500k iterations. Parallelised the problem by creating P many threads. Each thread then goes through all iterations updating its own block in the larger image. Figure 10 shows the sequential results and Figure 11 shows the parallel with six threads.

Run	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
Time (ms)	12258	12050	11980	12029	12189	10634	12107	10579	10551	10618

Fig. 10   Sequential Laplace computation times

Run	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
Time (ms)	2037	2114	2450	2460	2440	1927	2706	2375	2497	2245

Fig. 11   Parallel Laplace computation times (six threads)

Six thread, Parallel Laplace achieved a speedup of 5.47 which is an extremely good speedup. However, this is without synchronization and barriers, which is why the threads all finished at different times and results cannot be trusted. Figure 11 shows results with synchronisation.

Run	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
Time (ms)	4859	5452	4643	4530	5584	5084	4550	5135	4680	5041

Fig. 11   Parallel Laplace computation times (six threads)

Synchronisation has added a lot of extra computation to the program and is now achieving a speedup of 2.32, however we can now be sure that the results are reliable. It was shown in subsequent experiments that as you increase the number of threads, and therefore the number of calls to Synchronisation() method, the timings get progressively worse.

Week 5 - Running MPJ Programs

MPJ Express is a Java message passing library that allows the development of distributed memory parallel programs in Java language. We were able to use the universities Hadoop Cluster which is a cluster of several machines that can be remote accessed and perform distributed memory parallel computations. The Cluster has two modes of operation; multicore and cluster mode. Multicore is the shared memory, single machine mode and Cluster mode is the distributed memory mode.

The algorithm to benchmark this week was our Pi problem from Week 1. Figure 12 shows the results of running the sequential version on the cluster and Figure 13 shows the results of the distributed memory version run with two threads and two nodes.

Run	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
Time (ms)	750	745	753	758	743	754	741	744	754	755

Fig. 12 Sequential Pi calculation

Run	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
Time (ms)	587	608	558	575	556	596	582	615	601	595

Fig. 13 Distributed Pi calculation (2 threads, 2 nodes)

When run in this configuration the distributed version achieved a parallel speedup of 1.33 which isn't very good. However, as you increase the number of threads and nodes being used the speedup gets better. This shows that distributed memory parallelism is better suited as solutions to massive problems with lots t of data to be computed.

Week 6 - MPJ Communication

This week focuses on exploring the differences of multicore and cluster mode as well as adding edge swapping to a MPJ version of the Laplace equation. Firstly we ran a shared memory version of Laplace with two threads, see Figure 14. This produced a speedup of 1.21 and when run again with 4 threads it achieved a speedup of 1.56.

Run	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
Time	8206	7943	7080	7936	7449	7508	7326	8015	7023	8314

(ms)										

Fig. 14 Multicore Laplace computation times

Next, we explored the addition of edge swapping and ignoring redundant ghost regions that don't need to be updated. The results of adding these features are in Figure 15. As you can see a lot of time has been added to the computations.

Run	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
Time (ms)	43186	43112	43173	43101	43452	43309	43175	43402	43813	43144

Fig. 15 Edge Swapping Distributed Laplace Equation

Figure 16 shows the results without edge swapping. Evidently this is much faster, however I don't think it is making an accurate approximation of Laplace's equation based off of the graphical representation of what is happening. This experiment showed that the communication between nodes involved in the edge swapping functionality causes massive overheads in the execution of the code.

Week 7 - MPJ Task Farm

This week focused on implementing an MPI task farm to compute the members of the Mandelbrot Set. This approach gets each thread to work on small parts of the problem until they are complete, then get new tasks, until the whole problem is solved. The sequential version was also changed to simulate a higher workload. Results are in Figure 16.

Run	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
Time (ms)	26643	26524	26619	26638	26770	26573	26644	26595	26566	26564

Fig. 16 Sequential 'slow' Mandelbrot times

The task farm architecture has one master node and several worker nodes. The master node sends start and end points to the workers, once finished the workers request more work and the master sends it until the problem is solved. This was run on the Hadoop cluster in Cluster mode with two worker nodes.

Run	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
Time	14312	13695	13656	13962	13659	13679	13945	13689	13676	13649

(ms)										

Fig. 17 Task farm mandelbrot calculation speeds

Figure 17 shows a parallel speedup of 1.94 which is close enough to 2 to say that this is an ideal parallel solution to this problem. The next experiment done with the task farm is to test the maximum capabilities of parallel speedup that can be achieved with this setup. The Hadoop Cluster has five machines with twelve cores each means a total of 60 logical processors. Figure 18 shows the results of running 5 workers and 60 processes.

Run	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
Time (ms)	939	769	798	788	809	1338	925	953	941	1143

Fig. 18 Task farm mandelbrot calculation speeds

Achieving a fastest run of 796 milliseconds, the speedup was 34.49. This test really showed the power of parallel computing for jobs with massive workloads.

Week 8 - Choosing a Project

This week was just about choosing a project to complete for the development part of the coursework. I chose to develop a prime number sieve using an MPI task farm, and additionally attempt to develop Eratosthenes Sieve to be completed as a task farm.

Week 10 - Aparapi

Aparapi is a framework allowing developers to create programs that can be executed by the GPU. We will be comparing the performance of a matrix multiplication algorithm in sequential on the CPU and in parallel on the GPU. The sequential baseline results are in Figure 19. Another metric has been added here in Gigaflops (GFlops) which represents the floating point operations per second.

Run	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
Time (ms)	7347	7407	7415	7352	7353	7365	7424	7387	7388	7389
GFlops	2.338	2.319	2.316	2.336	2.336	2.332	2.314	2.325	2.325	2.325

Fig. 19 Sequential Matrix Multiplier Results

The instructions were followed to install aparapi dependency and then the parallel code was run. The results of running with the same problem size (N = 2048) as the sequential version are in Figure 20.

Run	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
Time (ms)	877	347	353	423	362	398	356	350	388	349
GFlops	19.589	49.509	48.668	40.614	47.458	43.165	48.258	49.085	44.278	49.225

Fig. 20 Aparapi GPU Matrix Multiplier Results

The GPU in my PC is a Nvdidia RTX 2070 with 8GB VRAM and 2304 cores. For comparison, my CPU is an Intel Core i5 with 6 cores. The speedup parallel speedup produced by the GPU is 21.17. This shows that even though GPU cores are operating at much lower clock speeds they are able to process large amounts of data in a fraction of the time due to the sheer number of cores in a GPU.

Development Project

Introduction

The second half of this coursework required students to choose a problem to parallelise. We were given a list of possible problems to tackle and I chose to attempt to parallelise and benchmark a prime number sieve, as well as attempting to parallelise Eratosthenes Sieve which is a famous Ancient Greek algorithm for finding prime numbers.

The aim of this was to amalgamate the skills and knowledge gained from the lab sessions to be able to parallelise and benchmark programs on our own without being given a lab script. The following will outline, in greater detail, my chosen problem, the development of it and how problems adaptations were made when problems arose. The benchmarking results will be displayed and analysed and there will be a final more brief section on the development of Eratosthenes Sieve.

Prime Number Sieve

A prime number Sieve will 'sieve' out all of the composite numbers in a range, leaving you only the prime numbers in that range. Prime numbers are those whose only factors are 1 and itself, excluding 0 and 1, and non primes are called composite numbers. This section will outline the algorithm that will be used and how it was developed and altered to get better performance.

The first method that was explored was to take a given integer number, N, and test every integer from 2 to N , X, for primeness by simply seeing if any number less than N, will divide it. If X is divisible by some value then the number cannot be prime, this is repeated for every value up to N. This will identify every composite number in the range and therefore, the numbers not identified are the prime numbers. This method will find all the primes however it is doing a lot of unnecessary computations. There is no need to divide by all values up until N. This can be made evident with an example; if $N = 500$ and $X = 36$ then there is no need to check numbers past $\sqrt{X}$ because, by definition, the square root will always be a factor of its square and if there is no integer root then there will be another factor before it and if there is not, then the number is prime.

The second method that was considered was exactly the same as the first except X would be in the range $2 <= X <= \sqrt{X}$. This eliminates a lot of unnecessary calculations and ensures that a solution is found in a faster time. However this method could still be improved upon. The program was still dividing X by more values after it had found the first factor, this slowed the program down again as it only needs to confirm that a number has 1 factor other than 1 or 0 to qualify as composite.

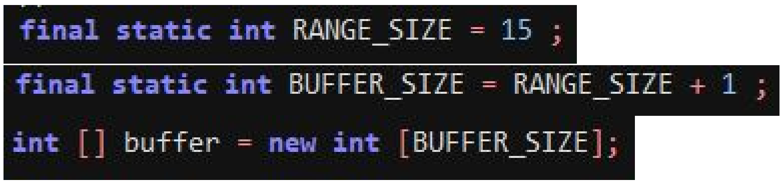
The final addition of the algorithm that was used in the main block of work for both the sequential and parallel versions of the algorithm incorporated all of the changes previously talked about with the addition of the use of the modulo operator to check if a number is divisible, If the result of the modulo function is 0 then X is composite. Now that I had an efficient and effective method for finding primes in a sequential manner, I had to decompose the problem and make it perform in parallel.

Parallelisation Process

The University has a Hadoop Cluster with 5 worker nodes capable of running MPJ Express which were utilised in the implementation of this program. The Hadoop Cluster and MPJ Express are used to develop the Message Passing Interface (MPI) task farm, prime number sieve in this project.

An MPI task farm is a method of distributed memory parallelisation where one computing node in the network acts as the `Master` and the others are the `Workers`. The `Workers` send requests for work to the master and the master decides what is the next block of work to be computed and sends that information back to the `Worker`, once the worker has finished that block it sends the results back to the `Master` and the process is repeated until all of the work has been completed and a final output is found. Typically the `Master` node gathers all results from workers and assembles them in the correct order before outputting the results somewhere.

I implemented this using the Universities Hadoop Cluster and the MPJ Express framework for developing MPI programs using the Java programming language. The `Master` node in my program only needs to send the workers a start number, telling it where to begin their range of work because the size of the range of numbers for them to compute is constant. Furthermore, the same array, `buffer`, is used to send the start number of the range to `Workers` and results of the range to `Master`, this makes it much easier to pass data because MPJ send and receive methods require you to pass the data size as an argument. Therefore, if every MPI communication carries an array that is the same size then there will be no errors. The code for this in shown in Figure 21. The array `buffer` is designed so that `buffer[0]` always holds the range start number and `buffer[1 - 14]` hold the results from the `Workers`. The main block of work that each `Worker` computes is the same as the sequential version.



```
final static int RANGE_SIZE = 15 ;
final static int BUFFER_SIZE = RANGE_SIZE + 1 ;
int [] buffer = new int [BUFFER_SIZE];
```

Fig. 21   Snippet of code from 'MPJPrimeSieve.Java'

Analysis of Results

To ensure reliability and fairness in the benchmarking the two algorithms must be run on the same machine. This is because the hardware differences between machines will affect the execution times and cause our results to be much less accurate and reliable. Therefore, all benchmarking has been performed on the Hadoop cluster, even the sequential version. The Universities Hadoop Cluster has 5 machines capable of running the MPJ daemon program and each has an Intel i7 processor with 12 virtual cores (6 physical). This means a total of 4 possible workers and 48 virtual cores to be used for the MPJ task farm. Finally results are for finding primes up to 2,000,000 so that it is sufficiently large enough for it to be worthwhile using parallel programming.

Results

As with the exercises in the labs, the programs were run 10 times and the fastest one was used as the sequential speed in benchmark calculations. It was found through trial and error that the equilibrium between spreading the workload over more processes and having an increase in communication overheads is approximately 24 processes. It was run with 24 processes in all tests.

N = 2,000,000

Run	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
Time (ms)	648	632	652	666	646	642	672	653	640	647

Fig. 22 Sequential prime number sieve results

Run	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
Time (ms)	552	598	552	350	329	665	332	342	453	341

Fig. 23 Parallel prime number sieve results

Figure 22 and 23 shows that the fastest time for the sequential prime number sieve was 632 milliseconds and the parallel version's fastest time was 329 milliseconds. This results in a parallel speedup of 1.92 which is good that it achieved a parallel speedup but being greater than 2 would be better. Furthermore, with a 1.92 speedup and 48 (4x12) virtual cores, the parallel efficiency was 4%, this is very low.

N = 3,000,000

Run	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
Time (ms)	1115	1066	1066	1088	1081	1086	1067	1060	1060	1049

Fig. 23 Sequential prime number sieve results

Run	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
Time (ms)	689	396	402	400	573	402	644	425	412	415

Fig. 24 Parallel prime number sieve results

When finding prime numbers up to 3 million the fastest sequential time was 1049 milliseconds and the fastest parallel run was 396 milliseconds. Therefore, parallel speedup was 2.65, managing to achieve a better than 2 parallel speedup. The parallel efficiency in this test was 5.5% which is better than before but still not very good.

N = 4,000,000

Run	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
Time (ms)	1531	1547	1534	1541	1566	1526	1536	1520	1529	1537

Fig. 25 Sequential prime number sieve results

Run	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
Time (ms)	463	595	450	746	872	481	824	496	518	624

Fig. 26 Parallel prime number sieve results

When N was equal to 4 million the sequential version’s fastest run was 1520 milliseconds and the parallel version managed to find all primes in 450 milliseconds. This resulted in a parallel speedup of 3.38 and a parallel efficiency of 7%.

Analysis & Improvements

All three tests displayed that the MPI task farm style of parallel computation works well, but for very large problems. If the problem size, in this instance, was smaller than 1,000,000 then parallel speedup was not achieved with any configuration of other settings e.g. numbers of processes or the size of a block of work. Furthermore, the highest parallel efficiency achieved was 7%, meaning that on average the cores in the worker nodes are idle for 93% of the time. The largest bottleneck in the execution time of the MPI task farm prime sieve is the time associated with communications between `Master` and `Worker` nodes. Therefore, reducing the number of communications should achieve a higher speedup and efficiency.

During these experiments it was found that having a method which altered the `RANGE_SIZE` variable based on the problem size made the results much faster by reducing communication overheads for very large problem sizes. Larger `RANGE_SIZE` means less blocks of work which means less communications between master and worker nodes. This change was implemented and produced better results. With a problem size of 4 million the parallel version got a fastest run of 314 milliseconds meaning a best parallel speedup of 4.84 and 10% efficiency. Unfortunately, the improvements were not enough to make a parallel speedup occur at less than when N = 1,000,000.

Extra: Eratosthenes Sieve

The algorithm

Eratosthenes Sieve is an algorithm for a prime number sieve invented by Ancient Greek Polymath Eratosthenes of Cyrene sometime in the 3rd century BCE. The algorithm works by multiplying every number from 2 to N, where $N =$ the number you want to find primes up to, by numbers from 2 until the product of multiplication is greater than N. The product of every multiplication is marked in some way so that you know that all marked numbers are composite and all unmarked numbers are prime.

This is a more efficient algorithm than the prime sieve described in the previous sections because it is based on the multiplication operator which is much faster than the modulo or division operators. However, there are several optimisations that can be to eratosthenes sieve. Firstly, there is no need to find multiples of 2 to N, you only need to multiply of 2 to $\sqrt{N}$, multiples after this will already have been identified.

Parallelisation

To parallelise this algorithm it also ran as an MPI task farm but with a few differences to the previous prime number sieve. There will be many less communications between the `Master` and `Worker` nodes because the workers are only computing for values up to $\sqrt{N}$, therefore less result sets will need to be communicated. However, because every one of those values needs to be multiplied until the product is greater than N you have to communicate the entire N size array of results from `Worker` to `Master`. For very large values of N, say 4,000,000, the communications will hold 4,000,000 values and it will take extremely long to communicate that array.

The architecture that was believed to have the best potential was to decompose the work by individual numbers and have the workers calculate a range of multiples for that one number. For example, `worker 1` would get sent (x, y) and from this it would know to multiply x by every number from y to (y + `RANGE_SIZE`) and `worker 2` would get sent something like (x, z) and do the same. Then the worker would send back to master only the identified numbers and the master would store them in the master primes array. This method would cut down on the size of data communications and make them faster.

However, because MPJ Express requires the size of the array being transmitted to be included as an argument it is essential to know how much data you're attempting to communicate. Therefore, a method to count how many composite numbers are in a range would be required to communicate an array with only the composite numbers or only the primes numbers. Given more time and more access to the Hadoop Cluster a solution could have been developed to this specific problem and the parallelization of this algorithm would have been more sophisticated.

Results

Unfortunately, because only one daemon program can be run on each Hadoop Cluster node at any given time it means only 1 student can be running their programs at a time on a particular node. For days leading up to the deadline, the Hadoop Cluster was completely full all hours of the day, on all nodes. Availability of the cluster was even checked at 2am the night before and it was being used. This meant that I could not test this program as it requires me to be able to use the nodes on the Hadoop Cluster and they were permanently in use.

However, I can time the sequential version of this algorithm and use Amdahl's Law to gain an approximation of the speedup that could be achieved by the MPJ program. Amdahl's Law is defined in Figure 27, where $S_{Latency}$ is the potential speedup, s is the number of threads used in the parallel program and p is the proportion of the program that can be run in parallel.

$$\boxed{S_{\text{latency}}(s) = \frac{1}{(1-p) + \frac{p}{s}}}$$

Fig. 27 Amdahl's Law

Therefore, as the previous prime number sieve, Eratosthenes Sieve would have been run with 24 threads and 100% of the main work load would have been executed as an MPI task farm.

This gives: $S_{Latency}(24) = \dfrac{1}{(1-1) + \frac{1}{24}} = 24$, meaning a potential parallel speedup of

24, which would be very impressive. Unfortunately, when timing the sequential version the fastest time it could achieve over 12 runs and finding primes up to 1,000,000 was 289 milliseconds. This is slower than my original sequential prime sieve and does not lead me to believe that the parallel version would achieve anything close to a 24 times parallel speedup.

Conclusions

Lab exercises were completed every week that used the Java threads library to teach students how to decompose problems and to start changing how we think about sharing workloads in a parallel problem. Later, the labs taught more sophisticated, distributed memory parallelism through the use of MPI and MPJ Express to develop MPI programs in Java. This was extremely helpful for my development project. Finally, the labs touch on GPU programming and GPU leveraging, through Aparapi, to gain access to massive parallelism due to the thousands of cores in a GPU.

Putting the knowledge and skills that had learnt in the labs to use for the development project was also an opportunity to learn more about the intricacies and expand my understanding of developing projects with MPJ Express and using the Hadoop Cluster. There were many problems to overcome and the lessons learnt from overcoming one challenge would lead to ideas and optimisations of the MPI programs that would have otherwise been unthought of. I was able to produce an MPI task farm prime number sieve that reached a best parallel speedup of 4.84 which is really good, however considering it was spread over 5 machines and 48 cores the parallel efficiency could have been better. All things considered, I feel it was a successful project especially due to the difficulties of the Coronavirus and not being able to go to University buildings as often. On top of that there was a huge amount of difficulty in actually being able to use the resources required to benchmark Eratosthenes sieve because everyone was using the Hadoop Cluster leading up to the deadline.

Given more time, I would explore additional ways to reduce the number of communications between nodes as this is the main factor in slowing down the MPI task farm. Also the arrays being communicated between nodes are carrying all a lot of unnecessary data, data already determined to be composite, and they have to carry this data because the node receiving the array is expecting that array to be an exact, prescribed size. If the node receiving the array could accept an array of any size or be informed in advance of the incoming array size then the communication times could be greatly reduced as much less data would be in each communication.