Parallel Barnes-Hut Simulation

By

Alec Rusch

Project unit: PJE40
Supervisor: Jiacheng Tan

May 2019

Contents	**Page 2**
Abstract	**Page 3**
Introduction	**Page 3**
Project Idea	**Page 3**
Project Outline	**Page 3**
Software	**Page 3**
Project Overview	**Page 3**
Project Aims	**Page 3**
Research	**Page 3**
Methodology	**Page 3**
Requirements & Analysis	**Page 4**
Risk Assessment	**Page 4**
Task Breakdown	**Page 4**
Backup strategies and version control	**Page 4**
Software selection	**Page 5**
Testing strategy	**Page 5**
Design	**Page 5**
Pseudocode	**Page 5**
Implementation	**Page 5**
Testing	**Page 9**
Analysis	**Page 10**
Evaluation	**Page 14**
Comparison to PID	**Page 14**
Conclusion	**Page 15**
References	**Page 15**
Appendix	**Page 15**

Abstract (word count: 2374)
The Barnes-Hut simulation is an n-body simulation. In this project, it will be parallelized in order to achieve a speedup. This will be carried out using Java and the MPJ framework.

Introduction
Parallelisation of the Barnes-Hut simulation requires an understanding of how the simulation runs, and what approach to parallelisation is most suitable. Project code and related files can be found on https://github.com/alec-rusch3/PJE40

Project Idea
This paper will determine the best approach to parallelizing the Barnes-Hut simulation, analysing the performance difference between various methods and giving a conclusion on what approach achieves the best speedup.

Project Outline
In this project, parallelisation will be attempted using 2 different approaches; Java threads and MPI. Java threads are part of the Java language and can be accessed by extending the class to support threads. Message passing interface (MPI) requires the use of a framework, in this project MPJ Express will be used. These 2 approaches will be benchmarked, analysed and compared with each other and the sequential code, doing so will allow a conclusion to be reached on the optimal approach.

Software
Java8 will be the programming language used, with NetBeans 8.1 as the editor. MPJ Express will provide support for MPI.

Problem Overview
The project problem is parallelising a sequential version of the Barnes-Hut simulation.

Project Aims
The aim of this project is to achieve the highest speedup possible with the available equipment.

Research
In order to complete this project, parallelisation strategies had to be researched. This meant learning how to parallelise programs in Java, using both threads and MPJ. The majority of the project research focused on this area, but research was also conducted into the Barnes-Hut simulation, to see different approaches to parallelisation that have been attempted. During this research a parallelisation of Barnes-Hut using MPJ was not found however, there was many other approaches in different languages and different frameworks that offered some insight into how to go about parallelising the code.

Methodology
For this project, the agile software development model was used. “Agile projects are iterative insofar as they intentionally allow for "repeating" software development activities” (What is Agile Software Development?, 2019). This development model is well suited to this project, as there will be numerous changed to the code throughout the project making an iterative approach favourable.

Requirements & Analysis

Risk Assessment

This project faces numerous risks, the most prominent being time. In the event of running out of time, I should be aware in advance if this is likely to happen and can make the necessary adjustments to compensate. Some adjustments that could be made include cutting back the project to its minimum requirements, which should give me the time needed to complete the project but will result in a reduced speedup, so should be avoided if possible.

Another risk is bugs in the programming, bugs will lead to slowdowns in the project development depending on the difficulty to solve them. Vigorous testing should identify and resolve most bugs. Unresolved bugs could lead to slower than expected speedups, and the program not working as intended.

Task Breakdown

The following list identifies the tasks throughout the project:

1. Setup
    a. Install Java8
    b. Install Netbeans IDE 8.1
    c. Install MPJ Express
2. Coding
    a. Find or create sequential code
    b. Benchmark sequential code
    c. Research parallelisation
    d. Parallelise sequential code
    e. Benchmark parallel code
    f. Research MPI implementation
    g. Implement MPI into code
    h. Benchmark MPI code
3. Testing
    a. Check outputs of code for any unexpected results
    b. In the event of unexpected results resolve issues found
4. Analysis
    a. Analyse the benchmark results
    b. Parallel speedup and efficiency
5. Evaluation
    a. Further ways to improve the project
    b. Explain the analysis
6. Conclusion
    a. State best approach

Backup strategies and version control

All the project deliverables should be backed up regularly, in the event of the latest copy is lost it would be an enormous setback to the project timeline. Backups should not be deleted or overwritten, as it may be useful to analyse previous attempts at parallelisation to compare the performance difference between different versions of the code. Github could be used for both backing up and version control of the code, as well as the documentation. Other

strategies that could be used simultaneously include backing up onto a USB or other physical storage, and google drive for all written documentation.

Software selection
Java was selected as the programming language as it is well suited for parallelisation. MPJ Express was used for adding MPI communication.

Testing strategies
The code will be tested throughout the development process so that bugs are identified immediately. This is the best strategy as purely coding without testing will make it difficult to identify which changes led to a bug in the code.

Design
To start with the sequential code to be used was decided upon. The version used was authored by Bryan Carpenter, and the unmodified version of this can be found in the aforementioned GitHub repository.

Pseudocode for parallelisation
The basic structure for parallelising the code was first done in pseudo code, before being attempted in java.

```
threadNum = x

For threadNum
        thread.start();
For threadNum
        thread.join();

run()
        calculate which stars the thread will work on
        code for program execution

synch()
        code for synchronising threads
```

As shown in the above code, the first step is to create a variable for the number of threads, then a for loop will be used to start every thread, then join them. The run method will take the non-initialisation code from the main method, and will require new variables for threads calculating which stars to work on. A synchronisation method will also be required so that threads don't get ahead of each other, this could lead to an incorrect output, slowdowns, or the code failing to run.

Implementation
To start with the sequential code was modified so that it could be benchmarked, the following code shows the changes made:

```
int iter = 0;
long startTime = System.currentTimeMillis();
while (iter <= 4000) {
long endTime = System.currentTimeMillis();
System.out.println("Calculation completed in " + (endTime - startTime) + " milliseconds");
```

The new variables iter, startTime and endTime were created. Iter is used as a stopping condition for the program so that it doesn't run forever, and startTime and endTime are used to record the number of milliseconds the program took to complete.
With this code added the following benchmark was recorded:

Program	Results					Best
Sequential	31017	32081	29373	31932	29946	29373

After the sequential code was benchmarked development on the parallel version took place. To parallelise the code the variables covered in the pseudo code were added, in addition to the functions covered in the pseudo code some other changes were also made to prevent errors, as covered in the table below.

Added/Modified code	Purpose

`import java.util.concurrent.CyclicBarrier;`
 	
`public class BarnesHutParallel extends Thread {`
	Imports cyclic barriers and adds thread support.

`int me;`
`static CyclicBarrier barrier = new CyclicBarrier(P);`
`final static int B = N / P;`
	Declares the variables me, barrier and B. Me is the threadID, barrier is for synchronisation, and B is for allocating stars.

`BarnesHutParallel[] threads = new BarnesHutParallel[P];`
`for (int me = 0; me < P; me++) {`
`  threads[me] = new BarnesHutParallel(me);`
`  threads[me].start();`
`}`
 	
`for (int me = 0; me < P; me++) {`
`  threads[me].join();`
`}`
	Creates the threads, starts them and joins them.

`BarnesHutParallel(int me) {`
`  this.me = me;`
`}`
	Used for initialising each thread.

`public void run() {`
 	
`  int begin = me * B;`
`  int end = begin + B;`
	Assigns the stars that each thread will work on, uses the variables “B” created at start.

`if (me == 0 && iter % OUTPUT_FREQ == 0) {`
`  System.out.println("iter = " + iter + ", time = " + iter * DT);`
`  display.repaint();`
`}`
	Only thread 0 updates the screen and prints iteration.

`for (int i = begin; i < end; i++) {`	i starts at begin and ends at end so that threads only work on the stars they've been allocated.
`computeAccelerations(begin, end, me);` `synch();`	Passes begin, end & me to computeAccelerations then synchs the threads afterwards.

`static void computeAccelerations(int begin, int end, int me) {`
`//for (int i = begin ; i < end ; i++)`
`if (me == 0) {`
`tree = new Node(0.0, BOX_WIDTH, 0.0, BOX_WIDTH, 0.0, BOX_WIDTH);`
`for (int i = 0; i < N; i++) {`
`tree.addParticle(x[i], y[i], z[i]);`
`}`
`tree.preCompute();`
`}`
`synch();`
	Only thread 0 calculates the tree, syncs afterwards, otherwise programs crashes.

`static void synch() {`
`try {`
`barrier.await();`
`} catch (Exception e) {`
`e.printStackTrace();`
`System.exit(1);`
`}`
`}`
	Method for synchronisations of threads. Threads wait until all threads reach this stage.

Once all the necessary changes were made, the code was benchmarked and compared to the sequential code to calculate the speedup and efficiencies, as shown in the table below:

Program	Results					Best	Speedup	Efficiency
Sequential	31017	32081	29373	31932	29946	29373	n/a	n/a
Parallel 1 thread	31006	30236	29736	29467	29800	30192	0.9728736089	0.9728736089
Parallel 2 threads	16818	16595	17936	16000	15889	15889	1.8486374221	0.9243187111
Parallel 4 threads	15299	14717	15496	15372	14936	14717	1.9958551335	0.4989637834
Parallel 8 threads	12370	13499	13489	12697	13614	12370	2.3745351657	0.2968168957
Parallel 16 threads	11993	11528	11810	12442	11659	11528	2.5479701596	0.1592481350

The formula used for speedup and efficiency:
speedup = sequential time / parallel time
efficiency = speedup / number of threads

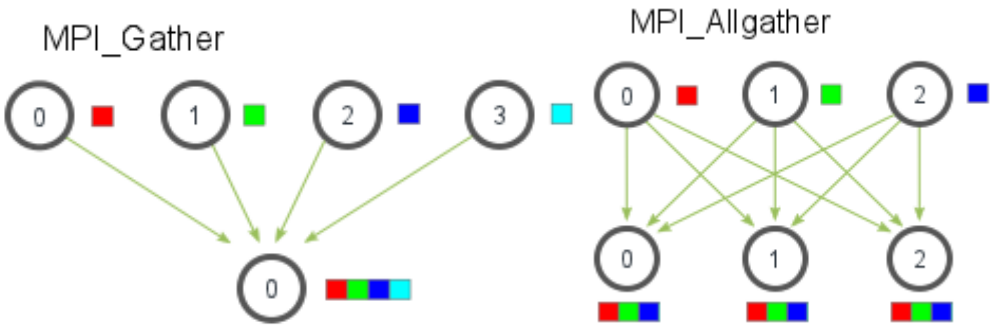
Observations

2 threads have a large jump in speedup compared to 1 thread, but the performance improvement from adding more than 2 threads is inefficient as shown in the table above.

2 threads achieved the most efficient speedup, 4 and more threads continued to improve on the timings but the efficiency decreased significantly.

MPJ Barnes-Hut

For the MPJ implementation of Barnes-Hut, a different approach to parallelisation has to be taken. Since MPI can work over multiple machines instead of just locally, it requires code for communication, something not needed in the previous parallelisation. To start with the AllGather function was used for communication, this sends the data in the buffer to all threads working on the problem. However, after further testing, the Gather function was used instead, as it reduces the amount of communication without impacting the output. Gather is a synchronous operation, so the use of barriers is not required. The following diagram (K, Wes. n.d.) illustrates how Gather works compared to AllGather.



K, Wes states that “Gather takes elements from each process and gathers them to the root process.” Gather is better suited to this project, as the simulation is only being drawn on the machine of the root process, so it is unnecessary and inefficient to send all the elements to all the processes.

As well as the communication function the following code had to be added:

Added/Modified code	Purpose
`import mpi.*;`	imports mpi

`static Display display;`
`static int P, me, B;`
 	
`public static void main(String args[]) throws Exception {`
`  MPI.Init(args);`
 	
`  me = MPI.COMM_WORLD.Rank();`
`  P = MPI.COMM_WORLD.Size();`
`  B = N / P;`
`  if (me == 0) {`
`  display = new Display();`
`  }`
	The display is only created on one machine, whichever one has its ip first in the machines file.

`if (me == 0) {`
`  display.repaint();`
`}`
	Only thread 0 needs to repaint since the display is only drawn on its machine.

`double[] xLoc = new double[B];`
`double[] yLoc = new double[B];`
`double[] zLoc = new double[B];`
`for (int i = 0; i < B; i++) {`
`  xLoc[i] = x[begin + i];`
`  yLoc[i] = y[begin + i];`
`  zLoc[i] = z[begin + i];`
`}`
`MPI.COMM_WORLD.Gather(xLoc, 0, B, MPI.DOUBLE, x, 0, B, MPI.DOUBLE, 0);`
`MPI.COMM_WORLD.Gather(yLoc, 0, B, MPI.DOUBLE, y, 0, B, MPI.DOUBLE, 0);`
`MPI.COMM_WORLD.Gather(zLoc, 0, B, MPI.DOUBLE, z, 0, B, MPI.DOUBLE, 0);`
	This code was added at the start of computeAcceleration. Gather used to send all

`MPI.Finalize();`	No more instructions will be issued, prevents anymore commands from being issued.

With these changes made the code can be successfully run over multiple machines without error, the following benchmark was done with the final version of the code:

np	1 pc					2 pc					3 pc					4 pc				
1	36126	37625	35797	34858	35848	(np < pc)					(np < pc)					(np < pc)				
2	22426	22147	22727	21110	21018	20943	21064	21139	22297	20513	(np < pc)					(np < pc)				
4	15817	15676	15450	15473	15699	15826	15821	16033	15035	14720	15445	15658	15103	16142	16158	17480	16347	16314	15814	16814
8	17954	17997	18759	16808	17508	10319	11093	10516	10937	10281	11189	11298	11344	11329	11392	11673	11548	11423	11407	11360
16	20410	19500	20406	20609	21481	13266	14250	13936	14581	14895	11708	10906	12281	11469	10675	9688	10266	10332	9734	9334

Observations

When 3 machines were running the code the performance was lower than expected, since the threads were divided up unevenly. MPJ with 1 thread performs worse than parallel with 1 thread and sequential because of the time spent initialising the program. Delay from communication overheads also impacts the performance of MPJ, however as there is more processing power available faster times are achieved. The fastest result was achieved the 4 machines and 16 threads, the maximum amount that was tested in this project.

Test

The program was tested throughout, as called for by the agile development model. All of the final versions of the program work as expected, with various amounts of threads, although all versions have inefficient thread utilization and lower than expected speedup.

Analysis

With all benchmarking completed, the results from all runs were compiled into the following table for comparison:

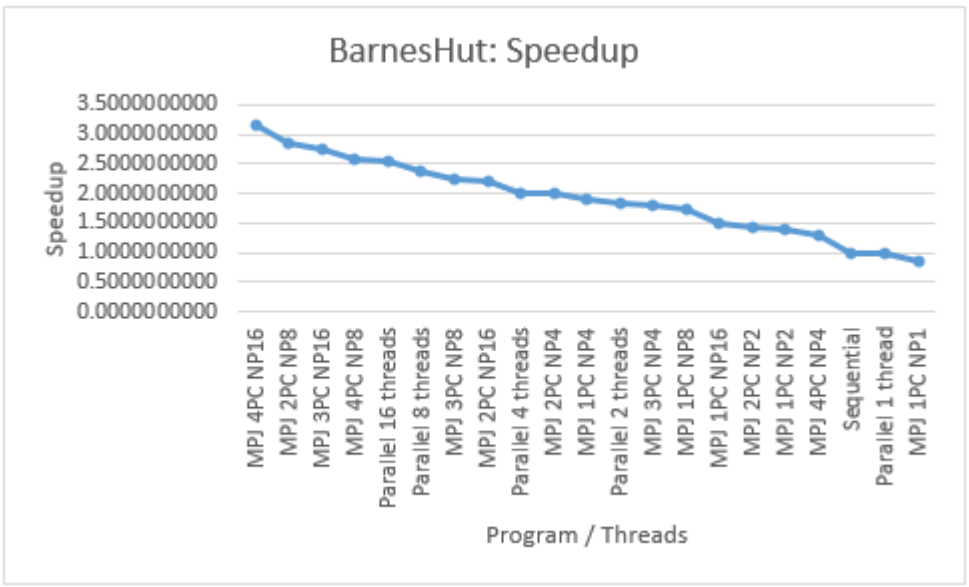
Program	Results					Best	Speedup	Efficiency
Sequential	31017	32081	29373	31932	29946	29373	n/a	n/a
Parallel 1 thread	31006	30236	29736	29467	29800	30192	0.9728736089	0.9728736089
Parallel 2 threads	16818	16595	17936	16000	15889	15889	1.8486374221	0.9243187111
Parallel 4 threads	15299	14717	15496	15372	14936	14717	1.9958551335	0.4989637834
Parallel 8 threads	12370	13499	13489	12697	13614	12370	2.3745351657	0.2968168957
Parallel 16 threads	11993	11528	11810	12442	11659	11528	2.5479701596	0.1592481350
MPJ 1PC NP1	36126	37625	35797	34858	35848	34858	0.8426473120	0.8426473120
MPJ 1PC NP2	22426	22147	22727	21110	21018	21018	1.3975164145	0.6987582073
MPJ 1PC NP4	15817	15676	15450	15473	15699	15450	1.9011650485	0.4752912621
MPJ 1PC NP8	17954	17997	18759	16808	17508	16808	1.7475606854	0.2184450857
MPJ 1PC NP16	20410	19500	20406	20609	21481	19500	1.5063076923	0.0579349112
MPJ 2PC NP2	20943	21064	21139	22297	20513	20513	1.4319212207	0.7159606103
MPJ 2PC NP4	15826	15821	16033	15035	14720	14720	1.9954483696	0.4988620924
MPJ 2PC NP8	10319	11093	10516	10937	10281	10281	2.8570177998	0.3571272250
MPJ 2PC NP16	13266	14250	13936	14581	14895	13266	2.2141564903	0.1383847806
MPJ 3PC NP4	17859	17514	18024	16310	16974	16310	1.8009196812	0.4502299203
MPJ 3PC NP8	14105	14400	14400	14844	13030	13030	2.2542594014	0.2817824252
MPJ 3PC NP16	11708	10906	12281	11469	10675	10675	2.7515690867	0.1719730679
MPJ 4PC NP4	22580	24471	24921	24302	25829	22580	1.3008414526	0.3252103632
MPJ 4PC NP8	11673	11548	11423	11407	11360	11360	2.5856514085	0.3232064261
MPJ 4PC NP16	9688	10266	10332	9734	9334	9334	3.1468823655	0.1966801478

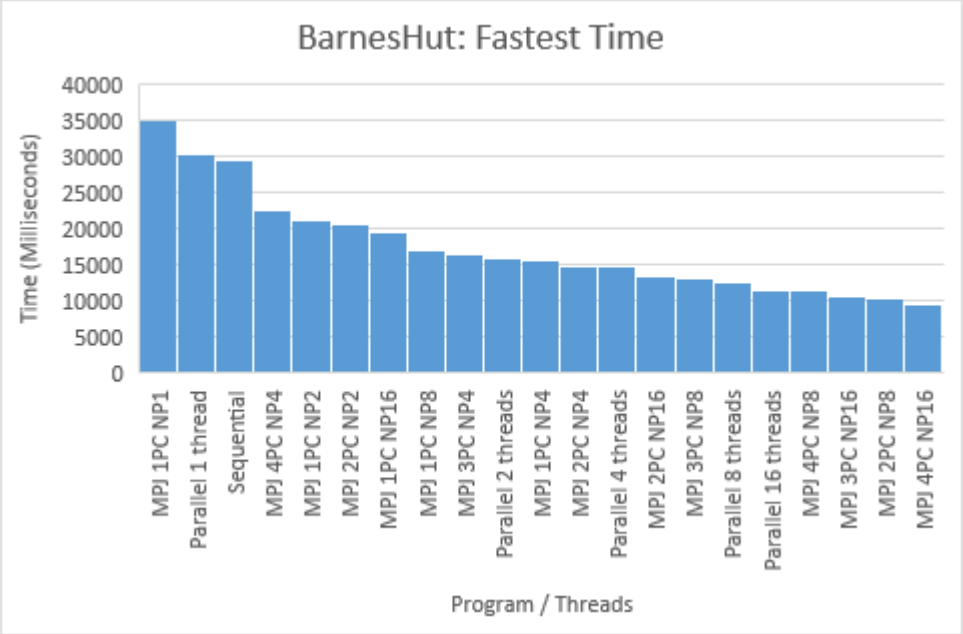
Some immediate observations can be made from the table above, most prominently the low efficiency of all programs, none of them having an efficiency over 1.

For further analysis the table was sorted by speedup:

Program	Results					Best	Speedup	Efficiency
MPJ 4PC NP16	9688	10266	10332	9734	9334	9334	3.1468823655	0.1966801478
MPJ 2PC NP8	10319	11093	10516	10937	10281	10281	2.8570177998	0.3571272250
MPJ 3PC NP16	11708	10906	12281	11469	10675	10675	2.7515690867	0.1719730679
MPJ 4PC NP8	11673	11548	11423	11407	11360	11360	2.5856514085	0.3232064261
Parallel 16 threads	11993	11528	11810	12442	11659	11528	2.5479701596	0.1592481350
Parallel 8 threads	12370	13499	13489	12697	13614	12370	2.3745351657	0.2968168957
MPJ 3PC NP8	14105	14400	14400	14844	13030	13030	2.2542594014	0.2817824252
MPJ 2PC NP16	13266	14250	13936	14581	14895	13266	2.2141564903	0.1383847806
Parallel 4 threads	15299	14717	15496	15372	14936	14717	1.9958551335	0.4989637834
MPJ 2PC NP4	15826	15821	16033	15035	14720	14720	1.9954483696	0.4988620924
MPJ 1PC NP4	15817	15676	15450	15473	15699	15450	1.9011650485	0.4752912621
Parallel 2 threads	16818	16595	17936	16000	15889	15889	1.8486374221	0.9243187111
MPJ 3PC NP4	17859	17514	18024	16310	16974	16310	1.8009196812	0.4502299203
MPJ 1PC NP8	17954	17997	18759	16808	17508	16808	1.7475606854	0.2184450857
MPJ 1PC NP16	20410	19500	20406	20609	21481	19500	1.5063076923	0.0579349112
MPJ 2PC NP2	20943	21064	21139	22297	20513	20513	1.4319212207	0.7159606103
MPJ 1PC NP2	22426	22147	22727	21110	21018	21018	1.3975164145	0.6987582073
MPJ 4PC NP4	22580	24471	24921	24302	25829	22580	1.3008414526	0.3252103632
Sequential	31017	32081	29373	31932	29946	29373	n/a	n/a
Parallel 1 thread	31006	30236	29736	29467	29800	30192	0.9728736089	0.9728736089
MPJ 1PC NP1	36126	37625	35797	34858	35848	34858	0.8426473120	0.8426473120

A graph was plotted using the above table:

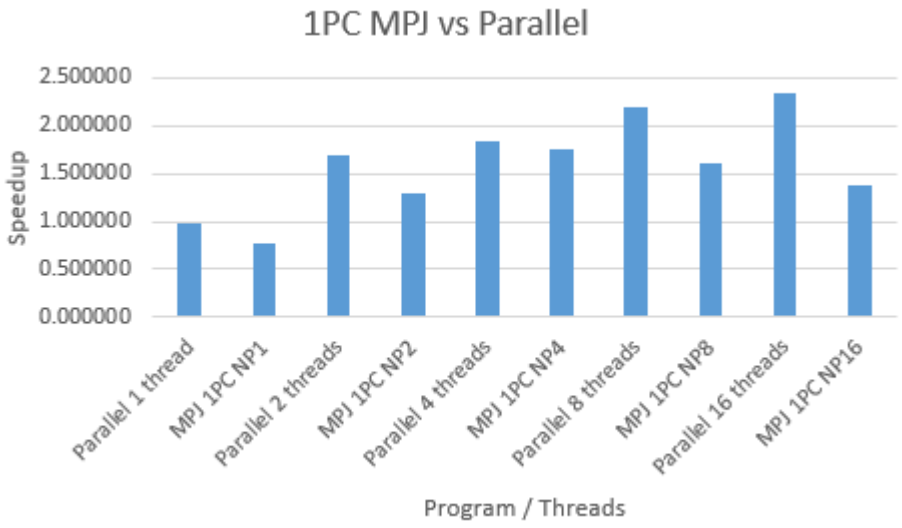




As shown in the graphs and table, the fastest speedup of 3.14 was achieved by the MPJ version using 4 machines and 16 threads. This is to be expected as the machines used were quad-core so they perform best with 4 threads, and in this run, there were 4 machines using 4 threads each.

Another observation that can be made is that MPJ code had the 4 best performing time, suggesting that using more machines is the ideal approach for achieving higher speedups. However, when comparing the MPJ code for 1 pc with parallel, it can be seen that across 1 pc MPJ performs significantly worse, having a 0.96 lower speedup on 16 threads. The table and graph below show this:

Program	Results					Best	Speedup	Efficiency
Sequential	27469	27702	27039	27094	27229	27039	n/a	n/a
Parallel 1 thread	27647	27445	27744	27410	28164	27410	0.986465	0.9864648
MPJ 1PC NP1	36126	37625	35797	34858	35848	34858	0.775690	0.7756899
Parallel 2 threads	16818	16595	17936	16000	15889	15889	1.701743	0.8508717
MPJ 1PC NP2	22426	22147	22727	21110	21018	21018	1.286469	0.6432344
Parallel 4 threads	15299	14717	15496	15372	14936	14717	1.837263	0.4593158
MPJ 1PC NP4	15817	15676	15450	15473	15699	15450	1.750097	0.4375243
Parallel 8 threads	12370	13499	13489	12697	13614	12370	2.185853	0.2732316
MPJ 1PC NP8	17954	17997	18759	16808	17508	16808	1.608698	0.2010873
Parallel 16 threads	11993	11528	11810	12442	11659	11528	2.345507	0.1465942
MPJ 1PC NP16	20410	19500	20406	20609	21481	19500	1.386615	0.0866635



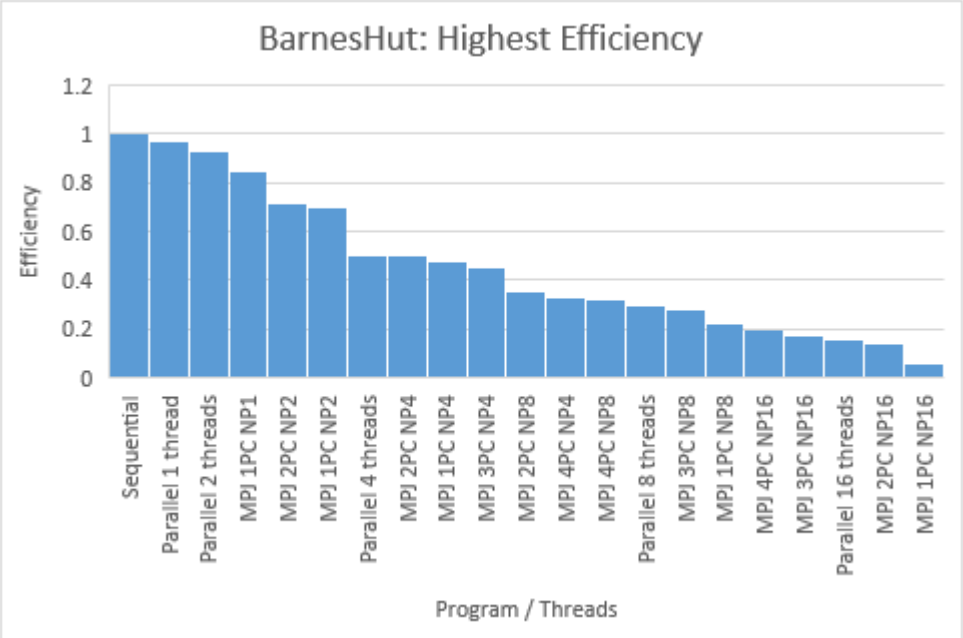
This suggests that a single high specification computer could outperform many low specification computers.

A comparison of program efficiency was also made, as shown in the table below:

Program	Results					Best	Speedup	Efficiency
Sequential	31017	32081	29373	31932	29946	29373	n/a	n/a
Parallel 1 thread	31006	30236	29736	29467	29800	30192	0.9728736089	0.9728736089
Parallel 2 threads	16818	16595	17936	16000	15889	15889	1.8486374221	0.9243187111
MPJ 1PC NP1	36126	37625	35797	34858	35848	34858	0.8426473120	0.8426473120
MPJ 2PC NP2	20943	21064	21139	22297	20513	20513	1.4319212207	0.7159606103
MPJ 1PC NP2	22426	22147	22727	21110	21018	21018	1.3975164145	0.6987582073
Parallel 4 threads	15299	14717	15496	15372	14936	14717	1.9958551335	0.4989637834
MPJ 2PC NP4	15826	15821	16033	15035	14720	14720	1.9954483696	0.4988620924
MPJ 1PC NP4	15817	15676	15450	15473	15699	15450	1.9011650485	0.4752912621
MPJ 3PC NP4	17859	17514	18024	16310	16974	16310	1.8009196812	0.4502299203
MPJ 2PC NP8	10319	11093	10516	10937	10281	10281	2.8570177998	0.3571272250
MPJ 4PC NP4	22580	24471	24921	24302	25829	22580	1.3008414526	0.3252103632
MPJ 4PC NP8	11673	11548	11423	11407	11360	11360	2.5856514085	0.3232064261
Parallel 8 threads	12370	13499	13489	12697	13614	12370	2.3745351657	0.2968168957
MPJ 3PC NP8	14105	14400	14400	14844	13030	13030	2.2542594014	0.2817824252
MPJ 1PC NP8	17954	17997	18759	16808	17508	16808	1.7475606854	0.2184450857
MPJ 4PC NP16	9688	10266	10332	9734	9334	9334	3.1468823655	0.1966801478
MPJ 3PC NP16	11708	10906	12281	11469	10675	10675	2.7515690867	0.1719730679
Parallel 16 threads	11993	11528	11810	12442	11659	11528	2.5479701596	0.1592481350
MPJ 2PC NP16	13266	14250	13936	14581	14895	13266	2.2141564903	0.1383847806
MPJ 1PC NP16	20410	19500	20406	20609	21481	19500	1.5063076923	0.0579349112

As shown in the table above and the graph below none of the modified programs had higher efficiency than the default sequential code. This is because in the MPJ version the star locations have to be broadcast every iteration, slowing down the program with the additional communication cost. Another cause is the sequential code that was modified is not well

suited to parallelism, it needs to be altered so that the stars are handled as a group instead of individually, the process for achieving this is discussed in the evaluation.



Evaluation

The program efficiency was fairly poor, and the speedups achieved were lower than expected. The highest speedup achieved was 3.41, with 16 threads over 4 pcs. An ideal solution would have a speedup that also maintained a similar efficiency to the sequential code. The code that came closest to achieving this was parallel with two threads, which had a speedup of 1.84 and an efficiency of 0.924. In terms of the speed gain per the number of threads, no other runs came close. Potential improvements that could resolve this are discussed below.

One improvement that could be made is modifying the code so that stars are stored in a list, this would improve the speed of the program as it would be no longer be necessary to broadcast the updated star positions each iteration. Another way to improve the code would be to run the MPJ code into hybdev mode, which runs the program in a hybrid between multicore and cluster. This was considered as an option in the course of developing this project but was deemed too complex. With a greater time allowance, this would be a suitable area of investigation for achieving faster speedups.

An additional improvement that could be made to the testing process is to run the MPJ code over a larger number of machines. In the course of this project only up to 4 pcs was tested, if further work was to be carried out a larger number of pcs could be used, or pcs with superior hardware could also be used to see what effect adding more computational power has on the speedup.

Comparison to PID

Looking back at the PID, it can be stated that the goal of parallelising the Barnes-Hut simulation was achieved, but not as successful as hoped for. With further time spent on the project, a greater speedup is possible.

Conclusion
Parallelism was successfully implemented however, the speedup was less than expected. Throughout the development of this project, my knowledge of parallelism approaches and implementations has improved, as well as my approach to debugging because this was a crucial part of identifying and resolving errors in the project. Analysing the results allowed me to estimate and measure the potential and actual performance improvements that occur, in order to determine whether the best approach to parallelising the problem had been made.

References
What is Agile Software Development?. (2019). Retrieved from https://www.agilealliance.org/agile101/

Kendall, W. (n.d.). MPI Scatter, Gather, and Allgather. Retrieved from: http://mpitutorial.com/tutorials/mpi-scatter-gather-and-allgather/

Appendix
Appendix A (PID)

Project Initiation Document

1. Basic details

Student name:	Alec Rusch
Draft project title:	Parallel Barnes-Hut Simulation
Course:	BSc (Hons) Computing
Client organisation:	
Client contact name:	
Project supervisor:	Dr. Jiacheng Tan

Project outline, aims and objectives.

The aim of the project is to parallelize the Barnes-Hut galaxy simulation.

Deliverables

The project will have the following deliverables; requirements, design, test strategy, testing results, program code, and report.

The requirements documentation will state the changes to be made in order to parallelize the code. The requirements are likely to change throughout the project so I expect this will be updated multiple times.

The initial design of how the program should work; the different classes and functions involved. This is also likely to change during the programming stage.

There will be a test plan detailing all the tests that need to be carried out and the expected result of each test; this can be compared with documentation demonstrating the result of each test, confirming whether the tests results were as expected or if issues were found and whether they were resolved or not.

The code will be documented, explaining how different functions contribute to a speedup.

Constraints

The main projects constraints are time and my own ability.

Another constraint is my own ability; if I struggle with any part of the project I am likely to fall behind schedule and this could have a sizeable impact on the final deliverables.

Approach

The first stage of approaching this project is deciding upon the hardware and software to be used. It has already been decided that the hardware will be university computers, in order to get a fair comparison when benchmarking over multiple machines. The decision on the software I use will be made after the research stage.

During the programming stage I will need to decide on the different approaches to parallelizing the code. Approaches which offer an improved speedup may be more difficult to implement, and therefore not time effective to attempt.

Facilities and resources

I will use java for coding, I have previous experience with java but not related to parallelization so will need to learn some new programming techniques in order to complete the project.

There should be an abundance of resources online, so as long as I understand them I should be able to overcome must problems.

Risks

One risk is bugs in my programming, bugs will lead to slowdowns depending on the difficulty to solve them. Vigorous testing should identify and resolve most bugs.

Running out of time is another risk; I should know in advance if this is likely to happen and can make the necessary adjustments to compensate. Some adjustments that could be made include cutting back the project to its minimum requirements, which should give me the time needed to complete the project but will result in a reduced speedup, so should be avoided if possible.

Starting point for research

The first thing to look at is existing technology and projects carried out by other people. This is to get an idea of what direction my project could take.

Once I get to the programming part of the project, I will need to research solutions to any parts I get stuck on.

Tasks

Research
- Look into existing approaches
- Decide upon frameworks to be used

Hardware
- Same computers used for benchmarking throughout, or results unreliable

Programming
- Start coding different functions

Testing
- Testing should be carried out throughout the programming stage to identify bugs and performance improvements/losses as soon as possible.

Documentation
- I will document the development process for use in my report.

Report
- The final submission will be worked on throughout the project, updated regularly as tasks are completed and an evaluation will be written at the end.

Legal, ethical, professional, social issues

No legal issues should arise during the project; I shouldn't be breaking any laws during this project. For professional issues there should not be breaking any code of conducts. There are no ethical or social issues.

Project plan

Start by modifying the sequential code so it can be benchmarked, this will be achieved by adding a timer. As parallel code is developed record times to determine what approaches offer the best speedup. Once sequential and parallelized code benchmarked use MPJ framework to run code over multiple machines, then as before benchmark and analyse time improvements.

Appendix B (Ethics Review)

Certificate of Ethics Review

Project Title: Parallel Barnes-Hut simulation

Name: Alec Rusch **User ID:** 825804 **Application Date:** 09-May-2019 16:30

ER Number: ETHIC-2019-593 You must download your referral certificate, print a copy and keep it as a record of this review.

The FEC representative for the School of Computing is Carl Adams
It is your responsibility to follow the University Code of Practice on Ethical Standards and any Department/School or professional guidelines in the conduct of your study including relevant guidelines regarding health and safety of researchers including the following

- University Policy
- Safety on Geological Fieldwork

It is also your responsibility to follow University guidance on Data Protection Policy:

- General guidance for al data protection issues
- University Data Protection Policy

Which school/department do you belong to?: **SOC**
What is your primary role at the University?: **UndergraduateStudent**
What is the name of the member of staff who is responsible for supervising your project?: **Jiacheng Tan**
Is the study likely to involve human subjects (observation) or participants?: **No**
Are there risks of significant damage to physical and/or ecological environmental features?: **No**
Are there risks of significant damage to features of historical or cultural heritage (e.g. impacts of study techniques, taking of samples)?: **No** Does the project involve animals in any way?: **No**
Could the research outputs potentially be harmful to third parties?: **No** Could your research/artefact be adapted and be misused?: **No**

Does your project or project deliverable have any security implications?: **No** Please read and confirmthat you agree with the fo**l**owing statements: **Confirmed** Please read and confirmthat you agree with the fo**l**owing statements: **Confirmed** Please read and confirmthat you agree with the fo**l**owing statements: **Confirmed**

Supervisor Review

As supervisor, I will ensure that this work will be conducted in an ethical manner in line with the University

Supervisor **Date**