Final Report
Mary Chrishani Jayaweera
Magnus Sahlin

Contents

1 Introduction **2**
1.1 Theory 2

2 Implementation **2**
2.1 Plummer Spheres 2
2.2 Direct-sum Implementation 3
2.3 Barnes-Hut Implementation 3
2.4 Parallelization 4
2.4.1 PThreads 5
2.4.2 OpenMP 5

3 Performance **5**
3.1 Direct-Sum Implementation 5
3.2 Barnes-Hut Implementation 6
3.3 Parallelization 7
3.3.1 PThreads 8
3.3.2 OpenMP 9

4 Optimization **10**
4.1 Compiler Optimization Flags 10
4.2 Serial Optimizations 10
4.3 Further Parallelization 11

5 Conclusions **11**

1 Introduction

The aim of this report is to build and optimize a program that can simulate the forces between masses and thereby simulate the movement of stars and planets. This was first done with the simple Plummer Model, and then using the Barnes-Hut algorithm. The complexity of each implementation was then evaluated.

1.1 Theory

Simulating the galaxy is an n-body problem. The governing equation include the Newton's law of gravitation for force calculations.

$$\mathbf{f_{ij}} = \frac{Gm_i m_j}{\mathbf{r}_{ij}^3}\mathbf{r_{ij}} = \frac{Gm_i m_j}{\mathbf{r}_{ij}^3}\hat{\mathbf{r}}_{ij} \tag{1}$$

where G is the gravitational constant, m_i and m_j are masses of the particles and $\mathbf{r}_{ij}$ is the position of particle i relative to particle j.

The force on one star can be calculated as follows:

$$\mathbf{F_i} = -Gm_i \sum_{j=0, j\neq i}^{N-1} \frac{m_j}{\mathbf{r}_{ij}^2}\hat{\mathbf{r}}_{ij} \tag{2}$$

After the forces are calculated the acceleration and velocity can be determined.

$$\mathbf{a}_t = \frac{\mathbf{F}}{m} \tag{3}$$

Choosing a time step dt, the velocity and position change can also be determined using the sympletic Euler Method.

$$\mathbf{v}_{t+dt} = \mathbf{v}_t + \mathbf{a}_t \cdot dt \tag{4}$$

$$\mathbf{p}_{t+dt} = \mathbf{p}_t + \mathbf{v}_{t+dt} \cdot dt \tag{5}$$

2 Implementation

2.1 Plummer Spheres

The particles or the planets and the stars in the simulation in reality have a certain extension in space and are not infinitely small points. To account for this and the instability induced

in the simulation when the denominator in equation 6 goes to zero, we add a constant ϵ to the denominator. What this does in practice is setting an upper limit to the magnitude of the force. This constant is set to 0.001.

$$\mathbf{F_i} = -Gm_i \sum_{j=0, j\neq i}^{N-1} \frac{m_j}{\mathbf{r}_{ij}^2 + \epsilon} \hat{\mathbf{r}}_{ij} \tag{6}$$

2.2 Direct-sum Implementation

The simplest implementation is calculating the force between each and every star separately. This is in theory also the most accurate and most performance heavy implementation with n^2 efficiency. For a simulation with 10 stars, each star will have 9 forces acting on it for a total of $10 \cdot 9 = 90$ forces in the simulation. If we would double the number of stars, then every star will have twice as many forces acting on it, but the total number of forces in the simulation have quadrupled, $20 \cdot 19 = 380$. This means that the complexity of the implementation is $\mathcal{O}(n^2)$

One easy optimization is recognizing that each force actually have a duplicate but opposite force from the other star. Implementing that these "twin forces" are only calculated once will in theory make the code twice as fast. However it will not change the complexity of the implementation.

2.3 Barnes-Hut Implementation

The Barnes-Hut algorithm is implemented by creating a quad tree containing the stars and is faster with $nlog(n)$ time consumption. Each branch contains four children, and each leaf in the tree is either empty or contains one star. The mass and center of mass for the quads is calculated in each time step of the simulation.

When a new star is added it traverses the tree structure until it finds a leaf. If the leaf already contains a star the leaf subdivides and becomes a branch, and both stars is are added to its children.

In figure 1 we can see graphically how the quad tree is built from the stars.

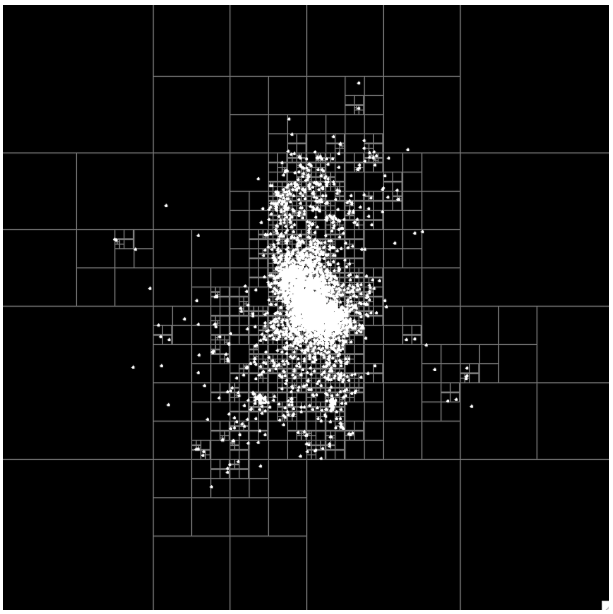

Figure 1: Simulation of 5000 stars

Now when the forces are being calculated it takes the combined mass already calculated in the quad instead of for each star contained in the quad if criteria 7 is met.

$$\theta = \frac{\text{width of current box containing particles}}{\text{distance from particle to center of box}} \leq \theta_{\max} \tag{7}$$

Where $\theta_{\max}$ is a number between 0 and 1 that determines how accurate the simulation will be. Running the provided simulation of 2000 stars 200 time steps showed that $\theta = 0.21$ consistently gets an error less than 0.001, and $\theta = 0.22$ consistently gets an error higher than 0.001. We have therefore decided to use $\theta = 0.21$ for our implementation.

2.4 Parallelization

Parallelization is implemeted by running time consuming parts of the code on multiple cores or threads. After running profiling with macOS Activity Monitor, we could conclude that the function calculating forces answered for roughly 93% of the used computational resources, when running a simulation of 2000 stars. See figure 2. We therefore decided to implement multithreading for that function.

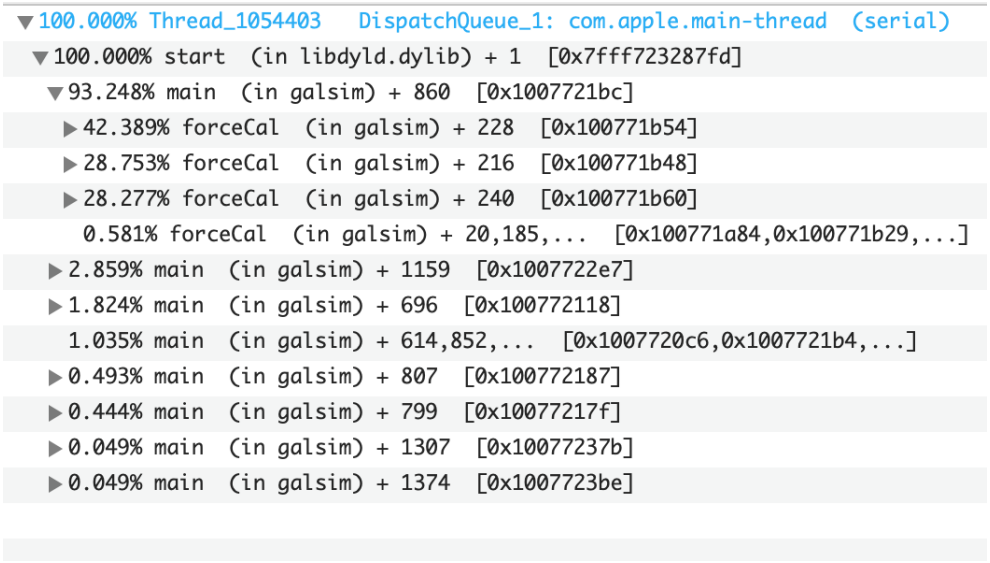

Figure 2: Profiling in macOS Activity Montior with one single thread

2.4.1 PThreads

The multithreading with pthreads works by giving each thread a number of stars to calculate the forces on. If the number of stars is evenly divisible with the number of threads, then each thread will have the same number of stars. Else the last thread will have the additional residual of the division. For example with 2000 stars and three threads, the first two threads will have 666 stars and the last will have 668. This way of dividing the work also ensures that program doesn't crash when the user requests more threads than there are stars in the simulation.

2.4.2 OpenMP

With OpenMP we decided to run the for-loop that calls the function that calculates the force for each star in parallel. This was easily implemented with a single line of code above the for-loop. We can be sure that the loop is safe to run in parallel since no actual updates are made to the stars in terms of position yet, and the forces are not dependant on anything else than the positions.

3 Performance

3.1 Direct-Sum Implementation

Performance was evaluated by measuring the time of a set number of time steps but with different amount of stars in the simulation.

In figure 3 we can see that as N increases, the time increases squared. This is according to the specification in the assignment. For small N the pattern is not apparent, since most of the computing time is not directed towards the actual simulation, but rather memory accesses and other initialization things.

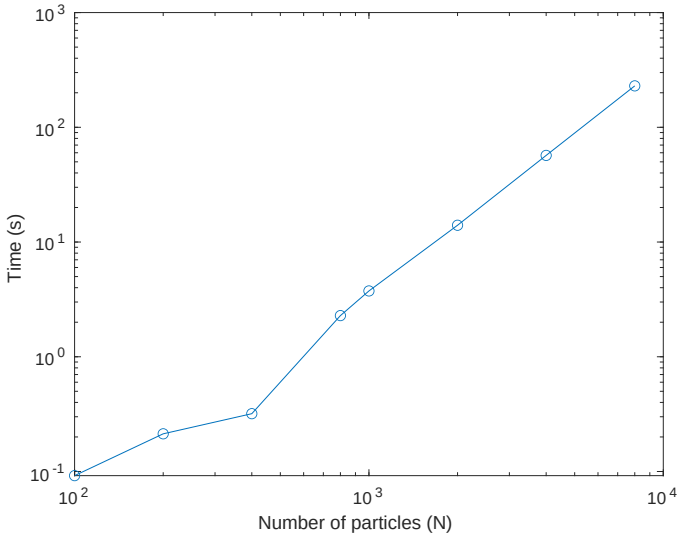

Figure 3: Time for 200 iterations with different N

3.2 Barnes-Hut Implementation

Performance was evaluated by measuring the time of 200 steps with different amount of stars in the simulation with the Unix time command. All runs were done with $\theta_{\max} = 0.21$.

In figure 4 we can see that the time has improved significantly from the direct sum implementation. We also included a model of the time consumption as $t = c \cdot N \cdot log(N)$ and compare to the data. The constant c was chosen as 0.00107 to match the final measurement with $N = 8000$.

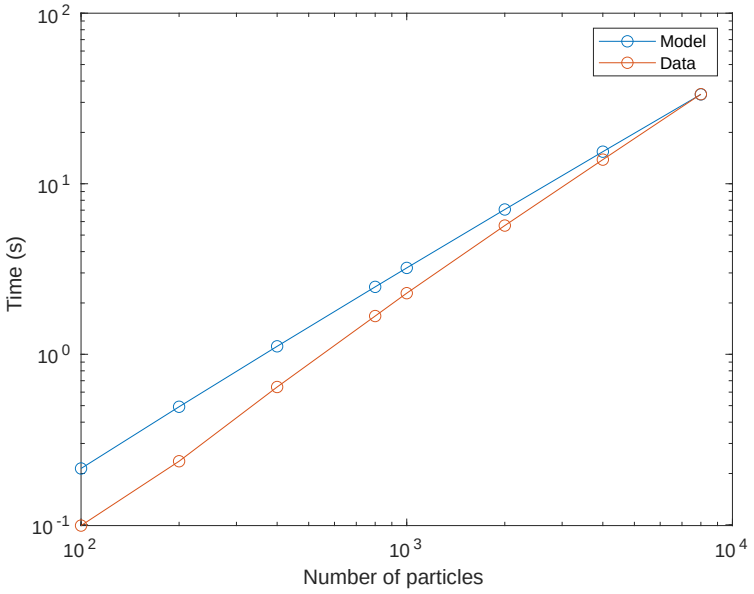

Figure 4: Complexity

3.3 Parallelization

In figure 5 we see the time for different number of particles run for 200 iterations on Vitsippa with 14 threads. Comparing these numbers to previous measurements we notice that parallelization is significantly faster when using both PThreads and OpenMP.

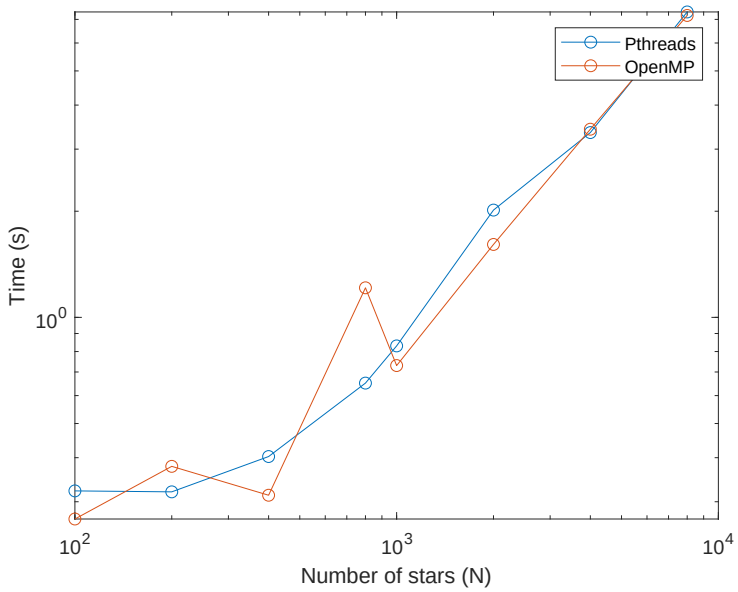

Figure 5: Time for 200 iterations with different number particles N for 14 threads for PThreads and OpenMP run on Vitsippa

3.3.1 PThreads

In figure 6 we can see the speedup of simulations with 2000 and 8000 stars run for 200 iterations for different number of threads. We notice that for two threads the speedup is roughly two, and for three it is (very) roughly three. This pattern continues up until about ten threads, when the speedup becomes less consistent and not as ideal. The pattern is more apparent for 8000 stars than for 2000, this is because for longer simulations the measurements are less sensitive to noise.

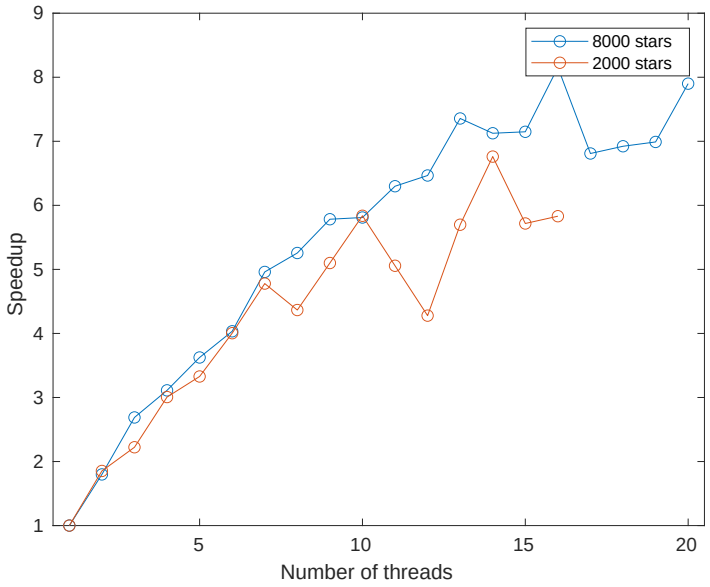

Figure 6: 2000 and 8000 stars run for 200 iterations on Vitsippa with different number of threads

3.3.2 OpenMP

We make the same observations for OpenMP as for PThreads. The simulations becomes faster for more threads, but the speedup is not as ideal for more than ten threads.

In figure 5 we can see that the performance of both PThreads and OpenMP are comparable.

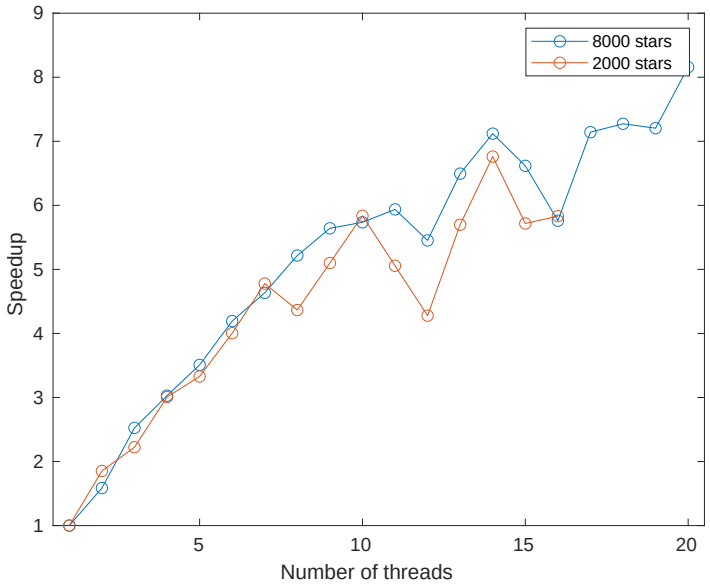

Figure 7: 8000 and 2000 stars run for 200 iterations with different amount of threads on Vitsippa using one parallelization

4 Optimization

We looked at both serial and and parallell optimizations to improve the speed of our program.

4.1 Compiler Optimization Flags

We found that the -Ofast flag is the best for our purpose. Running the same simulation with 3000 stars for 100 iterations without graphics takes 3.771 seconds with the O2 flag, 3.539 second with the O3 flag, and 3.092 seconds with the Ofast flag. For correctness and reproducibility all tests were made on the same computer with processor i5-6200U and the same compiler GCC version 7.4.

4.2 Serial Optimizations

The code was first written to function but then certain simplifications were made. For example the stars were made to be initialized directly in the main function instead of doing an expensive function call to initialize each star.

In the beginning we had several structs that included other structs for example a vector struct to keep track of the two dimensions but this was simplified and removed. The contents of the structs were written so as to have better alignment so that memory is used more efficiently.

To optimize our for-loops by loop unrolling which is built into the compiler we rewrote some of our variables. Loop unrolling is more likely to work if loop control variables such as N are declared as const, because then the compiler knows that the value will not be altered by the loop execution. This was implemented.

4.3 Further Parallelization

We can see in figure 8 that adding another parallelized for-loop in the OpenMP-code resulted in no significant further speedup. This is expected. From the profiling in figure 2 we know that the already parallelized function that calculates force is responsible for most of the computation time, so adding further parallelizations doesn't make a big impact.

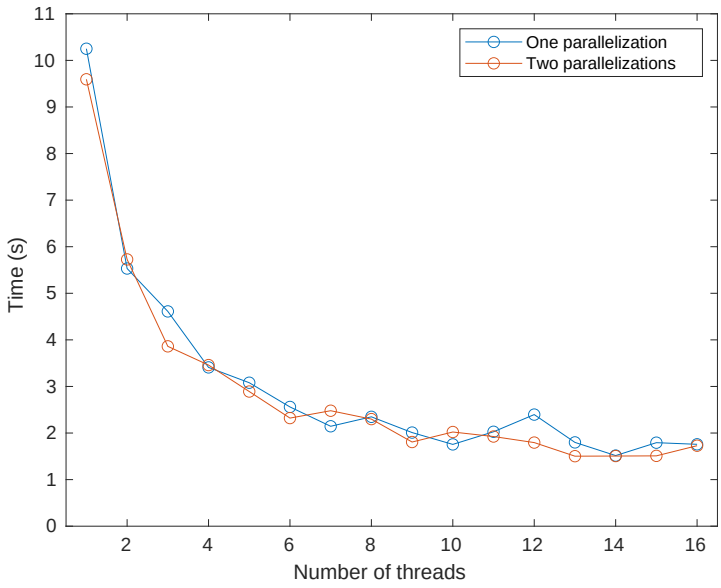

Figure 8: 2000 stars run for 200 iterations with different amount of threads on Vitsippa using one and two parallelizations

5 Conclusions

Implementing the same galaxy simulation in different ways we can design a much faster program than the initial solution. By exploring both serial and parallel optimizations the code is made

The direct-sum implementation and the Barnes-Hut implementations vary much in their efficiency and by simple rethinking the implementation much faster results could be achieved. By parallelizing the code we could see even faster results and both PThreads and OpenMP methods work well.