UPPSALA UNIVERSITET

DEPARTMENT OF MATHEMATICS - Course 2020

HIGH PERFORMANCE PROGRAMMING - FINAL REPORT

KIERAN BARBER

PANAGIOTIS PAPIAS

Teacher
Sverker Holmgren

Contents

1 The Problem **2**

2 The Solution - N Body Problem **3**

3 Performance and Discussion - N Body Problem **6**
- 3.1 Timings . . . 6
- 3.2 Complexity . . . 7

4 The Solution - Barnes Hut Algorithm **7**
- 4.1 Quadtree . . . 8
- 4.2 Forces . . . 11
- 4.3 Other Files . . . 12

5 Performance and Discussion - Barnes Hut Algorithm **13**
- 5.1 Timings - Getting up to speed . . . 13
- 5.2 Timings - Which section to focus on . . . 13
- 5.3 Timings - Optimised . . . 13
- 5.4 Unsuccessful Optimisations . . . 15
- 5.5 Complexity . . . 16

6 Optimisation - Pthreads **17**

7 Optimisation - OpenMP **19**

8 CPU Configuration **20**

1 The Problem

For this assignment, we are going to implement a program that will calculate the evolution of N particles in a gravitational simulation, whereby we are given an initial set of particles. The simulation will be done in 2 spatial dimensions with an x and y coordinate.

To help describe the evolution of these particles, we use Newton's law of gravitation in two dimensions, which states that the force exerted on particle i by particle j is given by

$$\boldsymbol{f}_{ij} = -\frac{Gm_i m_j}{\|\boldsymbol{r}_{ij}\|^3}\boldsymbol{r}_{ij}.$$

where G is the gravitational constant, m_i and m_j are the masses of the particles, and $\boldsymbol{r}_{ij}$ is the distance vector given by $\boldsymbol{x}_i - \boldsymbol{x}_j$ with $\boldsymbol{x}_i$ being the position of particle i.

We will make use of the following force equation (Plummer Sphere Force Equation) to describe the forces on the particles. The force on particle i is given by

$$\boldsymbol{F}_i = -Gm_i \sum_{j=0, j\neq i}^{N-1} \frac{m_j}{(\|\boldsymbol{r}_{ij}\| + \epsilon_0)^3}\boldsymbol{r}_{ij}.$$

where ϵ_0 is a smoothing parameter and in the computations, we used the value $\epsilon = 10^{-3}$.

To update the particle positions, we use the Euler Symplectic Time Integration method. The equations describing this are

$$\begin{cases} \boldsymbol{a}_i^n = \frac{\boldsymbol{F}_i^n}{m_i}; \\ \boldsymbol{u}_i^{n+1} = \boldsymbol{u}_i^n + \Delta t \boldsymbol{a}_i^n; \\ \boldsymbol{x}_i^{n+1} = \boldsymbol{x}_i^n + \Delta t \boldsymbol{u}_i^{n+1}, \end{cases}$$

where Δt is the time step size, $\boldsymbol{a}_i$ is the acceleration of particle i, $\boldsymbol{u}_i$ is the velocity and $\boldsymbol{x}_i$ is the position of particle i. In the computations, we used $\Delta t = 10^{-5}$. The value of G depends on N and is given by $G = 100/N$.

2 The Solution - N Body Problem

We began by writing our code in a file we named galsim.c. This is where all the functionality that we require is stored. We used code from graphics to be able to visualise the results of the simulation.

Within the galsim.c file, we write our main program that will enable us to calculate the evolution of a system of N particles. This file begins with calling the relevant libraries we require. Below this, we take in the variables for the graphics, gravity and the smoothing constant epsilon. To improve runtime, we have set epsilon as a constant.

Below this, we have our structs, one for the various vectors we will use and one for the particles attributes. The vector struct takes two doubles, one for x and one for y. The particle struct takes two vectors for position and and velocity, and two doubles, one for mass and brightness. These have been ordered so that they are in the same order as the data that's read in from the gal-files.

Below this, we introduced print functions that we would use for debugging. This includes print_vector_info, print_particle_info and print_all_particle_info. After this, we include a function we then call in the main function to get computation times for each optimisation.

We now look to the building of the force functions. Since the Plummer spheres force equation has smoothing involved, we looked to implement this. To do this, we built each component of the equation in separate functions. The first function to be built was $\boldsymbol{r}_{ij}$. We only computed this in one way. The next to compute was r_{ij}^3. This was implemented in two different ways.

Version 1

```
double norm3(vector_t v) {
    return pow((sqrt(pow((v.x), 2) + pow((v.y), 2)) + epsilon), 3);
}
```

Version 2 - 4

```
double norm3(vector_t v) {
    double n3;

    n3 = sqrt(v.x*v.x + v.y*v.y) + epsilon;
    n3 = n3 * n3 * n3;
    return n3;
}
```

With the above versions of norm3, we then implemented them into two versions of a function to calculate the force on particle i from particle j. To do this, we had to calculate in both the x and y directions. Initially, we chose to include all terms for both of these directions.

Version 1 - 2

```
vector_t force(particle_t pi, particle_t pj) {
    vector_t f;

    f.x = -(G * pi.mass * pj.mass * r_vector(pi.position, pj.position).x)
            / norm3(r_vector(pi.position, pj.position));
    f.y = -(G * pi.mass * pj.mass * r_vector(pi.position, pj.position).y)
            / norm3(r_vector(pi.position, pj.position));
```

```
    return f;
}
```

We decided then that we would calculate any repetitions in each equation just once, so that this may improve computation time.

Version 3 - 4

```
vector_t force(particle_t pi, particle_t pj) {
    vector_t f;
    vector_t r;
    double a;

    r = r_vector(pi.position, pj.position);
    a = -(G * pi.mass * pj.mass) / norm3(r);

    f.x = a * r.x;
    f.y = a * r.y;

    return f;
}
```

Now that we have functions to compute the force applied to each particle, we can now loop through each particle, and compute the total force applied to each particle.

Version 1 - 3

```
vector_t total_force(particle_t* p, int i, int N) {
    vector_t f = {0.0,0.0};

    for(int j = 0; j < N; j++) {
        if(i != j) {
            vector_t f_temp = force(p[j], p[i]);
            f.x += f_temp.x;
            f.y += f_temp.y;
        }
    }
    return f;
}
```

One final improvement we applied to this was to note that since we had the forces computed from particle i to particle j, we also had the negative of this, so we then had the force applied from particle j to particle i.

Version 4

```
void compute_forces(vector_t* forces, particle_t* particles, int N) {

    for(int i = 0; i < N; i++) {
        forces[i].x = 0.0;
        forces[i].y = 0.0;
    }

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

        for(int j = i + 1; j < N; j++) {
                vector_t f = force(particles[j], particles[i]);
                forces[i].x += f.x;
                forces[i].y += f.y;
```

```
                forces[j].x -= f.x;
                forces[j].y -= f.y;
        }

    }
}
```

One final thing that we used to help us was to implement timings directly into the Makefile code so that when it came to computing wall timings at the end with different optimisations, we could do this directly from the Makefile.

3 Performance and Discussion - N Body Problem

3.1 Timings

Version	-OO	-O3	-Ofast
V1	199.702909	110.818015	3.372535
V2	94.216680	4.225227	4.065753
V3	45.380386	3.521552	3.271567
V4	24.352817	2.538566	2.454337

Given the optimisations described through code changes in the previous section, we have above, our timings for various optimisation flags.

Starting with Version 1, we can see that setting no optimisation flags to the code leaves us computing for a significant amount of time. this is almost halved by taking the -O3 flag. The -Ofast flag takes a significantly shorter amount of time to compute this system. A potential reason we can see here is that it sees the use of the pow function and realises it just simply needs to compute $x * x$ rather than x^2 for instance.

The second version of the code that we produced moved from calculating norm3 in the version we described above. This gave significant increases in time performance for the -O0 and -O3 flags, however, it actually increased computation time for the -Ofast flag. We have not understood why this is the case.

For Version 3, we altered the way in which we computed the force for each particle. Instead of calling functions for both the x and y directions, we decided to take out the elements of each equation that are the same and compute them before we computed the force in each of the x and y directions, thus meant that we were doing half of the original calculations. Again, this gave significant improvements in performance for -O0. Proportionately, it gave big performance improvements for -O3 and -Ofast.

The final Version took into account that we could compute the forces in half the time. This is achieved since once we know the force applied to particle i from particle j, we know that the force applied to particle j from particle i is just the negative. This gave significant performance improvements with each flag.

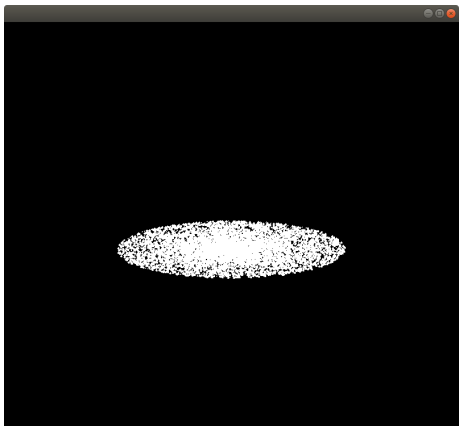

Figure 1: A simulation of a galaxy

3.2 Complexity

Finally, below we have a plot outlining the computation times measured against $O(N^2)$.

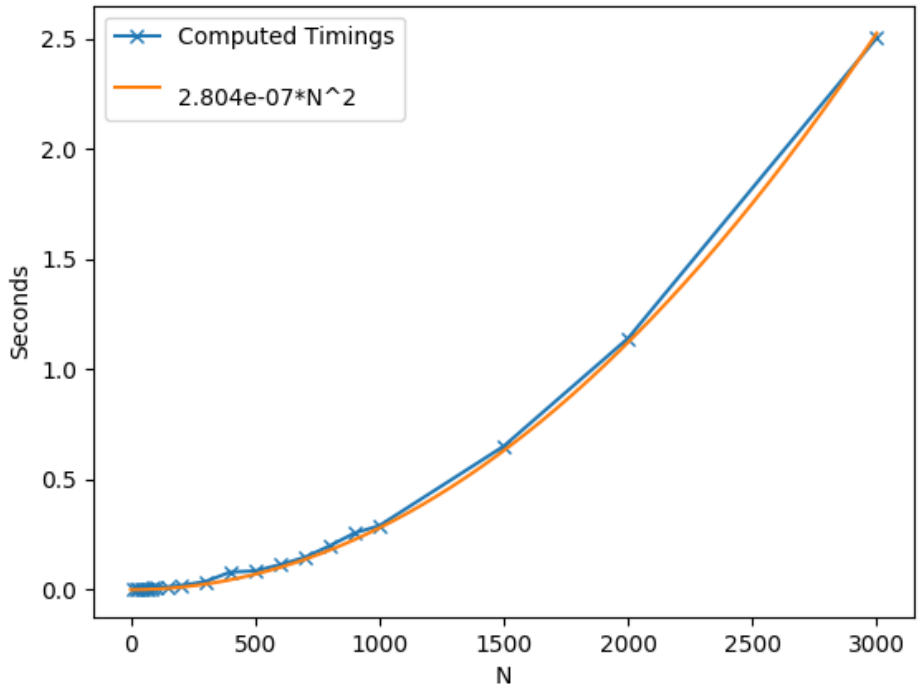

Figure 2: Time Complexity Plot

4 The Solution - Barnes Hut Algorithm

To solve the N body problem, we implemented something known as the Barnes-Hut algorithm. The purpose of this algorithm is to improve computation time, whilst maintaining a high level of accuracy. The idea is to take a group of particles located near to each other and instead of computing forces individually, we do it as group and calculate the force they exert on another particle. To maintain accuracy, we only do this for groups that are far away from the particle. This means that if we have M particles in the group, then we save M-1 force calculations in which we need to compute.

For this assignment, we initially chose to separate the code into files that represented each section of code we required. We chose to do this for a cleanliness aspect. These files are:

- `box_bounds.c box_bounds.h`
- forces.c forces.h
- galsim.c

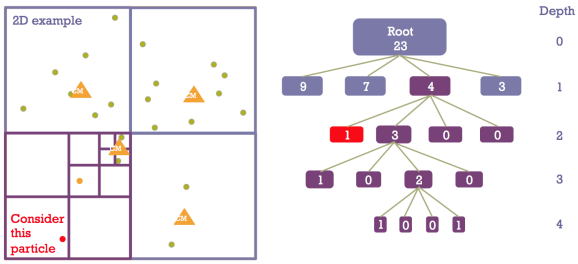

Figure 3: Diagram of Barnes Hut method

- particle.c particle.h
- quad.c quad.h
- vector.c vector.h
- Makefile
- timings.sh

4.1 Quadtree

To begin with, we constructed the quadtree in the file quad.c. We take an empty tree and insert the particles iteratively, one by one. The first case involves simply adding a particle to an empty tree. The other cases involve recursion.

So in case one, if the tree is empty, we insert the particle into the root node. When we add the particle here, we must also set its mass and centre of mass at the same time, which in this case is easy since it will take the same value as the single particle.

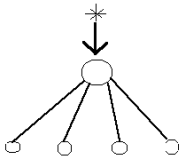

Figure 4: Diagram of Empty Tree

The second case, is that we traverse the tree and we end up on a node that is taken, but with no children. In this scenario, we must take the particle to one of the children and then also take the particle that was in the node into one of the other children. If they end up going to the same place again, then we traverse the tree

until they are split into separate nodes. We update the mass and centre of mass for each particle at the same time. The final case in our code is if we already have a particle in one of the children spots but not all of them.

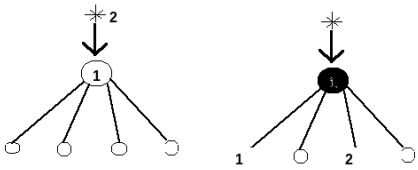

Figure 5: Diagram of Node with no children

In this scenario, we simply need to traverse the tree until we get to the node with space in its children so that this new particle can be placed here.

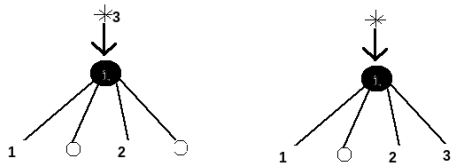

Figure 6: Diagram of Node with children

Quadtree code

```
void quadtree_add_particle(quadtree_t** tree, particle_t particle, box_bounds_t box) {

if(*tree == NULL) {
    *tree = (quadtree_t*)malloc(sizeof(quadtree_t));
    (*tree)->mass = particle.mass;
    (*tree)->centre_of_mass = particle.position;
    (*tree)->box = box;
    (*tree)->north_west = NULL;
    (*tree)->north_east = NULL;
    (*tree)->south_west = NULL;
    (*tree)->south_east = NULL;
} else if((*tree)->north_east == NULL && (*tree)->north_west == NULL && (*tree)->
south_west == NULL && (*tree)->south_east == NULL) {

    particle_t particle_holder = {(*tree)->centre_of_mass, (*tree)->mass, {0.0, 0.0}, 1};

    double xcentre = ((*tree)->box.xlower + (*tree)->box.xupper)/2;
    double ycentre = ((*tree)->box.ylower + (*tree)->box.yupper)/2;

    if(particle_holder.position.x <= xcentre && particle_holder.position.y < ycentre) {
        box_bounds_t new_box = {box.xlower, xcentre, box.ylower, ycentre};
        quadtree_add_particle(&(*tree)->south_west, particle_holder, new_box);
    } else if(particle_holder.position.x > xcentre && particle_holder.position.y <=
ycentre) {
        box_bounds_t new_box = {xcentre, box.xupper, box.ylower, ycentre};
        quadtree_add_particle(&(*tree)->south_east, particle_holder, new_box);
    } else if(particle_holder.position.x < xcentre && particle_holder.position.y >=
ycentre) {
        box_bounds_t new_box = {box.xlower, xcentre, ycentre, box.yupper};
        quadtree_add_particle(&(*tree)->north_west, particle_holder, new_box);
    } else if(particle_holder.position.x >= xcentre && particle_holder.position.y >
ycentre) {
        box_bounds_t new_box = {xcentre, box.xupper, ycentre, box.yupper};
        quadtree_add_particle(&(*tree)->north_east, particle_holder, new_box);
    }

    if(particle.position.x <= xcentre && particle.position.y < ycentre) {
        box_bounds_t new_box = {box.xlower, xcentre, box.ylower, ycentre};
        quadtree_add_particle(&(*tree)->south_west, particle, new_box);
    } else if(particle.position.x > xcentre && particle.position.y <= ycentre) {
        box_bounds_t new_box = {xcentre, box.xupper, box.ylower, ycentre};
        quadtree_add_particle(&(*tree)->south_east, particle, new_box);
    } else if(particle.position.x < xcentre && particle.position.y >= ycentre) {
        box_bounds_t new_box = {box.xlower, xcentre, ycentre, box.yupper};
        quadtree_add_particle(&(*tree)->north_west, particle, new_box);
    } else if(particle.position.x >= xcentre && particle.position.y > ycentre) {
        box_bounds_t new_box = {xcentre, box.xupper, ycentre, box.yupper};
        quadtree_add_particle(&(*tree)->north_east, particle, new_box);
    }

    (*tree)->centre_of_mass.x = ((*tree)->centre_of_mass.x*(*tree)->mass + particle.
position.x*particle.mass)/((*tree)->mass + particle.mass);
    (*tree)->centre_of_mass.y = ((*tree)->centre_of_mass.y*(*tree)->mass + particle.
position.y*particle.mass)/((*tree)->mass + particle.mass);
    (*tree)->mass += particle.mass;

} else {
    (*tree)->centre_of_mass.x = ((*tree)->centre_of_mass.x*(*tree)->mass + particle.
position.x*particle.mass)/((*tree)->mass + particle.mass);
    (*tree)->centre_of_mass.y = ((*tree)->centre_of_mass.y*(*tree)->mass + particle.
```

```
    position.y*particle.mass)/((*tree)->mass + particle.mass);
        (*tree)->mass += particle.mass;

        double xcentre = ((*tree)->box.xlower + (*tree)->box.xupper)/2;
        double ycentre = ((*tree)->box.ylower + (*tree)->box.yupper)/2;

        if(particle.position.x <= xcentre && particle.position.y < ycentre) {
            box_bounds_t new_box = {box.xlower, xcentre, box.ylower, ycentre};
            quadtree_add_particle(&(*tree)->south_west, particle, new_box);
        } else if(particle.position.x > xcentre && particle.position.y <= ycentre) {
            box_bounds_t new_box = {xcentre, box.xupper, box.ylower, ycentre};
            quadtree_add_particle(&(*tree)->south_east, particle, new_box);
        } else if(particle.position.x < xcentre && particle.position.y >= ycentre) {
            box_bounds_t new_box = {box.xlower, xcentre, ycentre, box.yupper};
            quadtree_add_particle(&(*tree)->north_west, particle, new_box);
        } else if(particle.position.x >= xcentre && particle.position.y > ycentre) {
            box_bounds_t new_box = {xcentre, box.xupper, ycentre, box.yupper};
            quadtree_add_particle(&(*tree)->north_east, particle, new_box);
        }
    }
}
```

4.2 Forces

In order to compute the forces we need the quadtree, which is built in quad.c, the particle that the force is computing for, and `theta_max`. Again, we have three scenarios in which we need to compute the forces. The first case is that if the tree is NULL then we have no forces in which there is any interaction.

The next case we have is if the particle that is inserted next in the tree is firstly not in a particular boxed region and either far enough away from this boxed region or it has no children, then we can compute the force applied to it from the boxed region. We have to check this, since care is needed to not include the particle itself.

The final case we have is to compute the forces of each child and then sum these together to compute the force of the node.

Forces code

```
    vector_t quad_force(quadtree_t* tree, particle_t particle, double theta_max) {
    vector_t f;
    vector_t r;
    double a;

    if(tree == NULL) {
        f.x = 0.0;
        f.y = 0.0;
    } else if(!vector_in_box(particle.position, tree->box) && (theta_bounds(tree, particle,
    theta_max)
            || (tree->north_west == NULL && tree->north_east == NULL && tree->south_west ==
    NULL && tree->south_east == NULL))) {
        r = r_vector(tree->centre_of_mass, particle.position);
        a = -(G * particle.mass * tree->mass) / norm3(r);
        f.x = a * r.x;
        f.y = a * r.y;
    } else {
        vector_t f1 = quad_force(tree->north_west, particle, theta_max);
        vector_t f2 = quad_force(tree->north_east, particle, theta_max);
        vector_t f3 = quad_force(tree->south_west, particle, theta_max);
        vector_t f4 = quad_force(tree->south_east, particle, theta_max);
```

```
            f.x = f1.x + f2.x + f3.x + f4.x;
            f.y = f1.y + f2.y + f3.y + f4.y;
        }
        return f;
}
}
```

4.3 Other Files

The purpose of `box_bounds.c` is to house a function called `vector_in_box` which is called into functions in forces. This function is used to determine whether a particle traversing the quad tree is within the bounds of a certain node.

Since our code has been separated out into different files, galsim.c only holds particle position updates and updates outside of the main function. In the main function, we take the arguments from the user, and call the compute force functions to get our output.

The particle.c file contains functions that help read and write the particle data involved.

The vector.c file contains the `r_vector` and norm functions that helps compute the forces on the particles.

For convenience, we have included a timings.sh file that runs the different versions of Makefile on different optimisation flags.

5 Performance and Discussion - Barnes Hut Algorithm

5.1 Timings - Getting up to speed

Version	-OO	-O3	-Ofast -march=native	-O4
V1.1	24.304349	13.470492	12.901046	13.474097
V1.2	24.204505	2.549037	2.035497	2.542937
V1.3	12.155917	2.900762	2.836586	2.902077

Since the functions are now spread over several files, we need to check that we have as efficient timings as in the version without the Barnes Hut algorithm. To test this, we first use a check file to make sure our results are accurate. The timing results of this we omit.

The next step is to then compare timing results to that of previous results we have obtained.

Version 1.1 above is based on previous results from which we can compare. This involves using 3000 particles and 100 time steps. For this version (1.1) we have the functions `r_vector`, norm and norm3 in the file vector.c, with the force functions from a previous version now in the forces.c file and all are called to the galsim.c main file. This shows some significant loss in computation time compared to a previous version. We now take all the vector functions into the forces.c file to see if this helps with computation time.

The next version (1.2) does indeed appear to speed up the computation times significantly for the flags apart from -O0.

Finally, in version (1.3), we make a comparison with the quadtree implemented for 3000 particles and 100 time steps to see how this compared. Before testing this, we ran this with θ set to zero first to ensure we maintained accuracy before optimising for θ. This is allowed an accuracy of up to 10^{-3}. The optimal value we obtained for θ is 0.25. This ran only marginally slower for all the flags apart from -O0, where this ran twice as fast.

5.2 Timings - Which section to focus on

Before we begin optimisation of the code, we look to which part of the code requires the most optimisation. We split the code into computing quadtree build time and forces computation time. We focused on the -Ofast and -march=native optimisation flags. The quadtree took 0.144213 seconds to compute for 5000 particles with 100 time steps. The forces took 5.302154 seconds to compute with the same parameters. On this evidence, we will look to focus our optimisations on computing the forces, since this took approximately 97% of the computation time.

5.3 Timings - Optimised

Version	-OO	-O3	-Ofast -march=native	-O4
V2.1	23.720430	5.669060	5.449106	5.842755
V2.2	23.545072	5.277797	4.420938	5.303275
V2.3	18.177161	5.102843	4.393660	5.105376
V2.4	17.941679	4.841763	4.289874	4.820603

Version 2.1 here involves having all the function calls in forces aside from `vector_in_box`, which has been left out in the `box_bounds.c` file.

This is where our first optimisation is. In Version 2.2, we chose to move `vector_in_box` in to the forces

file. This does indeed speed up the computation times for all apart from the -O0 flag.

In Version 2.3, we made an optimisation involving making a modification to `theta_bounds`, which in turn meant we would have to make a modification to `quad_force`. We took in two extra arguments in `theta_bounds`, where we take pointers so that we only have to compute norm(r) just the once. This has some computational time effects for each flag, but mostly -O0.

Version 3 - `theta_bounds`

```
    int theta_bounds(quadtree_t* tree, particle_t particle, double theta_max, vector_t* r,
    double* n) {
    double xwidth = tree->box.xupper - tree->box.xlower;
    *r = r_vector(tree->centre_of_mass, particle.position);
    *n = norm(*r);

    if(xwidth / *n < theta_max) {
        return 1;
    }
    else {
        return 0;
    }
}
```

In version 2.4, the final optimisation involved using our final `theta_bounds` function, where we switched from dividing to multiplying. We did this since dividing is computationally more expensive than multiplying. This made for a minimal increase in performance on all the flags.

Version 4 - `theta_bounds`

```
    int theta_bounds(quadtree_t* tree, particle_t particle, double theta_max, vector_t* r,
    double* n) {
    *r = r_vector(tree->centre_of_mass, particle.position);
    *n = norm(*r);

    return tree->box.xupper - tree->box.xlower < theta_max * (*n);
}
```

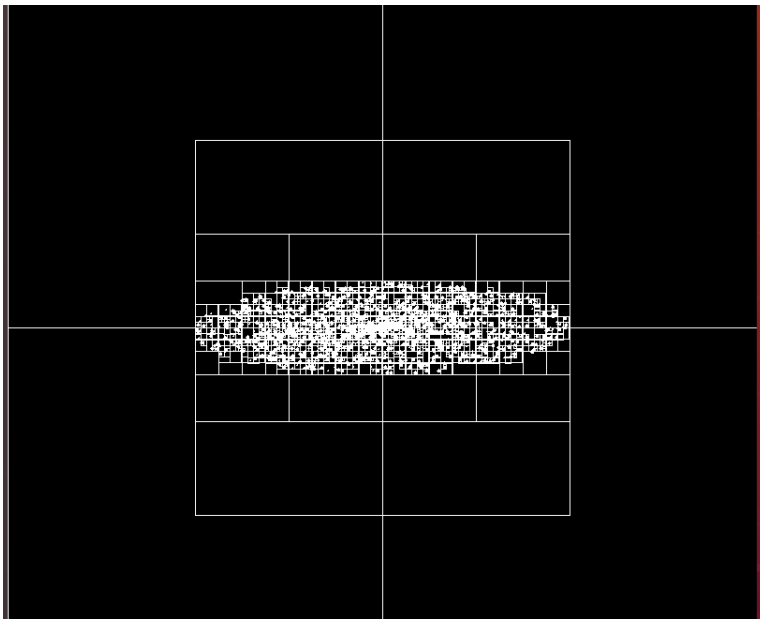

Figure 7: A simulation of a galaxy

5.4 Unsuccessful Optimisations

We tried a few more optimisations on top of the ones described above, however, these were unsuccessful.

The first of these optimisations was to bring the forces.c file into the galsim.c file to see if there could be further time reductions in computation time. When this didn't produce any better time performance we decided to also bring in the quad.c file into the galsim.c file so that they are all located in the same place. Both were unsuccessful optimisations.

5.5 Complexity

Finally, below we have a plot outlining the computation times measured against $O(Nlog(N))$.

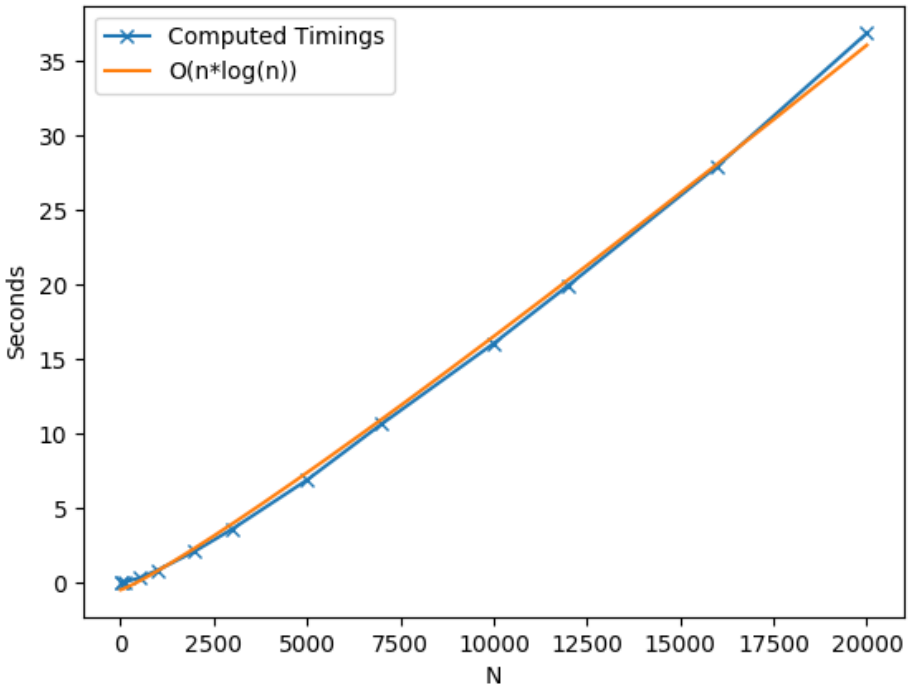

Figure 8: Time Complexity Plot

6 Optimisation - Pthreads

Since we saw from the previous section that the majority of computation time (approximately 95%) is spent computing the forces, we parallelize the code using pthreads in this part of the code.

Instead of computing the forces in a for loop on one thread, we have modified the code here to have a function point to all the relevant details of the particles updated. This function also included a start and stop point to compute forces given that we want to run certain sections of code on different threads.

To make sure this all runs fine, we need to include -pthread in the CCFLAGS.

We have the original problem size of 5000 particles and 100 time steps as one example and then 20000 particles with 200 time steps to see the potential speedup.

```
    typedef struct quad_force_args {
    quadtree_t* tree;
    particle_t* particles;
    double theta_max;
    vector_t* forces;
    int start;
    int stop;
} quad_force_args_t;

    void* quad_force_wrapper(void* argument) {
    quad_force_args_t* args  = argument;

    for(int i = args->start; i < args->stop; i++) {
        args->forces[i] = quad_force(args->tree, args->particles[i], args->theta_max);
    }

    return NULL;
}
//=============================================================================
void compute_quad_forces(vector_t* forces, quadtree_t* tree, particle_t* particles, int N,
    double theta_max, int nthreads) {

    pthread_t* thread_ptr = malloc(nthreads*sizeof(pthread_t));
    quad_force_args_t* args = malloc(nthreads*sizeof(quad_force_args_t));

    for(int i = 0; i < nthreads; i++) {
        args[i].tree = tree;
        args[i].particles = particles;
        args[i].theta_max = theta_max;
        args[i].forces = forces;
        args[i].start = (i * N) / nthreads;
        args[i].stop = ((i + 1)*N) / nthreads;
        pthread_create(&thread_ptr[i], NULL, quad_force_wrapper, &args[i]);
    }

    for(int i = 0; i < nthreads; i++) {
        pthread_join(thread_ptr[i], NULL);
    }
}
```

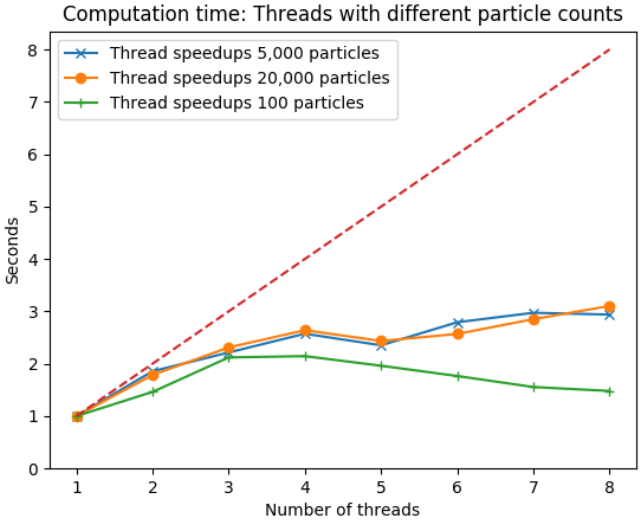

Figure 9: Computation Times

As we can see, larger files definitely benefit from extra cores at their disposal. Smaller files do not seem to require this, to the point where performance is actually reduced. The large difference in performance to ultimate best use is likely down to not having the 8 cores in which we can optimise with.

7 Optimisation - OpenMP

Finally, we optimised the code using OpenMP. For this, we reverted back to code used prior to pthreads. For this, we simply added a line of code above our computationally costly for loop with different number of threads used.

```
    void compute_quad_forces(vector_t* forces, quadtree_t* tree, particle_t* particles, int N,
     double theta_max) {
    #pragma omp parallel for
    for(int i = 0; i < N; i++) {
            forces[i] = quad_force(tree, particles[i], theta_max);
    }
}
```

Before running this code, we run in the command line 'export OMP_NUM_THREADS=x', where x can take any positive integer, but for our cases, we limited ourselves to 1 through 8.

What we see is that on up to 2 threads, pthreads and openMP give similar computation time. After this however, it shows that OpenMP is able to utilise the threads more efficiently than pthreads. This, however, remains a static increase in performance relative to one another. OpenMP only starts to look like its increasing in performance relative to pthreads at 8 threads.

We also attempted to increase performance time with the particle updates function. As we tried this however, it seems as though performance actually dropped, so we didn't include this in the final optimised timings.

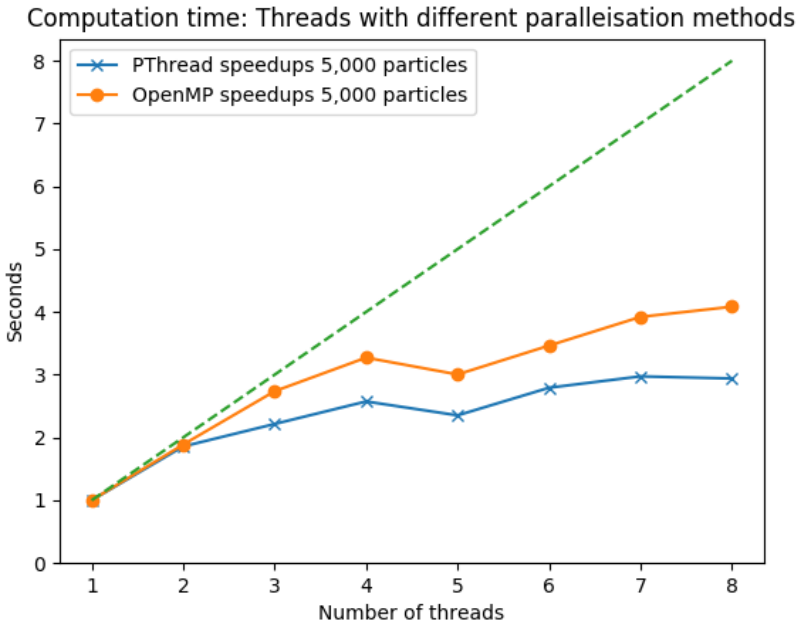

Figure 10: Time Complexity Plot

8 CPU Configuration

```
Architecture:        x86_64
CPU op-mode(s):      32-bit, 64-bit
Byte Order:          Little Endian
CPU(s):              8
On-line CPU(s) list: 0-7
Thread(s) per core:  2
Core(s) per socket:  4
Socket(s):           1
NUMA node(s):        1
Vendor ID:           GenuineIntel
CPU family:          6
Model:               142
Model name:          Intel(R) Core(TM) i5-8250U CPU @ 1.60GHz
Stepping:            10
CPU MHz:             700.041
CPU max MHz:         3400,0000
CPU min MHz:         400,0000
BogoMIPS:            3600.00
Virtualization:      VT-x
L1d cache:           32K
L1i cache:           32K
L2 cache:            256K
L3 cache:            6144K
NUMA node0 CPU(s):   0-7

(Ubuntu 7.4.0-1ubuntu1~18.04.1) 7.4.0
```