UPPSALA
UNIVERSITET

High Performance Programming
Assignment 4

Authors:

Amanda Seger

Lovisa Rygaard

Matilda Tiberg

Moa Li Lekander

Uppsala

February 28, 2021

Contents

1 The Problem

The goal of the assignment is to implement the Barnes-Hut algorithm in C to reduce the calculations required for simulating the N-body problem. The exact solution to the problem has an $\mathcal{O}(N^2)$ complexity for N amount of particles, ergo, the code quickly becomes slow for an increasing size of the problem. In this assignment, the performance and the complexity of the Barnes-Hut algorithm will be examined and compared to the exact way of solving the N-body problem. Additionally, the two algorithms were parallelized using Pthreads.

1.1 The N-body Problem

The two-body problem consists of determining how two particles, p_i and p_j, interacts with each other depending on their relative position to each other and their masses. According to Newton's law of gravitation, they will exert a force on each other, thus causing the two particles to move. Depending on their relative distance to each other, the forces will differ in both magnitude and direction. Hence, it is a time-dependent system since the forces will cause the particles to move, thus causing the forces to change etc. This system can then easily be expanded to handle N amount of particles, where the total force exerted on particle i is given by equation (1). Here, G is the gravitational constant defined as $100/N$, m_i is the mass of particle i, m_j is the mass of particle j, N is the amount of particles in the system, r_{ij} is the magnitude of the distance between the particles, and $\mathbf{r}_{ij}$ is the distance vector. The constant factor ε_0 is defined as 10^{-5} and exists for the purpose of improving the stability of the system by capping the maximum force between two particles.

$$\mathbf{F}_i = -Gm_i \sum_{j=0, j\neq i}^{N-1} \frac{m_j}{(r_{ij} + \varepsilon_0)^3} \mathbf{r}_{ij} \tag{1}$$

1.2 Time Stepping

The update of positions are done by using the *symplectic Euler* time-integration method, and the governing equations for updating the properties of particle p_i are presented in equation (2). $\mathbf{a}$ denotes the acceleration, $\mathbf{u}$ is the velocity and $\mathbf{x}$ is the position. The current time step is denoted by n, and Δt denotes size of the time steps. This algorithm has a time complexity of $\mathcal{O}(N^2)$, thus causing the time to increase quadratically with an increased amount of particles.

$$\begin{aligned} \mathbf{a}_i^n &= \frac{\mathbf{F}_i^n}{m_i} \\ \mathbf{u}_i^{n+1} &= \mathbf{u}_i^n + \Delta t \mathbf{a}_i^n \\ \mathbf{x}_i^{n+1} &= \mathbf{x}_i^n + \Delta t \mathbf{u}_i^{n+1} \end{aligned} \tag{2}$$

1.3 The Barnes-Hut Algorithm

By using using equation (1) to directly sum up the forces acting on particle p_i, the cost of the calculation will significantly grow with an increased number of particles in the simulation. To reduce the calculation time, an approximate method can be used instead. The Barnes-Hut method reduces the time complexity by dividing the domain into subgroups, where many particles can be treated as one object (for details, read below). Then, to calculate the net force on p_i from the particles in said group, only one calculation is performed. This means that if the group contains M particles, then $M - 1$ less calculations of the force needs to be done. The calculations follow equation (1) and (2) in the same manner as before, except that m_j in equation (1) now consists of the central mass of the approximated object instead of a seperate particle.

The essential part for this algorithm is the use of a quadtree, which is similar to a binary tree, except a parent node can have four children instead of two. Because of the quadtree, the whole simulation

domain can be stored as the root, and the domain can be divided into four quadrants, where each quadrant is stored in one of the children nodes. Then, if one quadrant contains more than one particle, that quadrant needs to be split into four more quadrants and store its particles in the correct sub-quadrant. The leaf nodes will then contain exactly one particle, whereas the nodes further up in the quadtree will store the center of mass and total mass of all the particles below it. This means that the domain will consist of an adaptive grid with many sub-quadrants, each with either exactly zero or one particles. For an illustrative example, see Figure 1, where the particles are marked as white, and the squares is how the grid was divided. As can be seen, the areas where more particles are situated, the squares are smaller, whereas the squares are bigger at the edge where only a few particles are positioned. The approximation of an object is satisfied if the distance between the particles in the quadrant is small enough compared to the distance between the external particle, p_i, and the midpoint of the quadrant. Mathematically, this is expressed as $h/r << 1$, where h is the maximum distance between two particles within the quadrant, and r is the distance to the quadrant.

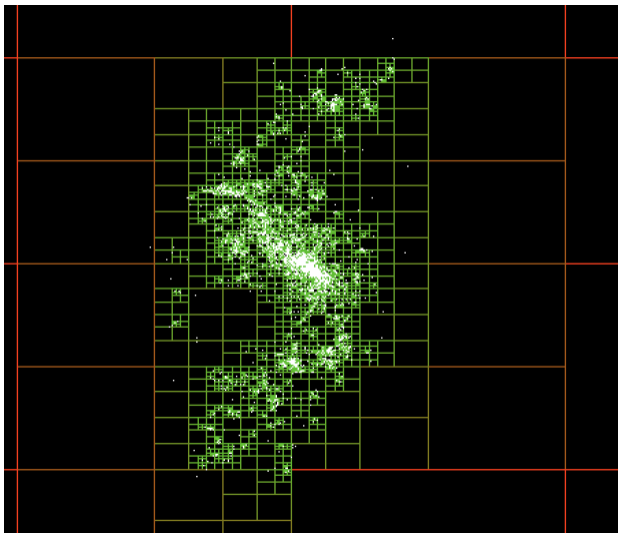

Figure 1: *Visualization of how the domain can be divided into multiple quadrants.*

In order to achieve an effective algorithm, a way of determining how deep into the tree the traversing should be made must be defined. The value of θ, calculated from equation (3), is used to decide if the node if sufficiently far away from p_i, or if the algorithm should keep traversing down the quadtree. The threshold value is defined as θ_{max}, and if the value of θ exceeds θ_{max}, then the quadtree needs to be traversed further. The threshold can attain the values $\theta_{max} \in [0,\ 1]$, where a value of $\theta_{max} = 0$ implies that no approximations are made, i.e. the exact solution should be found. Vice versa, a higher value of θ_{max} means that more approximations will be made, thus giving a less accurate result. The value of θ_{max} must therefore be chosen to be as high as possible (to speedup the time required for making the computations) while still achieving the desirable level of accuracy of the calculations.

$$\theta = \frac{\text{width of current quadrant}}{\text{distance between } p_i \text{ and middle of current quadrant}} \tag{3}$$

2 The Solution

The calculations were made in the code displayed in Listing 1 in Appendix A, and all created functions are explained in the sections below. In the code there are also three implemented structs to organize the values used in the functions. The main program is made in such a way that it achieves six extra parameters from the user which are stored in various variables. Moreover, an array of the struct `particles_t` and a struct `quadtree_t` are created to be used in functions to come. The functions `getInput`, `calculateNewPositions` and `setOutput` are returning an integer value to handle possible error messages. If the integer is -1 the main program will first make sure to free all allocated memory before terminating the program. The function `get_wall_seconds` is also included to calculate the run time of the program in wall seconds.

2.1 Structs

The program includes three structs for storing necessary values. The purpose of the struct `particle` is simply to hold all values of a particle. The struct `quadtree` is created to be used as a tree structure holding the necessary values for the Barnes-Hut algorithm. Therefore, it contains the values of the center of mass, a `particle` struct, a `quadrant` struct and four children quadtrees. The center of mass information is the x and y positions and the total mass stored in the struct and its children. The third struct `quadrant` stores the size of the current quadrant in the Barnes-Hut algorithm and the position of the quadrants origin in the whole simulated space. The origin was chosen to be placed in the left bottom corner.

2.2 getInput

The function `int getInput(const char* __restrict filename, quadtree_t** __restrict root, particle_t* __restrict particles, const int N)` uses the given input file `filename` and number of particles `N` to store the input values in the array of particles `particles` and places them in the quadtree `root`. The file is first read with `fread` into an array of doubles. The double array is thereafter used to write all values into their suitable variables in the `particle` array. To add each particle a `quadrant` with the size of the whole space is created and together with the `particle` is used in the function `insert` to update the quadtree `root`.

The function also includes to error checks where an error message is printed and a necessary `free` and `fclose` is used before returning to the main program. At the end of the function the functions `free` and `fclose` are also used.

2.3 insert

The purpose of the function `int insert(quadtree_t** node, particle_t* __restrict particle, quadrant_t* __restrict quadrant)` is to insert a `particle` into a quadtree `node` with the size of the `quadrant`.

There are three possibilities of handling the node, the first case is if the node is `NULL`. In that case, a new `quadtree` is allocated and is given the properties of the input particle together with a new allocated quadrant with the same size as the input quadrant. All of the children to the node is also set to `NULL`.

Another case is if the input quadtree node is an internal node, meaning that the node itself does not hold information of a single particle, but the children further down the tree structure have particles. In this case, the center of mass of the node, meaning the positions `cm_x` and `cm_y` and the total mass `mass_tot`, is updated to include the new particle to be inserted and afterwards moving the particle further down the tree with the function `insertQuadNode`.

The last case is if the input quadtree node is an external node, which means that the node holds

information of one single particle and all the children nodes are `NULL`. In this case, the node's already existing particle is given to a new pointer particle so that the particle of the node can be set to `NULL` and both the old and the new particle can move further down the quadtree to update the children of the node with the function `updateQuadNode`. Afterwards, the center of mass of the node is updated with the new added particle. This case also catches the error of having two particles in the exact same position.

2.4 insertQuadNode

The function `void insertQuadNode(quadtree_t** __restrict node, particle_t* __restrict particle)` finds in which quadrant the position of the input particle lies in order to insert the particle in the correct child of the input node. The first step is that the function creates a new quadrant with half the width and height of the quadrant of the input node, which is the correct size of the child nodes. Afterwards, the x and y positions of the particle are compared to the new width and height to find which new quadrant the particle lies in. When the quadrant is found, the origin of the created quadrant is set accordingly before calling the `insert` function again with the suitable child node, the input particle and the new quadrant. Afterwards, the allocated quadrant is freed.

2.5 deleteQuadtree

To delete a `quadtree` with all its children properly the function `void deleteQuadtree(quadtree_t **node)` is used. The function is recursively working its way down the tree by recalling the function with the child nodes as input. When the input node is zero the return call is used to stop the function, and when all functions have been called the `quadtree` and its assigned `quadrant` are freed and the `quadtree` node is set to `NULL`.

2.6 setOutput

The function `int setOutput(const char* __restrict filename, particle_t* __restrict data, const int N)` is implemented in a similarly way as `getInput`. The main task of this function is to write the result to the output file. However, the used function `fwrite` can use the created array of particles straight away since all values have been implemented in the correct order as the reference files. Thus, a temporary double array is not necessary to implement in the `setOutput` function.

2.7 calculateNewPositions

To calculate the particle positions over time the function `int calculateNewPositions(quadtree_t** root, particle_t* __restrict particles, const int N, const int nsteps, const double dt, const double theta_max))` was implemented. The function uses the given parameters and also the given value of the gravitational constant `G` to update the position and velocity values of parameters `root` and `particles`. The calculation of the new positions is made with the symplectic Euler time integration method, where the positions and velocities are calculated and updated `nsteps` number of times with the time step `dt`, as given in the input arguments.

Instead of using two loops to calculate the forces on one particle created by the other particles in the system, there is only one loop over `int i` where the function `calculateForce` is called to calculate the force exerted on each particle `i`. The force `F` is a double array and is calculated in the x and y directions, where `F[0]` represents the x direction and `F[0]` represents the y direction. To use the function `calculateForce`, a quadtree node is used, but because it is a recursive function which updates the node, the parameter `root` can not be used directly. Instead a new quadtree `root_new` is allocated and first given the same values as the input `root`, to be used as the input in `calculateForce`. When the forces have been calculated, they are multiplied with the gravitational constant and the mass

of the particle `i`. All calculations have been made according to equation (1). Afterwards, the accelerations are calculated and used to calculate the new velocities and positions, according to the equations in (2).

If the velocity and position values of the particle in `particle[i]` would be updated immediately it would affect the values of the remaining forces yet to be calculated, and give incorrect results. Therefore, another array of particles `particles_new` was allocated where the new position and velocity values are stored while the old values can continue to be used in calculations. When all calculations are done, the positions and the velocities of the parameter `particle_t* particles` are updated with the values from `particles_new`, so that it can be updated in the main function. Similarly, the new particle values have to be updated in the quadtree `root_new`. This is done by first calling the function `deleteQuadtree` to properly delete all nodes in `root_new` and afterwards inserting all new particles in `root_new` with the function `insert`. The `insert` function also needs an `quadrant` as input, which is allocated at the beginning of `calculateNewPositions` and updated to represent to whole simulated space before calling `insert`.

When all calculations are made, the input `root` is updated and all allocated variables `particles_new`, `root_new`, `F` and `quadrant` are freed. The function also includes two error checks, first if the new calculated positions of the particle exceeds the perimeter of the simulated space and secondly if the insert function returns an error.

2.8 calculateForce

The purpose of the function `void calculateForce(quadtree_t** __restrict node, particle_t* __restrict particle, double* __restrict F, const double theta_max)` is to calculate the force `F` exerted on the parameter `particle` from all other particles in the `quadtree_t` parameter `node`. To reduce the calculations, the value of θ (equation (3)) is calculated for each internal node and if the result is smaller or equal to the threshold value `theta_max`, the center of mass values are used to calculate the force instead of the particle values further down the tree. This is possible because the tree structure is following the Barnes-Hut algorithm.

`calculateForce` is a recursive function, where the first case is if the input node is `NULL` and the return function is called. Secondly, if the input node is an external node, the values of the distance vector `r` and the force in the x (`F[0]`) and y (`F[1]`) directions are calculated. The values of the force are added to the input parameter `F`. The calculations are also using the value of `epsilon` as given in the beginning of the function. However, if the x and y positions of the input particle and the particle of the node are the same, the particles are assumed to be the same and the return call is used instead of the calculations.

It is when the input node is an internal node that the value of θ is calculated in attempt to reduce the calculations. As explained above, if the resulting value is within the threshold, the calculation is treated as a single body problem. However, if θ exceeds the threshold `theta_max`, the calculations continue as a many body problem by recursively calling the function again for each of the four child nodes with the same particle, force vector `F` and threshold `theta_max`.

3 Performance and Discussion

In this section, the method of optimizing the serial code is presented as well as discussed. When optimizing the code, a Macbook Pro, Intel® Core™ i5-6267U CPU @ 2.90GHz together with compiler version *Clang-900.0.39.2* was used. The evaluation was done for the file *ellipse_N_03000.gal* with 200 time steps of 0.00001 seconds each and $\theta_{max} = 0.25$. Additionally, the time complexity is evaluated and discussed. Here, the performance is compared between two computers.

When the optimizations were made, the application Zoom (and possibly a few others) was running in the background, which could have had an impact on the time required for the code to run. However, the final result presented in `best_timing.txt` was obtained when no other applications were running in the background on the computer. To clarify, all optimizations and time measurements in this section were made without parallelization.

3.1 Code Optimizations

The time was measured for different compiler flags, as can be seen in Table 1, where the fastest compiler flag was *-O3 -march=native -ffast-math* and was therefore the one chosen. Then, *-funroll-loops*, the flag for for loop unrolling, was added but did not result in either an improvement or deterioration, so it was kept.

Table 1: *Performance difference by using different compiler flags in the Makefile. Measured in wall seconds. The quickest run time is marked in bold.*

Flags	**Time** $[s]$
-O0 (default)	30.104
-O1	7.260
-O2	7.438
-O3	7.280
-Ofast	7.448
-Os	7.435
-O3 -ffast-math	7.477
-O3 -march=native	7.387
-O3 -march=native -ffast-math	**7.067**
-O2 -march=native -ffast-math	7.125
-Ofast -march=native -ffast-math	7.148

The threshold value, θ_{max} was selected to be as large as possible, to speed up the code, but still keeping the maximum difference between the exact result and the computed result below 10^{-3}. This was done by testing different values of θ_{max} between 0.02 and 0.5 and evaluating the obtained results, and from this $\theta_{max} = 0.25$ turned out to be the optimal threshold value.

In order to reduce the amount of function calls, some optimization changes were tested. In the function `calculateForce`, the function `pow` was replaced with multiplication. This led to a small improvement in the code computation and therefore the change was kept. However, this change negatively affects the readability for the user. Also, the function `fabs` was replaced by the function `sqrt` together with multiplication. This change resulted in a slower code, which was expected since the number of function calls was the same as before and multiplication was needed. That change was therefore removed.

In order to save space, removal of unnecessary variables was tested. In the function `calculateForce`, the two variables for acceleration were removed. This change did not make any improvement on the code regarding time. Since there were only two doubles, meaning that the save in space is very little and removing these would affect the readability in an negative way, the change was not kept.

The keyword `__restrict` was added to the functions where more than one pointer was used as input and where it was guaranteed that they did not point at the same memory space.

"Structure packing" was also added by adding the line `__attribute__((__packed__))` in the declaration of the structs `particle_t` and `quadrant`, in order to minimize the size of the structs used in the code by letting the compiler remove the padding for the structs automatically.

3.2 Time Complexity

In Figure 2, the time consumption plotted as a function of N can be seen for the direct-sum algorithm. In Figure 3 the time consumption for the Barnes-Hut algorithm can be seen. Both algorithms are plotted on a linear scale in theses figures, and it can be seen that the direct-sum has a more quadratical appearance, while the plotted result for Barnes-Hut appears to have a more linear dependence (however, not perfectly linear). In Figure 4, the measured time results for the Barnes-Hut method are plotted in logarithmic scale and compared to the theoretical slope of $Nlog(N)$, $log(N)$ and N^2. This to demonstrate the results visually,and from this plot, it can be concluded that the time complexity for the Barnes-Hut algorithm is $\mathcal{O}(N \log N)$.

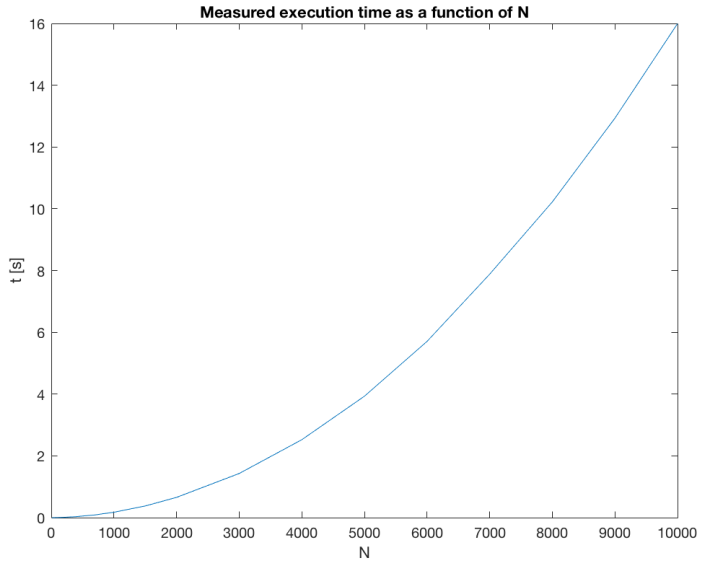

Figure 2: *The execution time plotted as a function of N for all available input data, for the $\mathcal{O}(N^2)$ case. $N \in [10, 10000]$.*

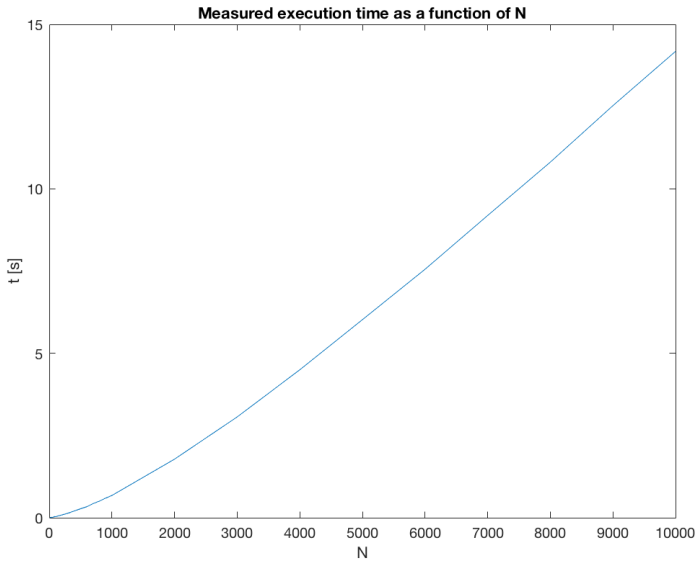

Figure 3: *The execution time plotted as a function of N for all available input data, for the Barnes-Hut case. $N \in [10, 10000]$.*

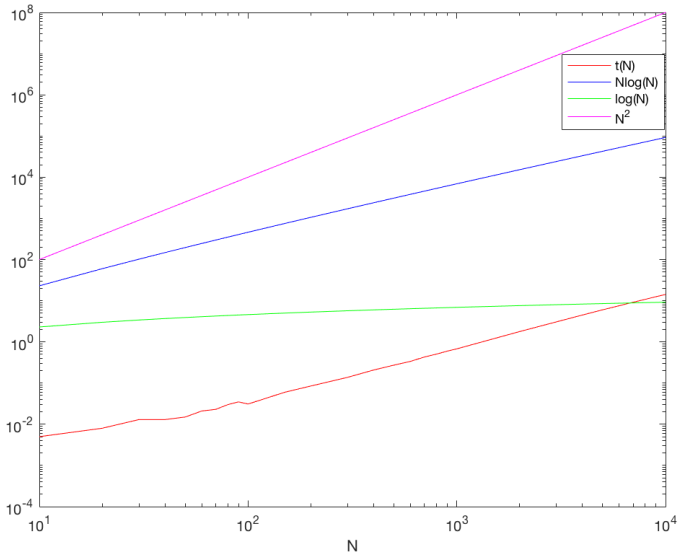

Figure 4: *Logarithmic plot of the execution time to analyze the time complexity. Here, the slope with logarithmic complexity, log(t(N)) (red), is compared to the slope of the functions: $Nlog(N)$ (blue), $log(N)$(green) and N^2 (pink).*

3.2.1 Barnes-Hut Efficiency

To analyze the efficiency of the implemented Barnes-Hut algorithm compared to the $\mathcal{O}(N^2)$ algorithm, the run time of the programs was measured for different numbers of particles N. Both programs were executed for 200 time steps of 0.00001 seconds. The Barnes-Hut algorithm was made with a threshold of $\theta_{max} = 0.25$.

The measurements were made on two computers with different CPU models and compiler versions. The results are displayed in Figure 2. For the first computer with the CPU model Intel® Core™ i5-3427U and compiler *Apple LLVM version 10.0.0 (clang-1000.10.44.4)* the Barnes-Hut algorithm showed a higher efficiency for 4000 particles and more. However, the computer with the CPU model Intel® Core™ i5-6267U and compiler *Apple LLVM version 9.0.0 (clang-900.0.39.2)* needed 10 000 particles before the Barnes-Hut algorithm was more efficient.

Table 2: *Comparison of run times for serial codes using two different algorithms, as well as for two different computers. Compiler versions and input data for the code specified in the paragraph above. The time is measured in wall seconds.*

	Intel® Core™ i5-3427U		Intel® Core™ i5-6267U	
N	**Direct-sum** [s]	**Barnes-Hut** [s]	**Direct-sum** [s]	**Barnes-Hut** [s]
2000	5.467	6.429	1.274	3.434
3000	11.832	11.845	2.961	5.963
4000	20.191	18.548	5.071	8.782
5000	31.559	24.919	7.876	11.746
10000	124.464	66.526	32.132	28.083

4 Parallelization Using Pthreads

In a previous assignment, the direct-sum algorithm for solving the same N-body problem was implemented. That algorithm had efficiency $\mathcal{O}(N^2)$ and the code for that implementation can be found in Appendix B. For the purpose of optimizing the mentioned code as well as the Barnes-Hut algorithm, developed in this assignment, even further, these implementations have been parallelized using Pthreads. The codes for these threaded programs can be found in Appendix C.

In order to parallelize the codes as efficiently as possible, an analysis of what parts of the codes that took the most time to run was made. By using the tool **gprof** it was found that the function `CalculateNewPositions` took up a large majority of the time required for the $\mathcal{O}(N^2)$ algorithm to run, and thus the Pthreads was implemented in that part of the program. To do this, every thread was assigned a certain amount of the particles of which to find the new positions of, and hence a large amount of the computations made in the function `CalculateNewPositions` could be done in parallel.

In the code for the Barnes hut algorithm it was found using **gprof** that most time was spent in the `calculateForce` function. Thus, these computations made by the `calculateForce` was parallelized using Phtreads. Similarily to the parallelization of the $\mathcal{O}(N^2)$ algorithm, each thread was assigned a certain amount of the particles of which to find the new positions of. Thus, much of the computations made by the function `calculateForce` could be done in parallel, which made the code more efficient.

When testing the performance of these threaded programs one of the university's Linux computers was used, running on an Intel® Xeon® E5520 @ 2.27GHz, with a total of 16 CPU:s (4 cores × 2 sockets, 2 threads per core), and compiler *gcc version 7.5.0 (Ubuntu 7.5.0-3ubuntu1 18.04)*. This was done to get more data points to evaluate the performance's dependence of the available threads. The codes were run with input file *ellipse_N_05000.gal* (which consists of 5000 particles), and 200 time steps of size 0.00001, together with $\theta_{max} = 0.25$.

In Figure 5 the time required for the code to run is shown as a function of the number of threads, for the two algorithms. It can be noted that the time reduces when a larger number of cores is used, for both algorithms. The curves correspond well to the expected result, since the time reduces in a $1/x$ like fashion, where x is the number of threads. When analyzing the time for when 16 threads was used however,

a small increase in the time can be noted when 7 threads were used. This result can be due to the fact that another person seemed to be using the same computer when running the simulations, thus occupying the last available core. It can also be observed that for the relatively large number of particles (N = 5000) the performance of the Barnes hut algorithm is significantly better than the $\mathcal{O}(N^2)$ algorithm.

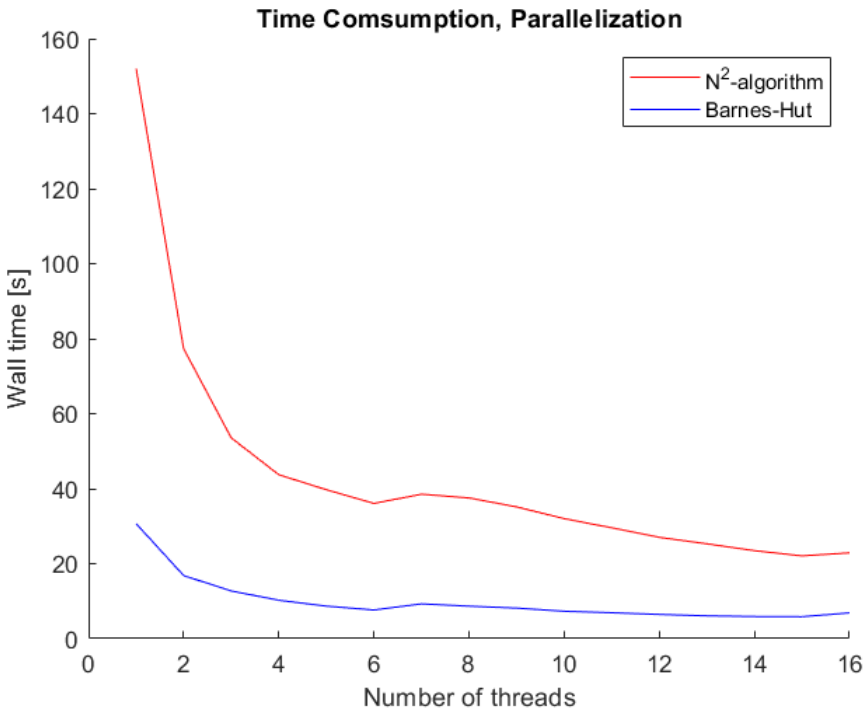

Figure 5: *Time required for the two parallelized algorithms to run depending on the number of threads, where the amount of threads range from 1-16. System information can be found in the paragraphs above. The Barnes-Hut algorithm has the complexity $Nlog(N)$.*

Appendix A

Listing 1: *Optimized galsim.c from Assignment 4*

```
#include <stdlib.h>
#include <math.h>
#include <stdio.h>
#include <sys/time.h>

typedef struct particle
{
  double pos_x;
  double pos_y;
  double mass;
  double vel_x;
  double vel_y;
  double brightness;
}__attribute__((__packed__)) particle_t;

typedef struct quadrant
{
  /* Position of specific quadrant in the whole space */
  double origin_x;
  double origin_y;

  double width;
  double height;
} __attribute__((__packed__)) quadrant_t;

typedef struct quadtree
{
  /* center of mass */
  double mass_tot;
  double cm_x;
  double cm_y;
  particle_t *particle;
  quadrant_t *quadrant;
  /* tree nodes */
  struct quadtree *top_left;
  struct quadtree *top_right;
  struct quadtree *bottom_left;
  struct quadtree *bottom_right;
}  quadtree_t;

int getInput(const char* __restrict filename, quadtree_t** __restrict root, particle_t* ...
    __restrict particles, const int N);
int insert(quadtree_t** __restrict node, particle_t* __restrict particle, quadrant_t* ...
    __restrict quadrant);
void insertQuadNode(quadtree_t** node, particle_t* __restrict particle);
void deleteQuadtree(quadtree_t **node);
void resetQuadtree(quadtree_t **node);
int setOutput(const char* __restrict filename, particle_t* __restrict data, const int N);
int calculateNewPositions(quadtree_t** __restrict root, particle_t* __restrict particles, ...
    const int N, const int nsteps, const double dt, const double theta_max);
void calculateForce(quadtree_t** __restrict node, particle_t* __restrict particle, double* ...
    __restrict F, const double theta_max);

/*
static double get_wall_seconds() {
  struct timeval tv;
  gettimeofday(&tv, NULL);
  double seconds = tv.tv_sec + (double)tv.tv_usec / 1000000;
  return seconds;
```

```
56  */
57
58  int main(int argc, char* argv[])
59  {
60    //double time1 = get_wall_seconds();
61    if (argc != 7)
62    {
63      printf("Check your input..\n");
64      return -1;
65    }
66    const int N = atoi(argv[1]);
67    const char* filename = argv[2];
68    const int nsteps = atoi(argv[3]);
69    const double dt = atof(argv[4]);
70    const double theta_max = atof(argv[5]); // theta_max = 0.25
71    //const int graphics = atoi(argv[6]);
72    const char* filename_out = "result.gal";
73    particle_t* particles = (particle_t*)malloc(N*sizeof(particle_t));
74    quadtree_t* root = NULL;
75
76    int err = getInput(filename, &root, particles, N);
77    if(err == -1)
78    {
79      deleteQuadtree(&root);
80      free(particles);
81      return -1;
82    }
83    err = calculateNewPositions(&root, particles, N, nsteps, dt, theta_max);
84    if(err == -1)
85    {
86      deleteQuadtree(&root);
87      free(particles);
88      return -1;
89    }
90
91    err = setOutput(filename_out, particles, N);
92    if(err == -1)
93    {
94      deleteQuadtree(&root);
95      free(particles);
96      return -1;
97    }
98
99    deleteQuadtree(&root);
100   free(particles);
101   //printf("galsim main took %7.3f wall seconds.\n", get_wall_seconds()-time1);
102   return 0;
103 }
104
105 int getInput(const char* __restrict filename, quadtree_t** __restrict root, particle_t* ...
        __restrict particles, const int N)
106 {
107   FILE *stream_in;
108   stream_in = fopen(filename,"rb");
109   double* arr = (double*)malloc(N*6*sizeof(double));
110   quadrant_t *quadrant = (quadrant_t*)malloc(sizeof(particle_t));
111
112   if(stream_in == NULL)
113   {
114     printf("Error: unable to open file: %s\n", filename);
115     fclose(stream_in);
116     free(arr);
117     free(quadrant);
118     return -1;
119   }
```

```
120   size_t input_size = N*6*sizeof(double);
121
122   /* Read input to data array */
123   size_t items_read = fread(arr, sizeof(char), input_size, stream_in);
124   int j = 0;
125   for (int i = 0; i < N*6; i += 6)
126   {
127     particles[j].pos_x = arr[i];
128     particles[j].pos_x = arr[i];
129     particles[j].pos_y = arr[i+1];
130     particles[j].mass = arr[i+2];
131     particles[j].vel_x = arr[i+3];
132     particles[j].vel_y = arr[i+4];
133     particles[j].brightness = arr[i+5];
134
135     quadrant->width = 1;
136     quadrant->height = 1;
137     quadrant->origin_x = 0;
138     quadrant->origin_y = 0;
139
140     insert(root, &particles[j], quadrant);
141     j++;
142   }
143
144   if (items_read != input_size)
145   {
146     printf("Error reading the input file.\n");
147     fclose(stream_in);
148     free(arr);
149     free(quadrant);
150     return -1;
151   }
152   free(quadrant);
153   fclose(stream_in);
154   free(arr);
155   return 1;
156 }
157
158 /* Note: we define origin in bottom-left corner! */
159 int insert(quadtree_t** node, particle_t* __restrict particle, quadrant_t* __restrict ...
        quadrant)
160 {
161   if(*node == NULL)
162   {
163     quadtree_t * new_node = (quadtree_t*)malloc(sizeof(quadtree_t));
164     quadrant_t * new_quadrant = (quadrant_t*)malloc(sizeof(quadrant_t));
165     new_node->particle = particle;
166     new_node->mass_tot = particle->mass;
167     new_node->cm_x = particle->pos_x;
168     new_node->cm_y = particle->pos_y;
169
170     new_quadrant->width = quadrant->width;
171     new_quadrant->height = quadrant->height;
172     new_quadrant->origin_x = quadrant->origin_x;
173     new_quadrant->origin_y = quadrant->origin_y;
174     new_node->quadrant = new_quadrant;
175
176     new_node->top_left = NULL;
177     new_node->top_right = NULL;
178     new_node->bottom_left = NULL;
179     new_node->bottom_right = NULL;
180     *node = new_node;
181   }
182   /*
183    * internal node: a node which already contains other nodes
```

```
184     * external node: a node which is initially empty - after its been filled it will only ...
            contian one particle
185     */
186   else if ((*node)->particle == NULL) // internal node
187   {
188     /* updating center of mass */
189     (*node)->cm_x = ((*node)->cm_x*(*node)->mass_tot + ...
            particle->pos_x*particle->mass)/((*node)->mass_tot + particle->mass);
190     (*node)->cm_y = ((*node)->cm_y*(*node)->mass_tot + ...
            particle->pos_y*particle->mass)/((*node)->mass_tot + particle->mass);
191     (*node)->mass_tot += particle->mass;
192     insertQuadNode(&(*node), particle);
193   }
194   else // external node: move the new and old nodes into a new lower quadrant
195   {
196     if((*node)->particle->pos_x == particle->pos_x && (*node)->particle->pos_y == ...
            particle->pos_y){
197       printf("Error: Particles at the same position \n");
198       return -1;
199     }
200     /* Making our node internal by removing the particle */
201     particle_t *particle_old = (*node)->particle;
202     (*node)->particle = NULL;
203     insertQuadNode(&(*node), particle);
204     insertQuadNode(&(*node), particle_old);
205
206     /* updating center of mass */
207     (*node)->cm_x = ((*node)->cm_x*(*node)->mass_tot + ...
            particle->pos_x*particle->mass)/((*node)->mass_tot + particle->mass);
208     (*node)->cm_y = ((*node)->cm_y*(*node)->mass_tot + ...
            particle->pos_y*particle->mass)/((*node)->mass_tot + particle->mass);
209     (*node)->mass_tot += particle->mass;
210   }
211   return 0;
212 }
213
214 void insertQuadNode(quadtree_t** __restrict node, particle_t* __restrict particle)
215 {
216   quadrant_t *quadrant_new = (quadrant_t*)malloc(sizeof(quadrant_t));
217   quadrant_new->width = (*node)->quadrant->width/2.0;
218   quadrant_new->height = (*node)->quadrant->height/2.0;
219
220   if(particle->pos_x < ((*node)->quadrant->origin_x + quadrant_new->width)) // left quadrant
221     {
222       if (particle->pos_y < ((*node)->quadrant->origin_y + quadrant_new->height)) // ...
              bottom left quadrant
223       {
224         quadrant_new->origin_y = (*node)->quadrant->origin_y;
225         quadrant_new->origin_x = (*node)->quadrant->origin_x;
226         insert(&(*node)->bottom_left, particle, quadrant_new);
227       }
228       else // top left quadrant
229       {
230         quadrant_new->origin_y = (*node)->quadrant->origin_y + quadrant_new->height;
231         quadrant_new->origin_x = (*node)->quadrant->origin_x;
232         insert(&(*node)->top_left, particle, quadrant_new);
233       }
234     }
235     else {  // right quadrant
236       if (particle->pos_y < ((*node)->quadrant->origin_y + quadrant_new->height)) // ...
              bottom right quadrant
237       {
238         quadrant_new->origin_x = (*node)->quadrant->origin_x + quadrant_new->width;
239         quadrant_new->origin_y = (*node)->quadrant->origin_y;
240         insert(&(*node)->bottom_right, particle, quadrant_new);
```

```
241         }
242         else // top right quadrant
243         {
244           quadrant_new->origin_x = (*node)->quadrant->origin_x + quadrant_new->width;
245           quadrant_new->origin_y = (*node)->quadrant->origin_y + quadrant_new->height;
246           insert(&(*node)->top_right, particle, quadrant_new);
247         }
248       }
249       free(quadrant_new);
250   }
251
252   void deleteQuadtree(quadtree_t **node)
253   {
254      if(*node == NULL){ // if empty node
255         return;
256       }
257
258     deleteQuadtree(&(*node)->top_left);
259     deleteQuadtree(&(*node)->top_right);
260     deleteQuadtree(&(*node)->bottom_left);
261     deleteQuadtree(&(*node)->bottom_right);
262
263     free((*node)->quadrant);
264     free(*node);
265     *node = NULL;
266   }
267
268   int setOutput(const char* __restrict filename, particle_t* __restrict data, const int N)
269   {
270     FILE *stream_out;
271     stream_out = fopen(filename,"wb");
272     if(stream_out == NULL)
273     {
274       printf("Error: unable to open file: %s\n", filename);
275       fclose(stream_out);
276       return -1;
277     }
278     fwrite(data, sizeof(particle_t), N, stream_out);
279     fclose(stream_out);
280     return 0;
281   }
282
283   int calculateNewPositions(quadtree_t** root, particle_t* __restrict particles, const int ...
       N, const int nsteps, const double dt, const double theta_max)
284   {
285     int err = 0;
286     double ax, ay;
287     const double G = 100.0/(double)(N);
288     double* F = (double*)malloc(2*sizeof(double));
289     particle_t *particles_new = (particle_t*)malloc(N*sizeof(particle_t));
290     quadrant_t *quadrant = (quadrant_t*)malloc(sizeof(quadrant_t));
291     quadtree_t **root_new = (quadtree_t**)malloc(sizeof(quadtree_t));
292     *root_new = *root;
293
294     for (int n = 0; n < nsteps; n++)
295     {
296       for(int i = 0; i < N; i++)
297       {
298         F[0] = 0;
299         F[1] = 0;
300
301         calculateForce(root_new, &particles[i], F, theta_max);
302
303         F[0] = -G*particles[i].mass*F[0];
304         F[1] = -G*particles[i].mass*F[1];
```

```
305
306         /* Acceleration */
307         ax = F[0]/(particles[i].mass);
308         ay = F[1]/(particles[i].mass);
309         /* Updating velocities */
310         particles_new[i].vel_x = particles[i].vel_x + dt*ax;
311         particles_new[i].vel_y = particles[i].vel_y + dt*ay;
312         /* Updating positions */
313         particles_new[i].pos_x = particles[i].pos_x + dt* particles_new[i].vel_x;
314         particles_new[i].pos_y = particles[i].pos_y + dt* particles_new[i].vel_y;
315         if(particles_new[i].pos_x >1|| particles_new[i].pos_x <0 || particles_new[i].pos_y ...
                >1|| particles_new[i].pos_y <0){
316           printf("Error: Particle outside interval \n");
317           free(particles_new);
318           free(F);
319           free(root_new);
320           free(quadrant);
321           return -1;
322         }
323       }
324       deleteQuadtree(root_new);
325
326       for (int i = 0; i < N; i++)
327       {
328         particles[i].pos_x = particles_new[i].pos_x;
329         particles[i].pos_y = particles_new[i].pos_y;
330         particles[i].vel_x = particles_new[i].vel_x;
331         particles[i].vel_y = particles_new[i].vel_y;
332
333         quadrant->width = 1;
334         quadrant->height = 1;
335         quadrant->origin_x = 0;
336         quadrant->origin_y = 0;
337         err += insert(root_new, &particles[i], quadrant);
338       }
339       if(err!=0)
340       {
341         free(particles_new);
342         free(F);
343         free(root_new);
344         free(quadrant);
345         return -1;
346       }
347     }
348     *root = *root_new;
349     free(particles_new);
350     free(root_new);
351     free(F);
352     free(quadrant);
353     return 0;
354   }
355
356   void calculateForce(quadtree_t** __restrict node, particle_t* __restrict particle, double* ...
          __restrict F, const double theta_max)
357   {
358     double r, theta;
359     const double epsilon = 0.001;
360     /* F[0] = Fx, F[1] = Fy */
361     if ((*node) == NULL)
362     {
363       return;
364     }
365     /* External node */
366     else if ((*node)->particle != NULL)
367     {
```

```
368       /* Same particle */
369       if ((fabs((*node)->particle->pos_x - particle->pos_x) < 0.0000001) && ...
              (fabs((*node)->particle->pos_y - particle->pos_y) < 0.0000001))
370       {
371         return;
372       }
373       /* x_i: particle, x_j: node->particle */
374       r = sqrt((particle->pos_x - (*node)->particle->pos_x)*(particle->pos_x - ...
              (*node)->particle->pos_x) + (particle->pos_y - ...
              (*node)->particle->pos_y)*(particle->pos_y - (*node)->particle->pos_y));
375       F[0] += (*node)->particle->mass*(particle->pos_x - ...
              (*node)->particle->pos_x)/((r+epsilon)*(r+epsilon)*(r+epsilon));
376       F[1] += (*node)->particle->mass*(particle->pos_y - ...
              (*node)->particle->pos_y)/((r+epsilon)*(r+epsilon)*(r+epsilon));
377     }
378     /* Internal node */
379     else
380     {
381       /* Calculate ratio */
382       theta = (*node)->quadrant->width/sqrt((particle->pos_x - ((*node)->quadrant->origin_x ...
              + (*node)->quadrant->width/2))*(particle->pos_x - ((*node)->quadrant->origin_x + ...
              (*node)->quadrant->width/2)) + (particle->pos_y - ((*node)->quadrant->origin_y + ...
              (*node)->quadrant->height/2))*(particle->pos_y - ((*node)->quadrant->origin_y + ...
              (*node)->quadrant->height/2)));
383
384       /* Single body problem */
385       if (theta ≤ theta_max)
386       {
387         /* x_i: particle, x_j: node->cm */
388         r = sqrt((particle->pos_x - (*node)->cm_x)*(particle->pos_x - (*node)->cm_x) + ...
                (particle->pos_y - (*node)->cm_y)*(particle->pos_y - (*node)->cm_y));
389         F[0] += (*node)->mass_tot*(particle->pos_x - ...
                (*node)->cm_x)/((r+epsilon)*(r+epsilon)*(r+epsilon));
390         F[1] += (*node)->mass_tot*(particle->pos_y - ...
                (*node)->cm_y)/((r+epsilon)*(r+epsilon)*(r+epsilon));
391       }
392       /* Many body problem */
393       else
394       {
395         calculateForce(&(*node)->top_left, particle, F, theta_max);
396         calculateForce(&(*node)->top_right, particle, F, theta_max);
397         calculateForce(&(*node)->bottom_left, particle, F, theta_max);
398         calculateForce(&(*node)->bottom_right, particle, F, theta_max);
399       }
400     }
401   }
```

Appendix B

Listing 2: *Optimized galsim.c from Assignment 3*

```c
#include <stdlib.h>
#include <math.h>
#include <stdio.h>
#include <sys/time.h>

typedef struct particle
{
  double pos_x;
  double pos_y;
  double mass;
  double vel_x;
  double vel_y;
  double brightness;
}particle_t;

int getInput(const char* __restrict filename, particle_t* __restrict data, const int N);
int setOutput(const char* __restrict filename, particle_t* __restrict data, const int N);
void calculateNewPositions(particle_t* data, const int N, const int nsteps, const double dt);

static double get_wall_seconds() {
  struct timeval tv;
  gettimeofday(&tv, NULL);
  double seconds = tv.tv_sec + (double)tv.tv_usec / 1000000;
  return seconds;
}

int main(int argc, char* argv[])
{
  double time1 = get_wall_seconds();
  if (argc != 6)
  {
    printf("Check your input..\n");
    return -1;
  }
  const int N = atoi(argv[1]);
  const char* filename = argv[2];
  const int nsteps = atoi(argv[3]);
  const double dt = atof(argv[4]);
  //const int graphics = atoi(argv[5]);
  const char* filename_out = "result.gal";
  particle_t* data = (particle_t*)malloc(N*sizeof(particle_t));

  int err = getInput(filename, data, N);
  if(err == -1)
  {
    return -1;
  }
  calculateNewPositions(data, N, nsteps, dt);

  err = setOutput(filename_out, data, N);
  if(err == -1)
  {
    return -1;
  }

  free(data);
  printf("galsim main took %7.3f wall seconds.\n", get_wall_seconds()-time1);
  return 0;
}
```

```
60
61  int getInput(const char* __restrict filename, particle_t* __restrict data, const int N)
62  {
63    FILE *stream_in;
64    stream_in = fopen(filename,"rb");
65    double* arr = (double*)malloc(N*6*sizeof(double));
66
67    if(stream_in == NULL)
68    {
69      printf("Error: unable to open file: %s\n", filename);
70      return -1;
71    }
72    size_t input_size = N*6*sizeof(double);
73
74    /* Read input to data array */
75    size_t items_read = fread(arr, sizeof(char), input_size, stream_in);
76    int j = 0;
77    for (int i = 0; i < N*6; i += 6)
78    {
79      data[j].pos_x = arr[i];
80      data[j].pos_y = arr[i+1];
81      data[j].mass = arr[i+2];
82      data[j].vel_x = arr[i+3];
83      data[j].vel_y = arr[i+4];
84      data[j].brightness = arr[i+5];
85      j++;
86    }
87
88    if (items_read != input_size)
89    {
90      printf("Error reading the input file.\n");
91      return -1;
92    }
93
94    fclose(stream_in);
95    free(arr);
96    return 1;
97  }
98
99  int setOutput(const char* __restrict filename, particle_t* __restrict data, const int N)
100 {
101   FILE *stream_out;
102   stream_out = fopen(filename,"wb");
103   if(stream_out == NULL)
104   {
105     printf("Error: unable to open file: %s\n", filename);
106     return -1;
107   }
108   fwrite(data, sizeof(particle_t), N, stream_out);
109   fclose(stream_out);
110   return 0;
111 }
112
113 void calculateNewPositions(particle_t* particles, const int N, const int nsteps, const ...
        double dt)
114 {
115   double Fx, Fy,r, ax, ay;
116   const double G = 100/(double)(N);
117   const double epsilon = 0.001;
118   particle_t *particles_new = (particle_t*)malloc(N*sizeof(particle_t));
119   for (int n = 0; n < nsteps; n++)
120   {
121     for(int i = 0; i < N; i++)
122     {
123       Fx = 0;
```

```
124         Fy = 0;
125         for(int j = 0; j < i; j++)
126         {
127             r = sqrt((particles[i].pos_x - particles[j].pos_x)*(particles[i].pos_x - ...
                    particles[j].pos_x) + (particles[i].pos_y - ...
                    particles[j].pos_y)*(particles[i].pos_y - particles[j].pos_y));
128             Fx += particles[j].mass*(particles[i].pos_x - ...
                    particles[j].pos_x)/((r+epsilon)*(r+epsilon)*(r+epsilon));
129             Fy += particles[j].mass*(particles[i].pos_y - ...
                    particles[j].pos_y)/((r+epsilon)*(r+epsilon)*(r+epsilon));
130         }
131         for(int j = i+1; j < N; j++)
132         {
133             r = sqrt((particles[i].pos_x - particles[j].pos_x)*(particles[i].pos_x - ...
                    particles[j].pos_x) + (particles[i].pos_y - ...
                    particles[j].pos_y)*(particles[i].pos_y - particles[j].pos_y));
134             Fx += particles[j].mass*(particles[i].pos_x - ...
                    particles[j].pos_x)/((r+epsilon)*(r+epsilon)*(r+epsilon));
135             Fy += particles[j].mass*(particles[i].pos_y - ...
                    particles[j].pos_y)/((r+epsilon)*(r+epsilon)*(r+epsilon));
136         }
137         /* Forces */
138         Fx *= -G*particles[i].mass;
139         Fy *= -G*particles[i].mass;
140         /* Acceleration */
141         ax = Fx/(particles[i].mass);
142         ay = Fy/(particles[i].mass);
143         /* Updating velocities */
144         particles_new[i].vel_x = particles[i].vel_x + dt*ax;
145         particles_new[i].vel_y = particles[i].vel_y + dt*ay;
146         /* Updating positions */
147         particles_new[i].pos_x = particles[i].pos_x + dt* particles_new[i].vel_x;
148         particles_new[i].pos_y = particles[i].pos_y + dt* particles_new[i].vel_y;
149       }
150       for (int i = 0; i < N; i++)
151       {
152         particles[i].pos_x = particles_new[i].pos_x;
153         particles[i].pos_y = particles_new[i].pos_y;
154         particles[i].vel_x = particles_new[i].vel_x;
155         particles[i].vel_y = particles_new[i].vel_y;
156       }
157     }
158     free(particles_new);
159   }
```

Appendix C

Listing 3: *Parallelized galsim.c from Assignment 3*

```c
#include <stdlib.h>
#include <math.h>
#include <stdio.h>
#include <sys/time.h>
#include <pthread.h>

typedef struct particle
{
  double pos_x;
  double pos_y;
  double mass;
  double vel_x;
  double vel_y;
  double brightness;
}particle_t;

typedef struct dataForThread
{
  int start;
  int stop;
  int N;
  double dt;
  pthread_t thread;
}dataForThread_t;

/* Global arrays */
particle_t* particles;
particle_t* particles_new;
int rest;

int getInput(const char* __restrict filename, const int N);
int setOutput(const char* __restrict filename, const int N);
void calculateNewPositions(const int N, const int nsteps, const double dt, int ...
    num_of_threads);
void* thread_func(void* arg);

static double get_wall_seconds() {
  struct timeval tv;
  gettimeofday(&tv, NULL);
  double seconds = tv.tv_sec + (double)tv.tv_usec / 1000000;
  return seconds;
}

int main(int argc, char* argv[])
{
  double time1 = get_wall_seconds();
  if (argc != 8)
  {
    printf("Expected input: ./galsim_A3 N filename nsteps Δ_t theta_max graphics ...
        n_threads\n");
    return -1;
  }
  const int N = atoi(argv[1]);
  const char* filename = argv[2];
  const int nsteps = atoi(argv[3]);
  const double dt = atof(argv[4]);
  //const double theta_max = atof(argv[5]);
  //const int graphics = atoi(argv[6]);
  const int num_of_threads = atoi(argv[7]);
```

```
58
59    rest = N%num_of_threads;
60
61    const char* filename_out = "result.gal";
62    particles = (particle_t*)malloc(N*sizeof(particle_t));
63
64    int err = getInput(filename, N);
65    if(err == -1)
66    {
67      return -1;
68    }
69
70    calculateNewPositions(N, nsteps, dt, num_of_threads);
71
72    err = setOutput(filename_out, N);
73    if(err == -1)
74    {
75      return -1;
76    }
77
78    free(particles);
79    printf("galsim main took %7.3f wall seconds.\n", get_wall_seconds()-time1);
80    return 0;
81  }
82
83  int getInput(const char* __restrict filename, const int N)
84  {
85    FILE *stream_in;
86    stream_in = fopen(filename,"rb");
87    double* arr = (double*)malloc(N*6*sizeof(double));
88
89    if(stream_in == NULL)
90    {
91      printf("Error: unable to open file: %s\n", filename);
92      return -1;
93    }
94    size_t input_size = N*6*sizeof(double);
95
96    /* Read input to data array */
97    size_t items_read = fread(arr, sizeof(char), input_size, stream_in);
98    int j = 0;
99    for (int i = 0; i < N*6; i += 6)
100   {
101     particles[j].pos_x = arr[i];
102     particles[j].pos_y = arr[i+1];
103     particles[j].mass = arr[i+2];
104     particles[j].vel_x = arr[i+3];
105     particles[j].vel_y = arr[i+4];
106     particles[j].brightness = arr[i+5];
107     j++;
108   }
109
110   if (items_read != input_size)
111   {
112     printf("Error reading the input file.\n");
113     return -1;
114   }
115
116   fclose(stream_in);
117   free(arr);
118   return 1;
119 }
120
121 int setOutput(const char* __restrict filename, const int N)
122 {
```

```
123   FILE *stream_out;
124   stream_out = fopen(filename,"wb");
125   if(stream_out == NULL)
126   {
127     printf("Error: unable to open file: %s\n", filename);
128     return -1;
129   }
130   fwrite(particles, sizeof(particle_t), N, stream_out);
131   fclose(stream_out);
132   return 0;
133 }
134
135 void calculateNewPositions(const int N, const int nsteps, const double dt, int num_of_threads)
136 {
137   dataForThread_t threads[num_of_threads];
138   particles_new = (particle_t*)malloc(N*sizeof(particle_t));
139   for (int n = 0; n < nsteps; n++)
140   {
141     int work_size = N/num_of_threads; // number of particles for every thread
142     /* Create threads */
143     for(int i = 0; i < num_of_threads-1; i++){
144       threads[i].start = (i+1) * work_size - work_size;
145       threads[i].stop = threads[i].start + work_size;
146       threads[i].N = N;
147       threads[i].dt = dt;
148
149       pthread_create(&(threads[i].thread), NULL, thread_func, &threads[i]);
150     }
151     /* Create last thread (with possibly different work_size) */
152     threads[num_of_threads-1].start = num_of_threads * work_size - work_size;
153     threads[num_of_threads-1].stop = threads[num_of_threads-1].start + work_size + rest;
154     threads[num_of_threads-1].N = N;
155     threads[num_of_threads-1].dt = dt;
156     pthread_create(&(threads[num_of_threads-1].thread), NULL, thread_func, ...
          &threads[num_of_threads-1]);
157
158     /* Join threads */
159     for(int i = 0; i < num_of_threads; i++){
160       pthread_join(threads[i].thread, NULL);
161     }
162
163     for (int i = 0; i < N; i++)
164     {
165       particles[i].pos_x = particles_new[i].pos_x;
166       particles[i].pos_y = particles_new[i].pos_y;
167       particles[i].vel_x = particles_new[i].vel_x;
168       particles[i].vel_y = particles_new[i].vel_y;
169     }
170   }
171   free(particles_new);
172 }
173
174 void* thread_func(void* arg){
175   dataForThread_t* info = (dataForThread_t *) arg;
176
177   double Fx, Fy, r, ax, ay;
178   const double G = 100/(double)(info->N);
179   const double epsilon = 0.001;
180   /* Compute new position of particle */
181   for(int i = info->start; i<info->stop; i++){ // for specific particles
182     Fx = 0;
183     Fy = 0;
184     for(int j = 0; j<i; j++){ // compare with every other particle
185       /* Distance between particles */
```

```
186         r = sqrt((particles[i].pos_x - particles[j].pos_x)*(particles[i].pos_x - ...
                particles[j].pos_x) + (particles[i].pos_y - ...
                particles[j].pos_y)*(particles[i].pos_y - particles[j].pos_y));
187
188         /* Forces*/
189         Fx += particles[j].mass*(particles[i].pos_x - ...
                particles[j].pos_x)/((r+epsilon)*(r+epsilon)*(r+epsilon));
190         Fy += particles[j].mass*(particles[i].pos_y - ...
                particles[j].pos_y)/((r+epsilon)*(r+epsilon)*(r+epsilon));
191       }
192       for(int j = i+1; j < info->N; j++){
193         /* Distance between particles */
194         r = sqrt((particles[i].pos_x - particles[j].pos_x)*(particles[i].pos_x - ...
                particles[j].pos_x) + (particles[i].pos_y - ...
                particles[j].pos_y)*(particles[i].pos_y - particles[j].pos_y));
195
196         /* Forces*/
197         Fx += particles[j].mass*(particles[i].pos_x - ...
                particles[j].pos_x)/((r+epsilon)*(r+epsilon)*(r+epsilon));
198         Fy += particles[j].mass*(particles[i].pos_y - ...
                particles[j].pos_y)/((r+epsilon)*(r+epsilon)*(r+epsilon));
199       }
200       /* Forces */
201       Fx *= -G * particles[i].mass;
202       Fy *= -G * particles[i].mass;
203       /* Acceleration */
204       ax = Fx/(particles[i].mass);
205       ay = Fy/(particles[i].mass);
206       /* Updating velocities */
207       particles_new[i].vel_x = particles[i].vel_x + info->dt * ax;
208       particles_new[i].vel_y = particles[i].vel_y + info->dt * ay;
209       /* Updating positions */
210       particles_new[i].pos_x = particles[i].pos_x + (double)info->dt * particles_new[i].vel_x;
211       particles_new[i].pos_y = particles[i].pos_y + (double)info->dt * particles_new[i].vel_y;
212     }
213     return NULL;
214   }
```

Listing 4: *Parallelized galsim.c from Assignment 4*

```
1   #include <stdlib.h>
2   #include <math.h>
3   #include <stdio.h>
4   #include <sys/time.h>
5   #include <pthread.h>
6
7   typedef struct particle
8   {
9     double pos_x;
10    double pos_y;
11    double mass;
12    double vel_x;
13    double vel_y;
14    double brightness;
15  }__attribute__((__packed__)) particle_t;
16
17  typedef struct quadrant
18  {
19    /* Position of specific quadrant in the whole space */
20    double origin_x;
21    double origin_y;
22
23    double width;
```

```
24    double height;
25  } __attribute__((__packed__)) quadrant_t;
26
27  typedef struct quadtree
28  {
29    /* center of mass */
30    double mass_tot;
31    double cm_x;
32    double cm_y;
33    particle_t *particle;
34    quadrant_t *quadrant;
35    /* tree nodes */
36    struct quadtree *top_left;
37    struct quadtree *top_right;
38    struct quadtree *bottom_left;
39    struct quadtree *bottom_right;
40  } quadtree_t;
41
42  typedef struct dataForThread
43  {
44    int start;
45    int stop;
46    int N;
47    double dt;
48    double theta_max;
49    pthread_t thread;
50    quadtree_t** root;
51  }dataForThread_t;
52
53  /* Global arrays */
54  particle_t* particles;
55  particle_t* particles_new;
56  int rest;
57
58  int getInput(const char* __restrict filename, quadtree_t** __restrict root, const int N);
59  int insert(quadtree_t** __restrict node, particle_t* __restrict particle, quadrant_t* ...
        __restrict quadrant);
60  void insertQuadNode(quadtree_t** node, particle_t* __restrict particle);
61  void deleteQuadtree(quadtree_t **node);
62  int setOutput(const char* __restrict filename, const int N);
63  int calculateNewPositions(quadtree_t** __restrict root, const int N, const int nsteps, ...
        const double dt, const double theta_max, int num_of_threads);
64  void calculateForce(quadtree_t** __restrict node, particle_t* __restrict particle, double* ...
        __restrict F, const double theta_max);
65  void* thread_func(void* arg);
66
67  static double get_wall_seconds() {
68    struct timeval tv;
69    gettimeofday(&tv, NULL);
70    double seconds = tv.tv_sec + (double)tv.tv_usec / 1000000;
71    return seconds;
72  }
73
74  int main(int argc, char* argv[])
75  {
76    double time1 = get_wall_seconds();
77    if (argc != 8)
78    {
79      printf("Expected input: ./galsim_A4 N filename nsteps Δ_t theta_max graphics ...
            n_threads\n");
80      return -1;
81    }
82    const int N = atoi(argv[1]);
83    const char* filename = argv[2];
84    const int nsteps = atoi(argv[3]);
```

```
85    const double dt = atof(argv[4]);
86    const double theta_max = atof(argv[5]); // theta_max = 0.25
87    //const int graphics = atoi(argv[6]);
88    const int num_of_threads = atoi(argv[7]);
89
90    rest = N%num_of_threads;
91
92    const char* filename_out = "result.gal";
93
94    particles = (particle_t*)malloc(N*sizeof(particle_t));
95    quadtree_t* root = NULL;
96
97    int err = getInput(filename, &root, N);
98    if(err == -1)
99    {
100     deleteQuadtree(&root);
101     free(particles);
102     return -1;
103   }
104
105   err = calculateNewPositions(&root, N, nsteps, dt, theta_max, num_of_threads);
106   if(err == -1)
107   {
108     deleteQuadtree(&root);
109     free(particles);
110     return -1;
111   }
112
113   err = setOutput(filename_out, N);
114   if(err == -1)
115   {
116     deleteQuadtree(&root);
117     free(particles);
118     return -1;
119   }
120
121   deleteQuadtree(&root);
122   free(particles);
123   printf("galsim main took %7.3f wall seconds.\n", get_wall_seconds()-time1);
124   return 0;
125 }
126
127 int getInput(const char* __restrict filename, quadtree_t** __restrict root, const int N)
128 {
129   FILE *stream_in;
130   stream_in = fopen(filename,"rb");
131   double* arr = (double*)malloc(N*6*sizeof(double));
132   quadrant_t *quadrant = (quadrant_t*)malloc(sizeof(particle_t));
133
134   if(stream_in == NULL)
135   {
136     printf("Error: unable to open file: %s\n", filename);
137     fclose(stream_in);
138     free(arr);
139     free(quadrant);
140     return -1;
141   }
142   size_t input_size = N*6*sizeof(double);
143
144   /* Read input to data array */
145   size_t items_read = fread(arr, sizeof(char), input_size, stream_in);
146   int j = 0;
147   for (int i = 0; i < N*6; i += 6)
148   {
149     particles[j].pos_x = arr[i];
```

```
150       particles[j].pos_x = arr[i];
151       particles[j].pos_y = arr[i+1];
152       particles[j].mass = arr[i+2];
153       particles[j].vel_x = arr[i+3];
154       particles[j].vel_y = arr[i+4];
155       particles[j].brightness = arr[i+5];
156
157       quadrant->width = 1;
158       quadrant->height = 1;
159       quadrant->origin_x = 0;
160       quadrant->origin_y = 0;
161
162       insert(root, &particles[j], quadrant);
163       j++;
164     }
165
166     if (items_read != input_size)
167     {
168       printf("Error reading the input file.\n");
169       fclose(stream_in);
170       free(arr);
171       free(quadrant);
172       return -1;
173     }
174
175     free(quadrant);
176     fclose(stream_in);
177     free(arr);
178     return 1;
179   }
180
181   int insert(quadtree_t** node, particle_t* __restrict particle, quadrant_t* __restrict ...
          quadrant)
182   {
183     if(*node == NULL)
184     {
185       quadtree_t * new_node = (quadtree_t*)malloc(sizeof(quadtree_t));
186       quadrant_t * new_quadrant = (quadrant_t*)malloc(sizeof(quadrant_t));
187       new_node->particle = particle;
188       new_node->mass_tot = particle->mass;
189       new_node->cm_x = particle->pos_x;
190       new_node->cm_y = particle->pos_y;
191
192       new_quadrant->width = quadrant->width;
193       new_quadrant->height = quadrant->height;
194       new_quadrant->origin_x = quadrant->origin_x;
195       new_quadrant->origin_y = quadrant->origin_y;
196       new_node->quadrant = new_quadrant;
197
198       new_node->top_left = NULL;
199       new_node->top_right = NULL;
200       new_node->bottom_left = NULL;
201       new_node->bottom_right = NULL;
202       *node = new_node;
203     }
204     else if ((*node)->particle == NULL) // internal node ("stores" several particles)
205     {
206       /* updating center of mass */
207       (*node)->cm_x = ((*node)->cm_x*(*node)->mass_tot + ...
              particle->pos_x*particle->mass)/((*node)->mass_tot + particle->mass);
208       (*node)->cm_y = ((*node)->cm_y*(*node)->mass_tot + ...
              particle->pos_y*particle->mass)/((*node)->mass_tot + particle->mass);
209       (*node)->mass_tot += particle->mass;
210       insertQuadNode(&(*node), particle);
211     }
```

```
212   else // external node (node with only one particle)
213   {
214     if((*node)->particle->pos_x == particle->pos_x && (*node)->particle->pos_y == ...
          particle->pos_y){
215       printf("Error: Particles at the same position \n");
216       return -1;
217     }
218     /* Making our node internal by removing the particle */
219     particle_t *particle_old = (*node)->particle;
220     (*node)->particle = NULL;
221     insertQuadNode(&(*node), particle);
222     insertQuadNode(&(*node), particle_old);
223
224     /* updating center of mass */
225     (*node)->cm_x = ((*node)->cm_x*(*node)->mass_tot + ...
          particle->pos_x*particle->mass)/((*node)->mass_tot + particle->mass);
226     (*node)->cm_y = ((*node)->cm_y*(*node)->mass_tot + ...
          particle->pos_y*particle->mass)/((*node)->mass_tot + particle->mass);
227     (*node)->mass_tot += particle->mass;
228   }
229   return 0;
230 }
231
232 void insertQuadNode(quadtree_t** __restrict node, particle_t* __restrict particle)
233 {
234   quadrant_t *quadrant_new = (quadrant_t*)malloc(sizeof(quadrant_t));
235   quadrant_new->width = (*node)->quadrant->width/2.0;
236   quadrant_new->height = (*node)->quadrant->height/2.0;
237
238   if(particle->pos_x < ((*node)->quadrant->origin_x + quadrant_new->width)) // left quadrant
239     {
240       if (particle->pos_y < ((*node)->quadrant->origin_y + quadrant_new->height)) // ...
            bottom left quadrant
241       {
242         quadrant_new->origin_y = (*node)->quadrant->origin_y;
243         quadrant_new->origin_x = (*node)->quadrant->origin_x;
244         insert(&(*node)->bottom_left, particle, quadrant_new);
245       }
246       else // top left quadrant
247       {
248         quadrant_new->origin_y = (*node)->quadrant->origin_y + quadrant_new->height;
249         quadrant_new->origin_x = (*node)->quadrant->origin_x;
250         insert(&(*node)->top_left, particle, quadrant_new);
251       }
252     }
253     else {  // right quadrant
254       if (particle->pos_y < ((*node)->quadrant->origin_y + quadrant_new->height)) // ...
            bottom right quadrant
255       {
256         quadrant_new->origin_x = (*node)->quadrant->origin_x + quadrant_new->width;
257         quadrant_new->origin_y = (*node)->quadrant->origin_y;
258         insert(&(*node)->bottom_right, particle, quadrant_new);
259       }
260       else // top right quadrant
261       {
262         quadrant_new->origin_x = (*node)->quadrant->origin_x + quadrant_new->width;
263         quadrant_new->origin_y = (*node)->quadrant->origin_y + quadrant_new->height;
264         insert(&(*node)->top_right, particle, quadrant_new);
265       }
266     }
267     free(quadrant_new);
268 }
269
270 void deleteQuadtree(quadtree_t **node)
271 {
```

```
272     if(*node == NULL){ // if empty node
273        return;
274     }
275
276   deleteQuadtree(&(*node)->top_left);
277   deleteQuadtree(&(*node)->top_right);
278   deleteQuadtree(&(*node)->bottom_left);
279   deleteQuadtree(&(*node)->bottom_right);
280
281   free((*node)->quadrant);
282   free(*node);
283   *node = NULL;
284 }
285
286 int setOutput(const char* __restrict filename, const int N)
287 {
288   FILE *stream_out;
289   stream_out = fopen(filename,"wb");
290   if(stream_out == NULL)
291   {
292     printf("Error: unable to open file: %s\n", filename);
293     fclose(stream_out);
294     return -1;
295   }
296   fwrite(particles, sizeof(particle_t), N, stream_out);
297   fclose(stream_out);
298   return 0;
299 }
300
301 int calculateNewPositions(quadtree_t** root, const int N, const int nsteps, const double ...
      dt, const double theta_max, int num_of_threads)
302 {
303   int err = 0;
304   particles_new = (particle_t*)malloc(N*sizeof(particle_t));
305   quadrant_t* quadrant = (quadrant_t*)malloc(sizeof(quadrant_t));
306
307   quadtree_t **root_new = (quadtree_t**)malloc(sizeof(quadtree_t));
308   *root_new = *root;
309
310   dataForThread_t threads[num_of_threads];
311   for (int n = 0; n < nsteps; n++)
312   {
313     int work_size = N/num_of_threads; // number of particles for every thread
314     /* Create threads */
315     for(int i = 0; i < num_of_threads-1; i++){
316       threads[i].start = (i+1) * work_size - work_size;
317       threads[i].stop = threads[i].start + work_size;
318       threads[i].N = N;
319       threads[i].dt = dt;
320       threads[i].theta_max = theta_max;
321       threads[i].root = root_new;
322
323       pthread_create(&(threads[i].thread), NULL, thread_func, &threads[i]);
324     }
325     /* Create last thread (with possibly different work_size) */
326     threads[num_of_threads-1].start = num_of_threads * work_size - work_size;
327     threads[num_of_threads-1].stop = threads[num_of_threads-1].start + work_size + rest;
328     threads[num_of_threads-1].N = N;
329     threads[num_of_threads-1].dt = dt;
330     threads[num_of_threads-1].theta_max = theta_max;
331     threads[num_of_threads-1].root = root_new;
332     pthread_create(&(threads[num_of_threads-1].thread), NULL, thread_func, ...
          &threads[num_of_threads-1]);
333
334     /* Join threads */
```

```
335       for(int i = 0; i < num_of_threads; i++){
336         pthread_join(threads[i].thread, NULL);
337       }
338
339       /* Check that particles are within interval */
340       for(int i = 0; i < N; i++){
341         if(particles_new[i].pos_x >1 || particles_new[i].pos_x <0 || particles_new[i].pos_y ...
              >1 || particles_new[i].pos_y <0){
342           printf("Error: Particle outside interval\n");
343           free(particles_new);
344           free(root_new);
345           free(quadrant);
346           return -1;
347         }
348       }
349
350       deleteQuadtree(root_new); // reset tree
351
352       /* Update particle positions and refill tree */
353       for (int i = 0; i < N; i++)
354       {
355         particles[i].pos_x = particles_new[i].pos_x;
356         particles[i].pos_y = particles_new[i].pos_y;
357         particles[i].vel_x = particles_new[i].vel_x;
358         particles[i].vel_y = particles_new[i].vel_y;
359
360         quadrant->width = 1;
361         quadrant->height = 1;
362         quadrant->origin_x = 0;
363         quadrant->origin_y = 0;
364         err += insert(root_new, &particles[i], quadrant);
365       }
366       if(err!=0)
367       {
368         free(particles_new);
369         free(root_new);
370         free(quadrant);
371         return -1;
372       }
373     }
374     *root = *root_new;
375     free(particles_new);
376     free(root_new);
377     free(quadrant);
378     return 0;
379   }
380
381   void calculateForce(quadtree_t** __restrict node, particle_t* __restrict particle, double* ...
          __restrict F, const double theta_max)
382   {
383     double r, theta;
384     const double epsilon = 0.001;
385     if ((*node) == NULL)
386     {
387       return;
388     }
389     /* External node */
390     else if ((*node)->particle != NULL)
391     {
392       /* Same particle */
393       if ((fabs((*node)->particle->pos_x - particle->pos_x) < 0.0000001) && ...
              (fabs((*node)->particle->pos_y - particle->pos_y) < 0.0000001))
394       {
395         return;
396       }
```

```
397       r = sqrt((particle->pos_x - (*node)->particle->pos_x)*(particle->pos_x - ...
              (*node)->particle->pos_x) + (particle->pos_y - ...
              (*node)->particle->pos_y)*(particle->pos_y - (*node)->particle->pos_y));
398       F[0] += (*node)->particle->mass*(particle->pos_x - ...
              (*node)->particle->pos_x)/((r+epsilon)*(r+epsilon)*(r+epsilon));
399       F[1] += (*node)->particle->mass*(particle->pos_y - ...
              (*node)->particle->pos_y)/((r+epsilon)*(r+epsilon)*(r+epsilon));
400     }
401     /* Internal node */
402     else
403     {
404       /* Calculate ratio */
405       theta = (*node)->quadrant->width/sqrt((particle->pos_x - ((*node)->quadrant->origin_x ...
              + (*node)->quadrant->width/2))*(particle->pos_x - ((*node)->quadrant->origin_x + ...
              (*node)->quadrant->width/2)) + (particle->pos_y - ((*node)->quadrant->origin_y + ...
              (*node)->quadrant->height/2))*(particle->pos_y - ((*node)->quadrant->origin_y + ...
              (*node)->quadrant->height/2)));
406
407       /* Single body problem */
408       if (theta ≤ theta_max)
409       {
410         /* x_i: particle, x_j: node->cm */
411         r = sqrt((particle->pos_x - (*node)->cm_x)*(particle->pos_x - (*node)->cm_x) + ...
              (particle->pos_y - (*node)->cm_y)*(particle->pos_y - (*node)->cm_y));
412         F[0] += (*node)->mass_tot*(particle->pos_x - ...
              (*node)->cm_x)/((r+epsilon)*(r+epsilon)*(r+epsilon));
413         F[1] += (*node)->mass_tot*(particle->pos_y - ...
              (*node)->cm_y)/((r+epsilon)*(r+epsilon)*(r+epsilon));
414       }
415       /* Many body problem */
416       else
417       {
418         calculateForce(&(*node)->top_left, particle, F, theta_max);
419         calculateForce(&(*node)->top_right, particle, F, theta_max);
420         calculateForce(&(*node)->bottom_left, particle, F, theta_max);
421         calculateForce(&(*node)->bottom_right, particle, F, theta_max);
422       }
423     }
424   }
425
426   void* thread_func(void* arg){
427     dataForThread_t* info = (dataForThread_t *) arg;
428
429     double* F = (double*)malloc(2*sizeof(double));
430     double ax, ay;
431     const double G = 100/(double)(info->N);
432
433     /* Compute new position of particle */
434     for(int i = info->start; i<info->stop; i++){ // for specific particles
435       F[0] = 0;
436       F[1] = 0;
437
438       calculateForce(info->root, &particles[i], F, info->theta_max);
439
440       F[0] = -G*particles[i].mass*F[0];
441       F[1] = -G*particles[i].mass*F[1];
442
443       /* Acceleration */
444       ax = F[0]/(particles[i].mass);
445       ay = F[1]/(particles[i].mass);
446       /* Updating velocities */
447       particles_new[i].vel_x = particles[i].vel_x + info->dt * ax;
448       particles_new[i].vel_y = particles[i].vel_y + info->dt * ay;
449       /* Updating positions */
450       particles_new[i].pos_x = particles[i].pos_x + info->dt * particles_new[i].vel_x;
```

```
451       particles_new[i].pos_y = particles[i].pos_y + info->dt * particles_new[i].vel_y;
452     }
453     free(F);
454     return NULL;
455   }
```