Assignment 4 & 5

Code Optimization in C: The Gravitational N-Body Problem Using the The Barnes-Hut Algorithm

Finn Joel Bjervig*, August Forsman†, Maja Linderholm‡, and Erik Turesson§

1TD062 - High Performance Programming,
Department of Information Technology,
Uppsala university, Sweden

August 31, 2022

*Electronic address: joelfbjervig@gmail.com
†Electronic address: aufo8456@student.uu.se
‡Electronic address: maja.linderholm@gmail.com
§Electronic address: ertu2293@student.uu.se

The Problem

Constructing simulations is at first glance a task of implementing equations and numerical methods to approximate solutions, up until the programs performance heavily rely on the scalability of its algorithms. Different methods scale differently, and even arithmetic operations in the computer can be chosen with precaution to make the program run faster. This assignment aims to simulate a closed, dynamical system of particles of varying mass and population (from a couple of particles up to thousands), subject to the exerted gravitational forces by each other. The gravitational force exerted on particle i by a particle j is

$$\mathbf{f}_{ij} = -G\frac{m_i m_j}{r_{ij}^2}\hat{\mathbf{r}}_{ij}$$

where $m_i,\ m_j$ are the masses of the particles, G is a gravitational constant, and $\mathbf{r}_{ij}$ is the vector from i to j. In this project, we consider a system of N particles with $G = 100/N$ and a slight modification in that we assume a minimal possible distance $\epsilon_0 = 10^{-3}$ between particles. Thus, each particle i is affected by the total force

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

The only analytical solution of such a system is limited to only two particles, well known as the two body problem (except for very specific cases of the equally famous three body problem). As soon as one introduces another body of significant mass, the system becomes chaotic. Computers have the advantage of numerical approximations but at a scale of thousands of particles the necessity of writing efficient code cannot be overstated. As the reader will see, there are many ways to bring down the running time of such a program, some of which can be used in general when coding, and other tricks which is acknowledged in this specific task.

The intuitive way of implementing the description above is to simply iterate through all particles, and calculate all forces asserted by all other particles. This *straightforward algorithm* (for lack of a better name), does yield a correct and accurate result, but is quite slow. Simulations and computational science boils down to a balance between accuracy and efficiency, and by acknowledging that compromise, one may look to algorithms that are far superior in speed but may not be as accurate as its competitor. For N-body simulations there exists an approximate solution called *The Barnes-Hut Algorithm*. The idea is to, in a quadrant, approximate a cluster of particles with a shared center of mass, and the total mass of all particles. Thereby it can compare one particle with this collective virtual particle and doesn't need to calculate all forces between all particles. Depending on the size of the cluster and the distance between the cluster and the particle, the approximation is more or less valid. To determine whether to proceed with the calculation or split up the cluster further, the *Theta Criterion* is evaluated. A constant θ_{max} is set by the user which is defined as

$$\theta = \frac{\text{side length of cluster square frame}}{\text{distance from particle to center of cluster frame}}$$

If $\theta < \theta_{max}$ there's sufficient precision and the current cluster can be used, otherwise the cluster is recursively split into smaller quadrant clusters until the condition is fulfilled.

The Solution

To calculate the evolution of particles N with a "straightforward"-algorithm and Barnes Hut algorithm in a gravitational simulation the code, which is listed in the Appendix, is structured in the following way.

Straightforward Algorithm

Data structures

Each particle is defined as a `struct` containing the mass, brightness, and vectors for position *pos*, velocity *vel*, and acceleration *acc*. Using only arrays to store the values of each particle did increase the performance but not significantly. Therefore, the less cluttered `struct` implementation was chosen.

Functions

`main`: Driver function of the simulation. Handles input arguments and graphics.
`read_doubles_from_file`: Reads initial data from the input file for each particle and stores it in a struct. An input file contains the initial position, velocity, mass and brightness for each particle.
`time_sim_vec`: Simulates the movement of the particles over time. The function will loop over each particle and calculates the force it is subjected to by other particles.
`write_to_file`: Writes results of final time step to output file.

Barnes Hut Algorithm

Data structures

The struct is implemented as in the straightforward algorithm. The N-particle structs are inserted into a quad-tree, which is a tree structure where each node has four children. It is essentially an adaption for a binary tree to store 2-dimensional data. In the root node, the domain of computation is subdivided into four fields (using notations NW, NE, SW, SE for each intercardinal direction) that can be accessed through a corresponding pointer. A node can be visualized as a rectangular domain and has the ability to split into four children according to the Barnes-Hut algorithm. When a layer is added, the node struct parameters `length` and `pos` are initialized from the values of its parent. Once a

particle is inserted into a node, the `total_mass`, and `center_mass` gets updated correspondingly and enables the program to compute forces from multiple node layers using a cluster of particles in the 2D-plane.

Functions

`main`: Driver function of the simulation. Handles input arguments and graphics.
`read_file`: Reads initial data from the input file for each particle and stores it in a struct. An input file contains the initial position, velocity, mass and brightness for each particle.
`write_to_file`: Writes results of final time step to output file.
`barnes_hut`: Simulates the movement of the particles over time. The function will loop over each particle and calculates the force it is subjected to by other particles. Calls functions `init`, `insert`, `force_traverse`, `del_tree` and `tree_draw`.
`init`: Initializes the root node of the quadtree.
`insert`: Inserts particles in the quadtree recursively.
`force_traverse`: Calculates the acceleration of each particle.
`del_tree`: Deletes the quadtree.
`tree_draw`: Draws the tree.

Optimizations

Variable Keywords

By using the `register` keyword the compiler tries to keep the variable in one of the CPU-registers instead of the memory. This minimizes memory accesses and optimizes the code on an instruction level. It was used on the most commonly used variables, such as the loop variables. Similarly, `const` was applied inside the time stepping function to applicable variables such as `N`, `delta_t`, and `G`. Together, these keywords reduced the execution time by almost 30% when no compiler optimization flags were being used. However, this did not seem to have a noticeable effect for more aggressive flags.

Vectorization

The vector datatype `vector_t` is defined as two allocated memory slots of size 2×8 bytes (corresponding to the size of two `double`) such that they're positioned next to each other. This memory alignment enables for arithmetic operations between such vectors, or vectors and scalars (piece-wise operations), to be computed during the same clock cycle. This method of *vectorizing* the code improves the running time significantly.

Calculating Distances

By storing θ_{max}^2 instead of just θ_{max} in our Barnes-Hut implementation, we can modify the condition to $\theta_{max}^2 \geq side^2/distance^2$. This means that we avoid

having to calculate the root of the distance which is a very costly operation. For larger N's, this alone reduced the execution time of our Barnes-Hut implementations by almost 25%, as can be seen in Figure 1.

Initalization of Child Nodes

When particles are inserted in the quadtree nodes are often split. When a split occurs, the former leaf node becomes a parent node and two different ways of initializing the new child nodes was implemented. Either all child nodes (NW, NE, SW, SE) was initialized even though only e.g. one of them was used to store a particle, or only the child node used to store the particle was initialized. Figure 1 shows the execution time for the two implementations and concludes that initializing only relevant child nodes yields faster execution.

Optimal θ_{max} for Barnes-Hut

In order to find a suitable value of θ_{max} we wanted to maximize it for higher performance while maintaining a target accuracy of 10^{-3}. The accuracy is measured as the maximal positional difference for any particle between its solutions in the exact and the Barnes-Hut algorithm after a certain amount of time steps. In table 1 we list measurements of this accuracy for some values of θ_{max}.

θ_{max}	Accuracy
1	0.0163
0.75	0.0122
0.5	0.0056
0.4	0.0047
0.3	0.0023
0.25	0.0009
0.0	0.0000

Table 1: The maximal positional difference between the exact and the Barnes-Hut solution after 200 time steps depending on the value of θ_{max}. These results were produced from a system of 2000 particles with $\Delta t = 10^{-3}$ s

As one can see, $\theta_{max} = 0.25$ just about reaches the target accuracy.

Performance and Discussion

The simulations ran on a *11th Gen Intel(R) Core(TM) i7-1165G7 @ 2.80GHz* CPU, compiled with GCC 10.2.0 on Ubuntu 20.04 through wsl (Windows Subsystem for Linux). We used `-Ofast`, `-march=native`, `-ffinite-math-only`, and `-fno-signed-zeros` as compiler flags to optimize for performance.

In figure 1 below, the execution times for our implementation times can be seen. For the Barnes-Hut algorithms, $\theta_{max} = 0.5$ was used. As can be seen,

the Barnes-Hut algorithms scale as $\mathcal{O}(n \log n)$ when increasing the number of particles and run much faster compared to the $\mathcal{O}(n^2)$ algorithm.

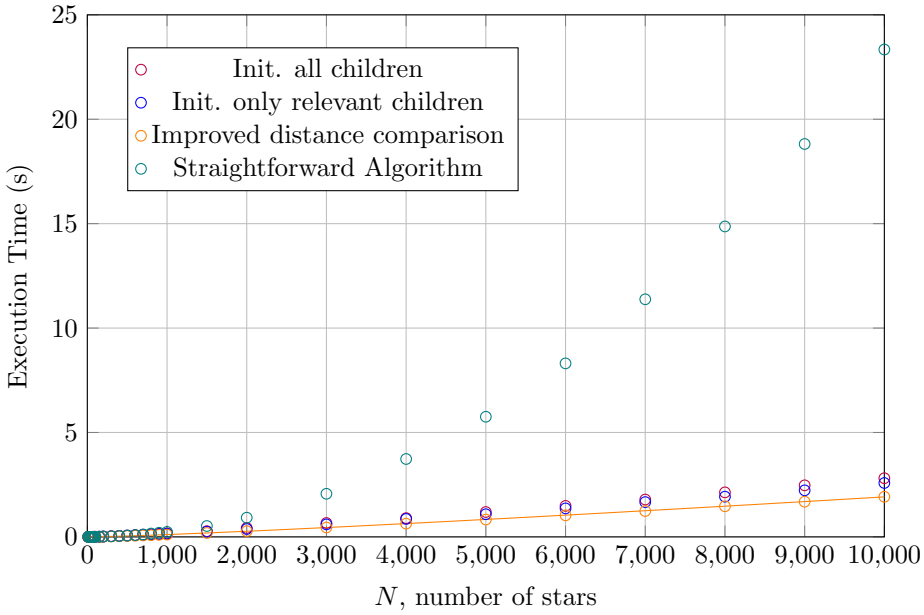


Figure 1: Serial execution times for 100 time steps using our optimizations for the $\mathcal{O}(n \log n)$ Barnes-Hut algorithm compared to our implementation of an $\mathcal{O}(n^2)$ straightforward algorithm. $\theta_{max} = 0.5$ was used for the Barnes-Hut executions. The solid line is a fitted $N \log N$ approximation of the best Barnes-Hut implementation.

Parallelization using Pthreads

Splitting up the work to several threads enables for tasks to be computed in parallel. The threads are then distributed between the CPU's cores. The number of cores and the number of threads per core depends on the CPU model, but many more recent personal computers will usually have 4-16 cores and 2 threads per core. As previously mentioned, our test were ran on a *11th Gen Intel(R) Core(TM) i7-1165G7 @ 2.80GHz* CPU, which has 4 cores with up to 2 threads per core.

If implemented perfectly, the speedup of an embarrassingly parallel problem should be directly proportional to the number of threads used. Two threads would then correspond to a double speedup, and three threads to a three-fold speedup, etc.

Parallelization of the Straightforward Algorithm

When examining our straightforward implementation, we found that we spent around 98% of the time in calculating the forces for problems of large Ns. Calculating the forces for particles can be done individually for each particle and should thus be suitable to parallelization. The amount of work when iterating through the particle array is always the same: $N - 1$ computations for each of the N particles (the scaling of this method is $\mathcal{O}(N(N-1))$). The optimal splitting of tasks is therefore an even split such that each thread manages N/P particles, where P is the number of threads used.

Figure 2 displays the *straightforward* algorithm's performance depending on the number of threads. Interestingly, beyond around four threads, there is not much of a difference performance-wise, no matter how many threads is used. This may be explained by the fact that the threads don't work independently. The particles handled by one thread still needs the forces exerted by particles in the other threads, which causes delay. Additionally, it might be due to the fact that while the CPU used can handle 2 threads per core, it still just has 4 cores. Going above this limit might be the reason for the performance decrease in scaling.

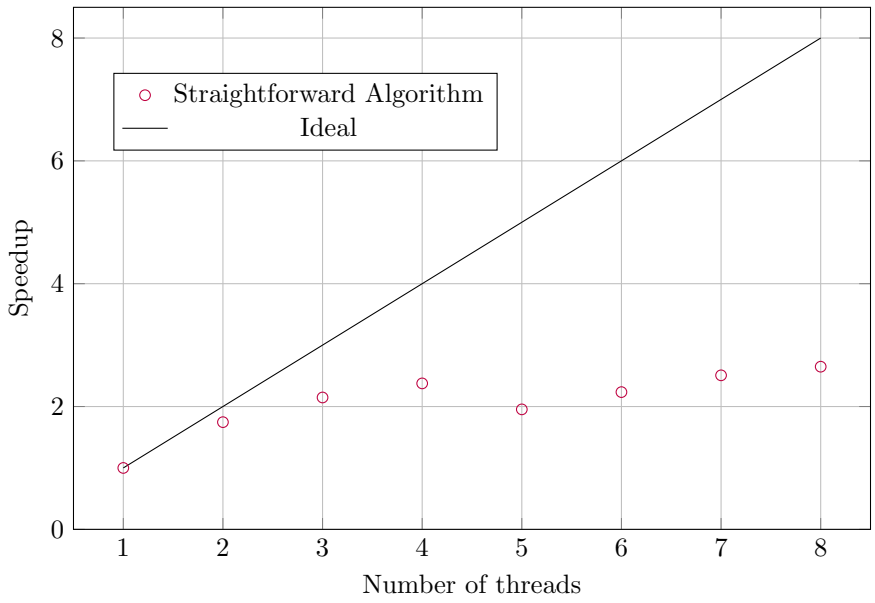
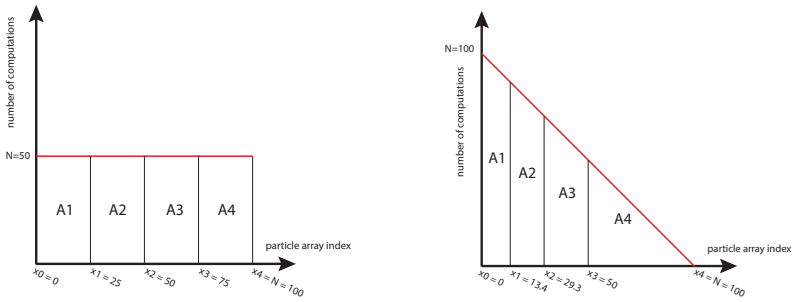


Figure 2: The relative speedup of our Straightforward implementation versus the number of threads used

Originally, we tried another approach where we used Newton's third law such that when particle i has its subjected forces computed by particle j, the latter will also be assigned the equal but opposite force. The scaling is therefore half

of the previous one, $\mathcal{O}(\frac{1}{2}N(N-1))$. While the serial implementation of this approach had about half the execution time, it showcased very bad scaling due to uneven workload between threads, and a lot of overhead. Thus we chose to not go further with implementing that solution in this assignment. However, below one can find a discussion of how workload could theoretically be more evenly divided. This splitting was not implemented but can be investigated for further work.

If we use Newton's third law, even if we would have the same number of particles in an evenly split the particle array, the number of computations in those intervals will not be equally many. In the beginning of the array there are $N-1$ forces to compute, then $N-2$ until the last particle where there are no forces left to compute. Therefore, the amount of work decreases linearly from N to 0 as: *number of computations* $= N -$ *particle array index*. In figure 3b the four intervals are set such that the total work for each thread (the area under the curve, A_i), is the same for all threads. See Figure 3a, for an example of splitting between four threads.



(a) An even work load distribution

(b) Linearly decreasing work load distribution

Figure 3: On the premise of equal work among the threads, $A_1 = A_2 = A_3 = A_4$ the number of particles handled by each thread is different for the two scenarios

The bounds are derived from the fact that $A_1 = A_2 = A_3 = A_4$ and that $P \cdot A = \frac{1}{2}N^2 \quad \forall A_i, \quad i \in [1,4]$. This yields the following equation

$$x_{i+1} = -\sqrt{(x_i - N)^2 - \frac{N^2}{P}} + N \tag{1}$$

where N is the number of particles in the system, P are the number of threads used and the initial point is $x_0 = 0$.

Parallelization of the Barnes-Hut Algorithm

For the parallelization of our Barnes-Hut Algorithm we started at looking what parts of the code took the longest time for large values of N. For our serial implementation of Barnes-Hut, we found that for a simulation of 100 time steps of 20000 particles, using $\theta_{max} = 0.25$, we spent about 96.5% of the total execution time calculating and updating the forces for each particle, 3.4% for building and deleting the quadtree at each time step, and the rest for file I/O. Thus, we chose to spend our effort in trying to parallelize only the force calculations.

Our multithreaded implementation utilizes one master thread that handles the construction of the quadtree at each time step. Once the tree is built, the master thread reaches a barrier. Worker threads wait at this barrier, and then starts updating the particles. Noting that the force calculation should be independent for each particle, we evenly divided the particles between the worker threads. Once all threads have updated all its particles, the main thread is unlocked, and deletes the tree. This process is then repeated until we have reached the desired number of time steps.

As seen in figure 4, the speedup is close to the ideal (1.85 for 2 threads) in the beginning, but stagnates with increasing number of threads. Important to note is however that the speedup varies significantly between different θ_{max}, which implies the importance of the parameter beyond the scaling of the number of particles.

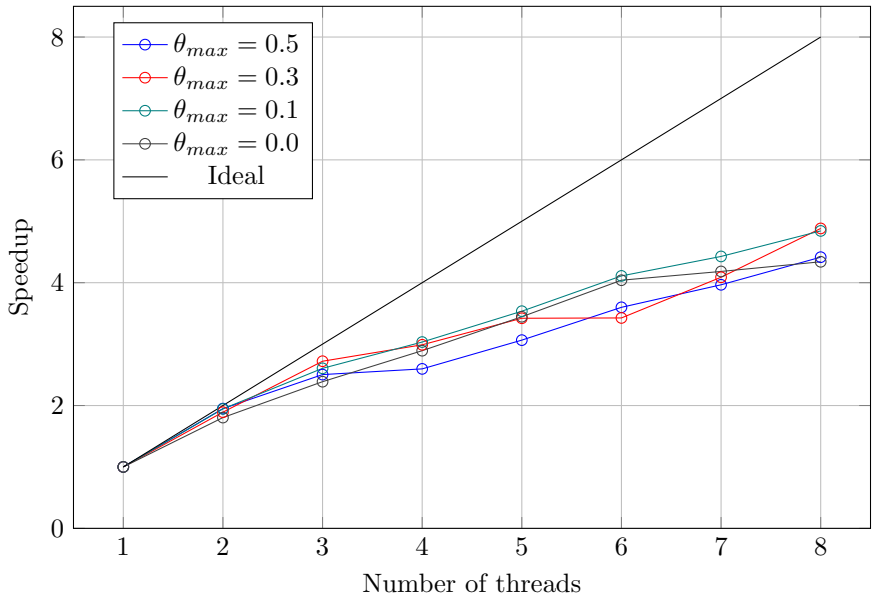


Figure 4: The relative speedup of our Barnes-Hut implementation versus the number of threads used for some different values of θ_{max}

When using tree structures as in the Barnes-Hut algorithm it is difficult to know how to split up the load between the threads, since some particles will require more recursive descent than others in order to fulfill the *Theta criterion*. We suspected the stagnant scaling to be due to uneven load-balancing between the threads. Therefore, we tried another queue-based approach where threads took a particle index from a global counter, starting at $N - 1$, before decrementing it until it reached 0. While this approach seemed to display a somewhat better scaling, the baseline performance was a lot worse, and thus we chose to not explore that possibility further.

It is evident in figure 4 that the program benefited from parallelization. The decreased slope of the data points can be derived from Amdahl's law

$$\frac{1}{1-p} = 4.88 \tag{2}$$

where 4.88 is the highest measured speedup for 8 threads. Solving $p \approx 0.80$ means that the implemented techniques was able to parallelize 80% of the program.

Appendix

Serial Straightforward galsim.c

```
#include <math.h>
#include <stdio.h>
#include <stdlib.h>
#include "graphics.h"

typedef double vector_t __attribute__((vector_size(2 * sizeof(
    double))));

#define EPSILON_0 1e-3

typedef struct particle
{
/*
    Particle properties:
    vel - velocity
    pos - coordinates
*/
    vector_t pos;
    double mass;
    vector_t vel;
    double brightness;
    vector_t acc;
} p_t;

void read_doubles_from_file(int N, p_t *p, const char *fileName);
void print_struct(p_t p);
//void time_sim(int nsteps, const double delta_t, p_t *p, int N,
    const int graphics);
void write_to_file(p_t *p, int N, char *fileName);
//void time_sim2(int nsteps, const double delta_t, p_t *input_array
    , int N, const int graphics);
void time_sim_vec(const int nsteps, const double delta_t, p_t *p,
    const int N,
                  const int graphics);

int main(int argc, char *argv[])
{
    // Check correct number of input arguments
    if (argc != 6)
    {
        printf("Wrong number of inputs \n");
        exit(EXIT_FAILURE);
    }

    // Define input variables
    const int N = atoi(argv[1]);
    const int nsteps = atoi(argv[3]);
    const double delta_t = atof(argv[4]);
    const int graphics = atoi(argv[5]);

    p_t p[N];

    if (graphics)
```

```
50      {
51          InitializeGraphics(argv[0], 800, 800);
52          SetCAxes(0, 1);
53          read_doubles_from_file(N, p, argv[2]);
54
55          // print structs if wanted
56          //for (int i=0; i<N; i++) print_struct(p[i]);
57          time_sim_vec(nsteps, delta_t, p, N, graphics);
58          FlushDisplay();
59          CloseDisplay();
60          write_to_file(p, N, "result.gal");
61      }
62      else
63      {
64          read_doubles_from_file(N, p, argv[2]);
65
66          // print structs if wanted
67          //for (int i=0; i<N; i++) print_struct(p[i]);
68          time_sim_vec(nsteps, delta_t, p, N, graphics);
69          write_to_file(p, N, "result.gal");
70      }
71      return 0;
72  }
73
74  void write_to_file(p_t *p, int N, char *fileName)
75  {
76    FILE *file = fopen(fileName, "wb");
77
78    for (int i=0; i<N; i++){
79      fwrite(&p[i].pos,        sizeof(p[i].pos),        1, file);
80      fwrite(&p[i].mass,       sizeof(p[i].mass),       1, file);
81      fwrite(&p[i].vel,        sizeof(p[i].vel),        1, file);
82      fwrite(&p[i].brightness, sizeof(p[i].brightness), 1, file);
83    }
84    fclose(file);
85  }
86
87  void print_struct(p_t p)
88  {
89      printf(
90          "x: %lf\t y: %lf\t mass: %lf\t vel_x: %lf\t vel_y: %lf\t
      brightness: "
91          "%lf\n",
92          p.pos[0], p.pos[1], p.mass, p.vel[0], p.vel[1], p.
      brightness);
93  }
94
95
96  ////////////////////
97  // READ DOUBLES //
98  ////////////////////
99  // NOTE: might not work because vector_t might not operate the same
        way as a regular variable.
100
101 void read_doubles_from_file(int N, p_t *p, const char *fileName)
102 {
103     /* Open input file and determine its size. */
```

```
104     FILE *file = fopen(fileName, "rb");
105     if (!file)
106     {
107         printf("read_doubles_from_file error: failed to open input
    file '%s'.\n",
108                fileName);
109         exit(EXIT_FAILURE);
110     }
111     // zerovector for vectorization convenience
112     vector_t zerovec;
113     zerovec[0] = 0;
114     zerovec[1] = 0;
115 
116     p_t temp;
117     double buffer[6];   // file reading buffer
118     for(int i = 0; i<N; i++) {
119         // Read doubles into particle struct
120         fread(&buffer, sizeof(double), 6, file);
121         temp.pos[0] = buffer[0];
122         temp.pos[1] = buffer[1];
123         temp.mass   = buffer[2];
124         temp.vel[0] = buffer[3];
125         temp.vel[1] = buffer[4];
126         temp.brightness = buffer[5];
127         temp.acc    = zerovec;
128         p[i] = temp;
129     }
130     // Close file and check if closed successfully
131     if (fclose(file) != 0)
132     {
133         printf("read_doubles_from_file error: error closing input
    file.\n");
134         exit(EXIT_FAILURE);
135     }
136 }
137 
138 // VECTORIZED
139 void time_sim_vec(const int nsteps, const double delta_t, p_t *p,
    const int N,
140                   const int graphics)
141 {
142   vector_t diff, diffsqrd, F;
143 
144   // zerovector for vectorization convenience
145   vector_t zerovec;
146   zerovec[0] = 0;
147   zerovec[1] = 0;
148 
149   double distance;
150   const double G = 100.0 / N;
151   float circleradius[N], circlecolor = 0;
152 
153   if (graphics)
154   {
155     // make radius dependent on the mass of the particle
156     for (register int i = 0; i < N; ++i)
157     {
```

```
158       circleradius[i] = p[i].mass * 1.5e-3;
159     }
160   }
161     // loop over all timesteps
162   for (int t = 0; t < nsteps; t++)
163   {
164
165     // reset all p's accelerations to zero for each time step
166     for (register int i = 0; i < N; i++)
167     {
168         p[i].acc = zerovec;
169     }
170
171     // iterate through all particles
172     for (register int i = N-1; i >= 0; i--)
173     {
174       for (register int j = i - 1; j >= 0; j--)   // only up til
    the particle in the above foor loop
175       {
176         // temp variables
177         diff = p[i].pos - p[j].pos; // particles relative position
    in x,y coordinates
178         diffsqrd = diff*diff;       // - squared
179         distance = sqrt(diffsqrd[0]+diffsqrd[1])+ EPSILON_0;     //
    pythagoras thrm + small number for
180                                                                  //
    smoothing when distance is small
181
182         // NOTE: maybe faster to just have everything on one long
    caluclations,
183         // because now we store them in temporary variables so its
    easier to see whats happening
184         // something we could do in the end?
185         F = G*diff/(distance*distance*distance);  // 1/r^2 in the
    direction of r
186         p[i].acc +=p[j].mass*F;
187         p[j].acc -=p[i].mass*F;
188       }
189     }
190     // when all the particles accelerations have been calculated,
191     // we can now, for the same time t_i update all particles
    velocities
192     // and positions, so that no particle is in a different
    timestep
193     for (register int i = 0; i < N; i++)
194     {
195       p[i].vel -=delta_t* p[i].acc;
196       p[i].pos += delta_t*p[i].vel;
197     }
198
199     if (graphics)
200     {
201       ClearScreen();
202       // picasso away!
203       for (int i = 0; i < N; i++) DrawCircle(p[i].pos[0], p[i].pos
    [1], 1, 1, circleradius[i], circlecolor);
204       Refresh();
```

```
      usleep(30);
    }
  }
}
```

Serial Barnes-Hut galsim.c

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include "graphics.h"

#define EPSILON_0 1e-3    //Plummer radius
#define W 1              // Window width
#define L 1               // Window length


typedef double vector_t __attribute__((vector_size(2 * sizeof(
    double))));

/*
    Particle:
    * pos - coordinates
    * mass - mass of particle
    * vel - velocity
    * brightness - brightness of particle
    * acc - acceleration
*/
typedef struct particle
{
    vector_t pos;
    double mass;
    vector_t vel;
    double brightness;
    vector_t acc;
} p_t;

/*
    Quad tree node:
    * particle - particle struct pointer
    * length - length of box/quadnode
    * total_mass - mass of all particles inside a box
    * center_mass - position of center of mass
    * pos          - coordinate of center of box
    * NE,NW,SW,SE   - quadrants of domain. NE = norteast, etc.. (
    these are the nodes children)
*/
typedef struct quad_tree_node
{
    p_t *particle;
    double length;
    double total_mass;
    vector_t center_mass;
    vector_t pos;
    struct quad_tree_node *NE;
    struct quad_tree_node *NW;
```

```
48      struct quad_tree_node *SW;
49      struct quad_tree_node *SE;
50
51  } q_node_t;
52
53  /*
54      Initialize global helpfull vectors
55  */
56  const vector_t zerovec = {0.0, 0.0};
57  const vector_t se_vec = {1.0, 1.0};
58  const vector_t sw_vec = {-1.0, 1.0};
59  const vector_t ne_vec = {1.0, -1.0};
60  const vector_t nw_vec = {-1.0, -1.0};
61
62
63
64  /*
65  Macro function that creates a child to the parent, with a vector
66  in the corresponding direction.
67      Example: Create a child to the north-west to a q_node_t ** node
68      create_child((*node)->NW, (*node), nw_vec);
69  */
70  #define create_child(child, parent, dir) \
71  ({ \
72    child = (q_node_t *) malloc(sizeof(q_node_t)); \
73    child->pos = parent->pos + 0.25 * (parent->length) * dir; \
74    child->length = parent->length * 0.5; \
75    child->total_mass = 0.0; \
76    child->particle = NULL; \
77    child->NW = NULL; \
78    child->NE = NULL; \
79    child->SW = NULL; \
80    child->SE = NULL; \
81    })
82
83
84  /*
85  Macro function that locates the correct child quadrant to insert a
        particle
86  into. If val==0: check if child already exists. If not, create it
        before inserting.
87  If val==1: insert knowing that child does not exist.
88  */
89  #define split_insert(val,particle) \
90  ({ \
91      if ((particle->pos[0] >= (*node)->pos[0]) && (particle->pos[1]
        < (*node)->pos[1])) { \
92          if (val||(*node)->NE == NULL) { \
93              create_child((*node)->NE, (*node), ne_vec); \
94          } \
95          insert(&((*node)->NE), particle); \
96      } \
97      else if ((particle->pos[0] < (*node)->pos[0]) && (particle->pos
        [1] < (*node)->pos[1])){ \
98          if (val||(*node)->NW == NULL) { \
99              create_child((*node)->NW, (*node), nw_vec); \
100         } \
```

```
101         insert(&((*node)->NW), particle); \
102     } \
103     else if ((particle->pos[0] < (*node)->pos[0]) && (particle->pos
        [1] >= (*node)->pos[1])) { \
104         if (val||(*node)->SW == NULL) { \
105             create_child((*node)->SW, (*node), sw_vec); \
106         } \
107         insert(&((*node)->SW), particle); \
108     } \
109     else { \
110         if (val||(*node)->SE == NULL) { \
111             create_child((*node)->SE, (*node), se_vec); \
112         } \
113         insert(&((*node)->SE), particle); \
114     } \
115     })
116
117
118
119 /*
120     Declare functions
121 */
122
123 //IO functions
124 void read_file(int N, p_t *p, const char *fileName);
125 void write_to_file(p_t *p, int N, char *fileName);
126 void print_struct(p_t p); //mainly for debugging
127
128 q_node_t *init(); //create tree root
129 void insert(q_node_t **node, p_t *particle); //recursive insertion
        of particle into tree
130 void tree_del(q_node_t **node); //free memory recursivly
131 void barnes_hut(const int nsteps, const double delta_t, p_t *p,
        const int N,
132                 const double theta_max, const int graphics); //the
        time stepping algorithm
133 void force_traverse(q_node_t **node, const double theta_max, p_t *
        particle); //calculation of the forces
134
135 void tree_draw(q_node_t **node); //recursive drawing of the
        quadnodes
136
137
138
139 /*
140     Driver function
141 */
142 int main(int argc, char *argv[])
143 {
144     // Check correct number of input arguments
145     if (argc != 7)
146     {
147         printf("Wrong number of inputs \n");
148         exit(EXIT_FAILURE);
149     }
150
151     // Define input variables
```

```
152     const int N = atoi(argv[1]);
153     const int nsteps = atoi(argv[3]);
154     const double delta_t = atof(argv[4]);
155     const double theta_max = atof(argv[5]) * atof(argv[5]); //
        square for cheaper comparison in the force calculation
156     const int graphics = atoi(argv[6]);
157 
158     //Read from input file and save in an array of particles
159     p_t p[N];
160     read_file(N, p, argv[2]);
161 
162     // initialize window if graphics is enabled
163     if (graphics) InitializeGraphics(argv[0], 800, 800);
164 
165     //Simulate particle system with Barnes Hut algorithm and save
        result
166     barnes_hut(nsteps, delta_t, p, N, theta_max, graphics);
167     write_to_file(p, N, "result.gal");
168     return 0;
169 }
170 
171 
172 
173 /*
174     Function that reads doubles from input file and stores them in
        an array of structs
175 */
176 void read_file(int N, p_t *p, const char *fileName)
177 {
178     /* Open input file and determine its size. */
179     FILE *file = fopen(fileName, "rb");
180     if (!file)
181     {
182         printf("read_doubles_from_file error: failed to open input
        file '%s'.\n",
183                fileName);
184         exit(EXIT_FAILURE);
185     }
186 
187     p_t temp;
188     double buffer[6]; // file reading buffer
189     for (int i = 0; i < N; i++)
190     {
191         // Read doubles into particle struct
192         if (0==fread(&buffer, sizeof(double), 6, file)) exit(
        EXIT_FAILURE);
193         temp.pos[0] = buffer[0];
194         temp.pos[1] = buffer[1];
195         temp.mass = buffer[2];
196         temp.vel[0] = buffer[3];
197         temp.vel[1] = buffer[4];
198         temp.brightness = buffer[5];
199         temp.acc = zerovec;
200         p[i] = temp;
201     }
202     // Close file and check if closed successfully
203     if (fclose(file) != 0)
```

```
204     {
205         printf("read_doubles_from_file error: error closing input
    file.\n");
206         exit(EXIT_FAILURE);
207     }
208 }
209
210 /*
211     Function that writes output to file
212 */
213 void write_to_file(p_t *p, int N, char *fileName)
214 {
215     FILE *file = fopen(fileName, "wb");
216
217     for (int i = 0; i < N; i++)
218     {
219         fwrite(&p[i].pos, sizeof(p[i].pos), 1, file);
220         fwrite(&p[i].mass, sizeof(p[i].mass), 1, file);
221         fwrite(&p[i].vel, sizeof(p[i].vel), 1, file);
222         fwrite(&p[i].brightness, sizeof(p[i].brightness), 1, file);
223     }
224     fclose(file);
225 }
226
227 /*
228     Function that prints values of a struct. Used for debugging
229 */
230 void print_struct(p_t p)
231 {
232     printf(
233         "x: %lf\t y: %lf\t mass: %lf\t vel_x: %lf\t vel_y: %lf\t
    brightness: "
234         "%lf\n",
235         p.pos[0], p.pos[1], p.mass, p.vel[0], p.vel[1], p.
    brightness);
236 }
237
238 /*
239     Function to initialize the root of the the quad tree
240 */
241 q_node_t *init()
242 {
243     q_node_t *root = (q_node_t *)malloc(sizeof(q_node_t));
244
245     //define the initial data
246     root->particle = NULL;
247     root->length = 1.0;
248     root->total_mass = 0.0;
249     root->center_mass = zerovec;
250     root->pos= 0.5 * se_vec;
251
252     //create empty children to the node
253     root->NE = NULL;
254     root->NW = NULL;
255     root->SW = NULL;
256     root->SE = NULL;
257
```

```
258     return root;
259 }
260 
261 
262 /*
263 Recursive insertion of a particle into the quadtree.
264 Updates mass + center of mass of each node,  then creates the
        required child node and inserts
265 particle into it
266 */
267 void insert(q_node_t **node, p_t *particle)
268 {
269     //update mass and center of mass
270     (*node)->center_mass = ((*node)->total_mass * (*node)->
    center_mass + particle->mass * particle->pos) / ((*node)->
    total_mass + particle->mass);
271     (*node)->total_mass = (*node)->total_mass + particle->mass;
272 
273     //check if a particle is present at the node
274     if ((*node)->particle == NULL)
275     { // if there is no particle, then check if it has children
276 
277         if ((*node)->NW == NULL && (*node)->NE == NULL && (*node)->
    SE == NULL && (*node)->SW == NULL)
278         {                                   // if it doesnt have
    any children,
279             (*node)->particle = particle;   // this means there is
    no particle or sub-quadrants. Free to add the new particle!!
280             return;
281         }
282         else //insert into the correct child
283         {
284             split_insert(0,particle); //finds the correct child,
    creates it if not existent, and inserts it into that node
285         }
286     }
287     else
288     {
289         //the particle in the node needs to be inserted into a
    child instead
290         p_t *old_particle = (*node)->particle;
291         (*node)->particle = NULL;
292 
293         // first argument (1) makes if statement in macro function
    obsolete since if statement
294         // will allways be true
295 
296         //insert the particles and create children if needed
297         split_insert(1,particle);
298         split_insert(0,old_particle);
299     }
300 }
301 
302 /*
303     Delete tree recursivly, starting at the root
304 */
305 void tree_del(q_node_t **node)
```

```
306 {
307     //check for the existance of children, and delete those that
    are present
308     if ((*node) != NULL) {
309     if ((*node)->NW != NULL)
310         tree_del(&(**node).NW);
311 
312     if ((*node)->NE != NULL)
313         tree_del(&(**node).NE);
314 
315     if ((*node)->SE != NULL)
316         tree_del(&(**node).SE);
317 
318     if ((*node)->SW != NULL)
319         tree_del(&(**node).SW);
320     }
321     free(*node);
322     *node=NULL;
323 }
324 
325 /*
326     Traverse down the quad tree and calculate what forces effect a
    particle,
327     and update its acceleration.
328 */
329 void force_traverse(q_node_t **node, const double theta_max, p_t *
    particle)
330 {
331     //If quadrant is empty or particle is current particle
332     if ((*node) == NULL || (*node)->total_mass == 0 || particle ==
    (*node)->particle)
333     {
334         return;
335     }
336 
337     //Get the distance
338     vector_t diff = particle->pos - (*node)->pos;       // particles
     relative position in x,y coordinates
339     vector_t diffsqrd = diff * diff;                    // squared
340     double distance = diffsqrd[0] + diffsqrd[1]; //squared distance
341 
342 
343     //If theta condition is met calculate acceleration of particle
344     //else traverse down to child nodes
345 
346     //NOTE: the original condition is theta_max >= length/distance,
     but below the
347     //comparison is instead theta_max^2 * distance^2 >= lenght^2.
    Thus we don't need
348     //to take the sqrt for the distance above. ~20% time reduction
    for N=10000 from this
349 
350     if (theta_max * distance>= ( ((*node)->length) * ((*node)->
    length) )  || (*node)->particle != NULL)
351     {
352         diff = particle->pos - (*node)->center_mass;            //
    particles relative position in x,y coordinates
```

```
353         diffsqrd = diff * diff;                                    //
    squared
354         distance = sqrt(diffsqrd[0] + diffsqrd[1]) + EPSILON_0; //
    pythagoras
355         particle->acc -= (*node)->total_mass * diff / (distance *
    distance * distance);
356     }
357     else
358     {
359         //if theta condition not fulfilled, need to take another
360         //recursive step to each of the children
361         if ((*node)->NW != NULL) force_traverse(&((*node)->NW),
    theta_max, particle);
362         if ((*node)->NE != NULL) force_traverse(&((*node)->NE),
    theta_max, particle);
363         if ((*node)->SW != NULL) force_traverse(&((*node)->SW),
    theta_max, particle);
364         if ((*node)->SE != NULL) force_traverse(&((*node)->SE),
    theta_max, particle);
365     }
366 }
367
368 /*
369     Barnes Hut algorithm
370 */
371 void barnes_hut(const int nsteps, const double delta_t, p_t *p,
    const int N, const double theta_max, const int graphics)
372 {
373
374     // if we use graphics initialize window
375     float circleradius[N], circlecolor = 0;
376     if (graphics) {
377         SetCAxes(0, 1);
378
379         // make radius dependent on the mass of the particle
380         for (register int i = 0; i < N; ++i)
381         {
382         circleradius[i] = fmax(1.5e-3, p[i].mass * 1.5e-3); //have
    a smallest possible particle size
383         }
384
385     }
386
387     const double G_dt = (100.0 * delta_t) / N;
388
389     for (int t = 0; t < nsteps; t++)
390     {
391         q_node_t *root = init();
392
393         for (register int i = N - 1; i >= 0; i--)
394         {
395             insert(&root, &(p[i]));
396         }
397
398         for (register int i = N - 1; i >= 0; i--)
399         {
400             force_traverse(&root, theta_max, &(p[i]));
```

```
401         }
402 
403         // When all the particles accelerations have been
    calculated,
404         // we can now, for the same time t_i update all particles
    velocities
405         // and positions, so that no particle is in a different
    timestep
406         for (register int i = N - 1; i >= 0; i--)
407         {
408             p[i].vel += G_dt * p[i].acc;
409             p[i].pos += delta_t * p[i].vel;
410             p[i].acc = zerovec;
411         }
412 
413         if (graphics) {
414             ClearScreen();
415             // picasso away!
416             for (int i = 0; i < N; i++) {
417                 DrawCircle(p[i].pos[0], p[i].pos[1], W, L,
    circleradius[i], circlecolor);
418             }
419             tree_draw(&root);
420 
421             Refresh();
422             usleep(1000);
423         }
424         tree_del(&root);
425     }
426 
427     // if we use graphics, flush and close window
428     if (graphics) {
429         FlushDisplay();
430         CloseDisplay();
431     }
432 }
433 
434 /*
435         1)  recursivly traverse the tree, until a region with one
    particle p_i is reached
436 
437         2)  traverse into another quadrant
438             - calculate distace r from particle p_i to center of
    mass of all particles in the quadrant
439             - calculate sidelength h of quadrant (or max distance
    between any two points in the quadrant)
440             - Theta criterion h/r < theta. theta usually set to 0.5
441 
442         3)  CRITERION
443                 IF criterion satisfied:
444                     - calculate force acting on p_i by the
    collective shared force of particles in group by center of mass
445                 IF criterion NOT satisfied:
446                     - traverse recursively into its four quadrants
447                     - jump to calculations of 2)
448     */
449 
```

```
/*
    Draws quadrants recursivly, starting with the root
*/
void tree_draw(q_node_t **node){
    if(&(**node) != NULL){
        DrawRectangle((**node).pos[0]-(((**node).length)/2.0), (**
    node).pos[1]-(((**node).length)/2.0), W, L, (**node).length,
    (**node).length, 0.85);
        tree_draw(&(**node).NW);
        tree_draw(&(**node).SW);
        tree_draw(&(**node).NE);
        tree_draw(&(**node).SE);
    }
}
```

Multithreaded Straighforward galsim.c

```
#include <math.h>
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include "graphics.h"

pthread_barrier_t barrier;

typedef double vector_t __attribute__((vector_size(2 * sizeof(
    double))));

#define EPSILON_0 1e-3
int nthreads;
const vector_t zerovec = {0.0, 0.0};

typedef struct particle
{
  /*
    Particle properties:
    vel - velocity
    pos - coordinates
*/
  vector_t pos;
  double mass;
  vector_t vel;
  double brightness;
  vector_t acc;
} p_t;

/*
struct that contains all the relevant data for a thread
*/
typedef struct thread_data
{
  int lower_bound;
  int upper_bound;
  int thread_id;
  int N;
  int nsteps;
```

```
39   double G_dt;
40   double delta_t;
41   p_t *particle_array;
42
43 } thread_data_t;
44
45 void read_doubles_from_file(int N, p_t *p, const char *fileName);
46 void print_struct(p_t p);
47 //void time_sim(int nsteps, const double delta_t, p_t *p, int N,
       const int graphics);
48 void write_to_file(p_t *p, int N, char *fileName);
49 //void time_sim2(int nsteps, const double delta_t, p_t *input_array
       , int N, const int graphics);
50 void time_sim_vec(const int nsteps, const double delta_t, p_t *p,
       const int N,
51                   const int graphics);
52 void *thread_force_update(void *args);
53
54 int main(int argc, char *argv[])
55 {
56   // Check correct number of input arguments
57   if (argc != 7)
58   {
59     printf("Wrong number of inputs \n");
60     exit(EXIT_FAILURE);
61   }
62
63   // Define input variables
64   const int N = atoi(argv[1]);
65   const int nsteps = atoi(argv[3]);
66   const double delta_t = atof(argv[4]);
67   const int graphics = atoi(argv[5]);
68   nthreads = atoi(argv[6]);
69
70   p_t p[N];
71
72   if (graphics)
73   {
74     InitializeGraphics(argv[0], 800, 800);
75     SetCAxes(0, 1);
76     read_doubles_from_file(N, p, argv[2]);
77
78     // print structs if wanted
79     //for (int i=0; i<N; i++) print_struct(p[i]);
80     time_sim_vec(nsteps, delta_t, p, N, graphics);
81     FlushDisplay();
82     CloseDisplay();
83     write_to_file(p, N, "result.gal");
84   }
85   else
86   {
87     read_doubles_from_file(N, p, argv[2]);
88
89     // print structs if wanted
90     //for (int i=0; i<N; i++) print_struct(p[i]);
91     time_sim_vec(nsteps, delta_t, p, N, graphics);
92     write_to_file(p, N, "result.gal");
```

```
93    }
94    return 0;
95  }
96
97  void write_to_file(p_t *p, int N, char *fileName)
98  {
99    FILE *file = fopen(fileName, "wb");
100
101   for (int i = 0; i < N; i++)
102   {
103     fwrite(&p[i].pos, sizeof(p[i].pos), 1, file);
104     fwrite(&p[i].mass, sizeof(p[i].mass), 1, file);
105     fwrite(&p[i].vel, sizeof(p[i].vel), 1, file);
106     fwrite(&p[i].brightness, sizeof(p[i].brightness), 1, file);
107   }
108   fclose(file);
109 }
110
111 void print_struct(p_t p)
112 {
113   printf(
114       "x: %lf\t y: %lf\t mass: %lf\t vel_x: %lf\t vel_y: %lf\t
      brightness: "
115       "%lf\n",
116       p.pos[0], p.pos[1], p.mass, p.vel[0], p.vel[1], p.brightness)
      ;
117 }
118
119 ////////////////////
120 // READ DOUBLES //
121 ////////////////////
122 // NOTE: might not work because vector_t might not operate the same
       way as a regular variable.
123
124 void read_doubles_from_file(int N, p_t *p, const char *fileName)
125 {
126   /* Open input file and determine its size. */
127   FILE *file = fopen(fileName, "rb");
128   if (!file)
129   {
130     printf("read_doubles_from_file error: failed to open input file
       '%s'.\n",
131            fileName);
132     exit(EXIT_FAILURE);
133   }
134
135   p_t temp;
136   double buffer[6]; // file reading buffer
137   for (int i = 0; i < N; i++)
138   {
139     // Read doubles into particle struct
140     fread(&buffer, sizeof(double), 6, file);
141     temp.pos[0] = buffer[0];
142     temp.pos[1] = buffer[1];
143     temp.mass = buffer[2];
144     temp.vel[0] = buffer[3];
145     temp.vel[1] = buffer[4];
```

```
146     temp.brightness = buffer[5];
147     temp.acc = zerovec;
148     p[i] = temp;
149   }
150   // Close file and check if closed successfully
151   if (fclose(file) != 0)
152   {
153     printf("read_doubles_from_file error: error closing input file
      .\n");
154     exit(EXIT_FAILURE);
155   }
156 }
157
158 // VECTORIZED
159 void time_sim_vec(const int nsteps, const double delta_t, p_t *p,
      const int N,
160                   const int graphics)
161 {
162   //vector_t diff, diffsqrd, F;
163
164   //double distance;
165   const double G_dt = delta_t * 100.0 / N;
166
167   float circleradius[N], circlecolor = 0;
168
169   pthread_t threads[nthreads];
170   thread_data_t thread_data[nthreads];
171   pthread_barrier_init(&barrier, NULL, nthreads); //+1 since main
      is also waiting
172   pthread_attr_t attr;
173   pthread_attr_init(&attr);
174   pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_JOINABLE);
175
176   for (int i = 0; i < nthreads; i++)
177   {
178     thread_data[i] = (thread_data_t){.nsteps=nsteps, .lower_bound =
       i * N / nthreads, .upper_bound = (i + 1) * N / nthreads, .
      thread_id = i, .G_dt = G_dt, .delta_t = delta_t, .
      particle_array = p, .N = N};
179     // printf("lbound: %d, ubound %d\n", thread_data[i].lower_bound
      , thread_data[i].upper_bound);
180   }
181
182   if (graphics)
183   {
184     // make radius dependent on the mass of the particle
185     for (register int i = 0; i < N; ++i)
186     {
187       circleradius[i] = p[i].mass * 1.5e-3;
188     }
189   }
190   for (int thread = 0; thread < nthreads; thread++)
191     pthread_create(&threads[thread], &attr, thread_force_update, &
      thread_data[thread]);
192
193
194
```

```
195   for (int thread = 0; thread < nthreads; thread++)
196     pthread_join(threads[thread], NULL);
197 
198   pthread_barrier_destroy(&barrier);
199 
200   //after all time steps, joint the threads
201 }
202 
203 void *thread_force_update(void *args)
204 {
205   thread_data_t *t_data = (thread_data_t *)args;
206   p_t *p = t_data->particle_array;
207   vector_t diff, diffsqrd;
208   double distance;
209 
210   for (int t=0; t<t_data->nsteps; t++) {
211   pthread_barrier_wait(&barrier);
212   for (int i = t_data->lower_bound; i < t_data->upper_bound; i++)
213   {
214     for (int j = 0; j < t_data->N; j++) // only up til the particle
       in the above foor loop
215     {
216       if (i == j)
217         continue;
218       // temp variables
219       diff = p[i].pos - p[j].pos; // particles relative position in
       x,y coordinates
220       diffsqrd = diff * diff;     // - squared
221       distance = sqrt(diffsqrd[0] + diffsqrd[1]) + EPSILON_0;
222       p[i].acc -= p[j].mass * diff / (distance * distance *
      distance);
223     }
224   }
225 
226   pthread_barrier_wait(&barrier);
227 
228   for (int i = t_data->lower_bound; i < t_data->upper_bound; i++)
229   {
230     p[i].vel += t_data->G_dt * p[i].acc;
231     p[i].pos += t_data->delta_t * p[i].vel;
232     p[i].acc = zerovec;
233   }
234   }
235   pthread_exit(NULL);
236 }
```

Multithreaded Barnes-Hut galsim.c

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <math.h>
4 #include <pthread.h>
5 #include "graphics.h"
6 
7 #define PRINT_TIMES 0 //set to 1 to print execution times of
    different code blocks
```

```
8
9  #define EPSILON_0 1e-3     //Plummer radius
10 #define W 1                // Window width for graphics
11 #define L 1                // Window length for graphics
12
13
14 /*
15 Globals for Pthread
16 */
17 pthread_barrier_t barrier;
18 int done; //global variable so that a worker thread can know once
       all timesteps are done
19
20
21 typedef double vector_t __attribute__((vector_size(2 * sizeof(
       double))));
22
23 /*
24     Particle:
25     * pos - coordinates
26     * mass - mass of particle
27     * vel - velocity
28     * brightness - brightness of particle
29     * acc - acceleration
30 */
31 typedef struct particle
32 {
33     vector_t pos;
34     double mass;
35     vector_t vel;
36     double brightness;
37     vector_t acc;
38 } p_t;
39
40 /*
41     Quad tree node:
42     * particle - particle struct pointer
43     * length - length of box/quadnode
44     * total_mass - mass of all particles inside a box
45     * center_mass - position of center of mass
46     * pos           - coordinate of center of box
47     * NE,NW,SW,SE   - quadrants of domain. NE = norteast, etc.. (
       these are the nodes children)
48 */
49 typedef struct quad_tree_node
50 {
51     p_t *particle;
52     double length;
53     double total_mass;
54     vector_t center_mass;
55     vector_t pos;
56     struct quad_tree_node *NE;
57     struct quad_tree_node *NW;
58     struct quad_tree_node *SW;
59     struct quad_tree_node *SE;
60
61 } q_node_t;
```

```
62
63 /*
64 struct that contains all the relevant data for a thread
65 */
66 typedef struct thread_data
67 {
68     uint lower_bound;
69     uint upper_bound;
70     uint thread_id;
71     double theta_max;
72     double G_dt;
73     double delta_t;
74     q_node_t * root;
75     p_t * particle_array;
76 } thread_data_t;
77
78 /*
79     Initialize global helpfull vectors
80 */
81 const vector_t zerovec = {0.0, 0.0};
82 const vector_t se_vec = {1.0, 1.0};
83 const vector_t sw_vec = {-1.0, 1.0};
84 const vector_t ne_vec = {1.0, -1.0};
85 const vector_t nw_vec = {-1.0, -1.0};
86
87
88
89 /*
90 Macro function that creates a child to the parent, with a vector
91 in the corresponding direction.
92     Example: Create a child to the north-west to a q_node_t ** node
93     create_child((*node)->NW, (*node), nw_vec);
94 */
95 #define create_child(child, parent, dir) \
96 ({ \
97   child = (q_node_t *) malloc(sizeof(q_node_t)); \
98   child->pos = parent->pos + 0.25 * (parent->length) * dir; \
99   child->length = parent->length * 0.5; \
100   child->total_mass = 0.0; \
101   child->particle = NULL; \
102   child->NW = NULL; \
103   child->NE = NULL; \
104   child->SW = NULL; \
105   child->SE = NULL; \
106   })
107
108
109 /*
110 Macro function that locates the correct child quadrant to insert a
    particle
111 into. If val==0: check if child already exists. If not, create it
    before inserting.
112 If val==1: insert knowing that child does not exist.
113 */
114 #define split_insert(val,particle) \
115 ({ \
116     if ((particle->pos[0] >= (*node)->pos[0]) && (particle->pos[1]
```

```
    < (*node)->pos[1])) { \
117         if (val||(*node)->NE == NULL) { \
118             create_child((*node)->NE, (*node), ne_vec); \
119         } \
120         insert(&((*node)->NE), particle); \
121     } \
122     else if ((particle->pos[0] < (*node)->pos[0]) && (particle->pos
    [1] < (*node)->pos[1])){ \
123         if (val||(*node)->NW == NULL) { \
124             create_child((*node)->NW, (*node), nw_vec); \
125         } \
126         insert(&((*node)->NW), particle); \
127     } \
128     else if ((particle->pos[0] < (*node)->pos[0]) && (particle->pos
    [1] >= (*node)->pos[1])) { \
129         if (val||(*node)->SW == NULL) { \
130             create_child((*node)->SW, (*node), sw_vec); \
131         } \
132         insert(&((*node)->SW), particle); \
133     } \
134     else { \
135         if (val||(*node)->SE == NULL) { \
136             create_child((*node)->SE, (*node), se_vec); \
137         } \
138         insert(&((*node)->SE), particle); \
139     } \
140     })
141
142
143
144 /*
145     Declare functions
146 */
147
148 //IO functions
149 void read_file(int N, p_t *p, const char *fileName);
150 void write_to_file(p_t *p, int N, char *fileName);
151 void print_struct(p_t p); //mainly for debugging
152 double get_wall_seconds(void); //measuring time
153
154 q_node_t *init(); //create tree root
155 void insert(q_node_t **node, p_t *particle); //recursive insertion
    of particle into tree
156 void tree_del(q_node_t **node); //free memory recursivly
157 void barnes_hut(const int nsteps, const double delta_t, p_t *p,
    const int N,
158                 const double theta_max, const int graphics, const
    int nthreads); //the time stepping algorithm
159 void *thread_force_update(void * args); //thread force
160 void force_traverse(q_node_t **node, const double theta_max, p_t *
    particle); //calculation of the forces
161
162 void tree_draw(q_node_t **node); //recursive drawing of the
    quadnodes
163
164
165
```

```
166 /*
167     Driver function
168 */
169 int main(int argc, char *argv[])
170 {
171     #if PRINT_TIMES
172     double main_time = get_wall_seconds();
173     #endif
174 
175 
176 
177     // Check correct number of input arguments
178     if (argc != 8)
179     {
180         printf("Wrong number of inputs \n");
181         exit(EXIT_FAILURE);
182     }
183 
184     // Define input variables
185     const int N = atoi(argv[1]);
186     const int nsteps = atoi(argv[3]);
187     const double delta_t = atof(argv[4]);
188     const double theta_max = atof(argv[5]) * atof(argv[5]); //
    square for cheaper comparison in the force calculation
189     const int graphics = atoi(argv[6]);
190     const int nthreads = atoi(argv[7]);
191 
192     //Read from input file and save in an array of particles
193     p_t p[N];
194 
195     #if PRINT_TIMES
196         double read_time = get_wall_seconds();
197         read_file(N, p, argv[2]);
198         read_time= get_wall_seconds() - read_time;
199         printf("Time spent reading input file: %lf s \n", read_time
    );
200     #else
201         read_file(N, p, argv[2]);
202     #endif
203 
204 
205     // initialize window if graphics is enabled
206     if (graphics) InitializeGraphics(argv[0], 800, 800);
207 
208     //Simulate particle system with Barnes Hut algorithm and save
    result
209     barnes_hut(nsteps, delta_t, p, N, theta_max, graphics, nthreads
    );
210 
211     #if PRINT_TIMES
212         double write_time = get_wall_seconds();
213         write_to_file(p, N, "result.gal");
214         write_time= get_wall_seconds() - write_time;
215         printf("Time spent saving input file: %lf s \n", write_time
    );
216 
217         main_time = get_wall_seconds() - main_time;
```

```
218         printf("--------\nTotal execution time: %lf s \n", main_time
    );
219     #else
220         write_to_file(p, N, "result.gal");
221     #endif
222 
223     return 0;
224 }
225 
226 /*
227     Function used for measuring times
228 */
229 double get_wall_seconds(void)
230 {
231     struct timeval tv;
232     gettimeofday(&tv, NULL);
233     double seconds = tv.tv_sec + (double)tv.tv_usec / 1000000;
234     return seconds;
235 }
236 
237 /*
238     Function that reads doubles from input file and stores them in
    an array of structs
239 */
240 void read_file(int N, p_t *p, const char *fileName)
241 {
242     /* Open input file and determine its size. */
243     FILE *file = fopen(fileName, "rb");
244     if (!file)
245     {
246         printf("read_doubles_from_file error: failed to open input
    file '%s'.\n",
247                 fileName);
248         exit(EXIT_FAILURE);
249     }
250 
251     p_t temp;
252     double buffer[6]; // file reading buffer
253     for (int i = 0; i < N; i++)
254     {
255         // Read doubles into particle struct
256         if (0==fread(&buffer, sizeof(double), 6, file)) exit(
    EXIT_FAILURE);
257         temp.pos[0] = buffer[0];
258         temp.pos[1] = buffer[1];
259         temp.mass = buffer[2];
260         temp.vel[0] = buffer[3];
261         temp.vel[1] = buffer[4];
262         temp.brightness = buffer[5];
263         temp.acc = zerovec;
264         p[i] = temp;
265     }
266     // Close file and check if closed successfully
267     if (fclose(file) != 0)
268     {
269         printf("read_doubles_from_file error: error closing input
    file.\n");
```

```
270         exit(EXIT_FAILURE);
271     }
272 }
273
274 /*
275     Function that writes output to file
276 */
277 void write_to_file(p_t *p, int N, char *fileName)
278 {
279     FILE *file = fopen(fileName, "wb");
280
281     for (int i = 0; i < N; i++)
282     {
283         fwrite(&p[i].pos, sizeof(p[i].pos), 1, file);
284         fwrite(&p[i].mass, sizeof(p[i].mass), 1, file);
285         fwrite(&p[i].vel, sizeof(p[i].vel), 1, file);
286         fwrite(&p[i].brightness, sizeof(p[i].brightness), 1, file);
287     }
288     fclose(file);
289 }
290
291 /*
292     Function that prints values of a struct. Used for debugging
293 */
294 void print_struct(p_t p)
295 {
296     printf(
297         "x: %lf\t y: %lf\t mass: %lf\t vel_x: %lf\t vel_y: %lf\t
    brightness: "
298         "%lf\n",
299         p.pos[0], p.pos[1], p.mass, p.vel[0], p.vel[1], p.
    brightness);
300 }
301
302 /*
303     Function to initialize the root of the the quad tree
304 */
305 q_node_t *init()
306 {
307     q_node_t *root = (q_node_t *)malloc(sizeof(q_node_t));
308
309     //define the initial data
310     root->particle = NULL;
311     root->length = 1.0;
312     root->total_mass = 0.0;
313     root->center_mass = zerovec;
314     root->pos= 0.5 * se_vec;
315
316     //create empty children to the node
317     root->NE = NULL;
318     root->NW = NULL;
319     root->SW = NULL;
320     root->SE = NULL;
321
322     return root;
323 }
324
```

```
325
326 /*
327 Recursive insertion of a particle into the quadtree.
328 Updates mass + center of mass of each node,  then creates the
        required child node and inserts
329 particle into it
330 */
331 void insert(q_node_t **node, p_t *particle)
332 {
333     //update mass and center of mass
334     (*node)->center_mass = ((*node)->total_mass * (*node)->
    center_mass + particle->mass * particle->pos) / ((*node)->
    total_mass + particle->mass);
335     (*node)->total_mass = (*node)->total_mass + particle->mass;
336
337     //check if a particle is present at the node
338     if ((*node)->particle == NULL)
339     { // if there is no particle, then check if it has children
340
341         if ((*node)->NW == NULL && (*node)->NE == NULL && (*node)->
    SE == NULL && (*node)->SW == NULL)
342         {                                   // if it doesnt have
    any children,
343             (*node)->particle = particle;   // this means there is
    no particle or sub-quadrants. Free to add the new particle!!
344             return;
345         }
346         else //insert into the correct child
347         {
348             split_insert(0,particle); //finds the correct child,
    creates it if not existent, and inserts it into that node
349         }
350     }
351     else
352     {
353         //the particle in the node needs to be inserted into a
    child instead
354         p_t *old_particle = (*node)->particle;
355         (*node)->particle = NULL;
356
357         // first argument (1) makes if statement in macro function
    obsolete since if statement
358         // will allways be true
359
360         //insert the particles and create children if needed
361         split_insert(1,particle);
362         split_insert(0,old_particle);
363     }
364 }
365
366 /*
367     Delete tree recursivly, starting at the root
368 */
369 void tree_del(q_node_t **node)
370 {
371     //check for the existance of children, and delete those that
    are present
```

```
372     if ((*node) != NULL) {
373     if ((*node)->NW != NULL)
374         tree_del(&(**node).NW);
375 
376     if ((*node)->NE != NULL)
377         tree_del(&(**node).NE);
378 
379     if ((*node)->SE != NULL)
380         tree_del(&(**node).SE);
381 
382     if ((*node)->SW != NULL)
383         tree_del(&(**node).SW);
384     }
385     free(*node);
386     *node=NULL;
387 }
388 
389 /*
390     Traverse down the quad tree and calculate what forces effect a
        particle,
391     and update its acceleration.
392 */
393 void force_traverse(q_node_t **node, const double theta_max, p_t *
        particle)
394 {
395     //If quadrant is empty or particle is current particle
396     if ((*node) == NULL || (*node)->total_mass == 0 || particle ==
        (*node)->particle)
397     {
398         return;
399     }
400 
401     //Get the distance
402     vector_t diff = particle->pos - (*node)->pos;       // particles
         relative position in x,y coordinates
403     vector_t diffsqrd = diff * diff;                    // squared
404     double distance = diffsqrd[0] + diffsqrd[1]; //squared distance
405 
406 
407     //If theta condition is met calculate acceleration of particle
408     //else traverse down to child nodes
409 
410     //NOTE: the original condition is theta_max >= length/distance,
         but below the
411     //comparison is instead theta_max^2 * distance^2 >= lenght^2.
        Thus we don't need
412     //to take the sqrt for the distance above. ~20% time reduction
        for N=10000 from this
413 
414     if (theta_max * distance>= ( ((*node)->length) * ((*node)->
        length) )  || (*node)->particle != NULL)
415     {
416         diff = particle->pos - (*node)->center_mass;            //
        particles relative position in x,y coordinates
417         diffsqrd = diff * diff;                                 //
        squared
418         distance = sqrt(diffsqrd[0] + diffsqrd[1]) + EPSILON_0; //
```

```
    pythagoras
419         particle->acc -= (*node)->total_mass * diff / (distance *
    distance * distance);
420     }
421     else
422     {
423         //if theta condition not fulfilled, need to take another
424         //recursive step to each of the children
425         if ((*node)->NW != NULL) force_traverse(&((*node)->NW),
    theta_max, particle);
426         if ((*node)->NE != NULL) force_traverse(&((*node)->NE),
    theta_max, particle);
427         if ((*node)->SW != NULL) force_traverse(&((*node)->SW),
    theta_max, particle);
428         if ((*node)->SE != NULL) force_traverse(&((*node)->SE),
    theta_max, particle);
429     }
430 }
431
432 /*
433     Barnes Hut algorithm
434 */
435 void barnes_hut(const int nsteps, const double delta_t, p_t *p,
    const int N, const double theta_max, const int graphics, const
    int nthreads)
436 {
437
438     #if PRINT_TIMES //declare variables if we want ot emasure times
439             double build_time=0.0;
440             double force_time=0.0;
441             double delete_time=0.0;
442     #endif
443
444     // if we use graphics initialize window
445     float circleradius[N], circlecolor = 0;
446     if (graphics) {
447         SetCAxes(0, 1);
448
449         // make radius dependent on the mass of the particle
450         for (register int i = 0; i < N; ++i)
451         {
452         circleradius[i] = fmax(1.5e-3, p[i].mass * 1.5e-3); //have
    a smallest possible particle size
453         }
454
455     }
456
457     const double G_dt = (100.0 * delta_t) / N;
458
459
460     pthread_t threads[nthreads];
461     thread_data_t  thread_data[nthreads];
462     pthread_barrier_init(&barrier, NULL, nthreads+1); //+1 since
    main is also waiting
463
464
465
```

```
466     q_node_t *root = init();
467     done = 0; //0 means we are not done with simulation
468 
469 
470     for (int i=0; i<nthreads; i++) {
471         thread_data[i] = (thread_data_t) { .lower_bound = i*N/
    nthreads, .upper_bound=(i+1)*N/nthreads,
472                         .thread_id=i, .theta_max=theta_max, .G_dt=
    G_dt, .delta_t=delta_t,
473                         .root = root, .particle_array=p};
474         // printf("lbound: %d, ubound %d\n", thread_data[i].
    lower_bound, thread_data[i].upper_bound);
475     }
476 
477     for (int thread=0; thread<nthreads; thread++) {
478             pthread_create(&threads[thread], NULL,
    thread_force_update, &thread_data[thread]);
479     }
480 
481 
482 
483     for (register int t = 0; t < nsteps; t++)
484     {
485 
486         #if PRINT_TIMES
487             build_time -= get_wall_seconds();
488             for (register int i = N - 1; i >= 0; i--) insert(&root,
     &(p[i]));
489             build_time += get_wall_seconds();
490         #else
491             for (register int i = N - 1; i >= 0; i--) insert(&root,
     &(p[i]));
492         #endif
493 
494         pthread_barrier_wait(&barrier); //unlocks barrier to start
    time steps in threads
495 
496 
497         #if PRINT_TIMES
498             force_time -= get_wall_seconds();
499             pthread_barrier_wait(&barrier);
500             force_time += get_wall_seconds();
501         #else
502             pthread_barrier_wait(&barrier);
503         #endif
504 
505         if (graphics) {
506             ClearScreen();
507             // picasso away!
508             for (int i = 0; i < N; i++) {
509                 DrawCircle(p[i].pos[0], p[i].pos[1], W, L,
    circleradius[i], circlecolor);
510             }
511             tree_draw(&root);
512 
513             Refresh();
514             usleep(3000);
```

```
515         }
516
517         // tree_del(&root);
518
519         //TEMP BLOCK
520         #if PRINT_TIMES
521             delete_time -= get_wall_seconds();
522             tree_del(&(root->NE)); tree_del(&(root->NW)); tree_del
    (&(root->SW)); tree_del(&(root->SE));
523             delete_time += get_wall_seconds();
524         #else
525             tree_del(&(root->NE)); tree_del(&(root->NW)); tree_del
    (&(root->SW)); tree_del(&(root->SE));
526         #endif
527
528
529         //RESET ROOT: SAME AS INIT BUT NO MALLOC. THIS WAY WE CAN
    KEEP THE ADRESS TO ROOT
530         //AND NOT CHANGE IT EACH TIME STEP
531         root->particle = NULL;
532         root->total_mass = 0.0;
533         root->center_mass = zerovec;
534         root->NE = NULL;
535         root->NW = NULL;
536         root->SW = NULL;
537         root->SE = NULL;
538
539     }
540
541     done=1;
542     pthread_barrier_wait(&barrier);
543     free(root);
544
545
546
547     // //after all time steps, joint the threads
548     for (int thread=0; thread<nthreads; thread++) pthread_join(
    threads[thread], NULL);
549     pthread_barrier_destroy(&barrier); //destroy barrier once it is
     no longer used
550     // if we use graphics, flush and close window
551     if (graphics) {
552         FlushDisplay();
553         CloseDisplay();
554     }
555
556     #if PRINT_TIMES
557         printf("Time spent building the quadtree: %lf s\n",
    build_time);
558         printf("Time spent calculating forces and updating
    particles: %lf s\n", force_time);
559         printf("Time spent on deleting the quadtree: %lf s\n",
    delete_time);
560     #endif
561 }
562
563 void *thread_force_update(void * args) {
```

```
564     thread_data_t * t_data = (thread_data_t *) args;
565     p_t * p = t_data->particle_array;
566 
567     while( True ) {
568         pthread_barrier_wait(&barrier); //wait until all threads
    are in position to start time step (tree built)
569         if (done) break; //we set done to 1 in barnes_hut() after
    all time steps complete
570 
571         for (uint i = t_data->lower_bound; i<t_data->upper_bound; i
    ++) {
572                 force_traverse(&(t_data->root), t_data->theta_max,
    &(p[i]));
573                 p[i].vel += t_data->G_dt * p[i].acc;
574                 p[i].pos += t_data->delta_t * p[i].vel;
575                 p[i].acc = zerovec;
576         }
577 
578         pthread_barrier_wait(&barrier);     //wait until all
    threads done (s.t. master can start deleting)
579     }
580 
581 
582 }
583 
584 
585 
586 /*
587     Draws quadrants recursivly, starting with the root
588 */
589 void tree_draw(q_node_t **node){
590     if(&(**node) != NULL){
591         DrawRectangle((**node).pos[0]-(((**node).length)/2.0), (**
    node).pos[1]-(((**node).length)/2.0), W, L, (**node).length,
    (**node).length, 0.85);
592         tree_draw(&(**node).NW);
593         tree_draw(&(**node).SW);
594         tree_draw(&(**node).NE);
595         tree_draw(&(**node).SE);
596     }
597 }
```