

# The Gravitational N-body System

Emil Tullstedt

2016-03-25

# Contents

<b>1</b>	<b>Introduction</b>	<b>2</b>
<b>2</b>	<b>Programs</b>	<b>2</b>
2.1	Naïve Sequential Program . . . . .	2
2.1.1	Calculating Forces . . . . .	2
2.1.2	Applying Forces . . . . .	3
2.2	Naïve Parallel Program . . . . .	4
2.3	Barnes-Hut Sequential Program . . . . .	4
2.4	Barnes-Hut Parallel Program . . . . .	6
<b>3</b>	<b>Evaluation</b>	<b>6</b>
<b>4</b>	<b>Conclusion</b>	<b>7</b>
	<b>Appendices</b>	<b>10</b>
<b>A</b>	<b>Performance.py Time Results</b>	<b>10</b>
<b>B</b>	<b>Common Listings for all applications</b>	<b>11</b>
<b>C</b>	<b>Listings for section 2.1</b>	<b>14</b>
<b>D</b>	<b>Listings for section 2.2</b>	<b>16</b>
<b>E</b>	<b>Listings for section 2.3</b>	<b>19</b>
<b>F</b>	<b>Listings for section 2.4</b>	<b>27</b>
<b>G</b>	<b>Broken parallelization of 2.4</b>	<b>36</b>
<b>H</b>	<b>Listings for help script plot.py</b>	<b>40</b>
<b>I</b>	<b>Listings for help script performance.py</b>	<b>41</b>
<b>J</b>	<b>Listings for Makefile</b>	<b>42</b>

# 1 Introduction

*The gravitational n-body problem* simulates how the gravitational force affects a number of bodies within a system of particles. By parallelising and approximating gravitational forces simulations can spend less time calculating the effects of different gravitational bodies reacting to each other. Efficient calculations of how particles interact to each other is crucial for astronomy to be aware of satellites, space debris, asteroids et.c. will affect human life both short- and long-term.

The simulations described in this report are two-dimensional and rudimentary as to show the different methods of increasing efficiency rather than constructing useful simulation data. A video showing a simulation in action can be seen on YouTube, <https://www.youtube.com/watch?v=AH0PfpAryfc> or generated as a mpeg4-video using the `plot.py` helper application described in appendix H together with the `output` file from the `*.debug.out`-versions of the program.

## 2 Programs

### 2.1 Naïve Sequential Program

The naive sequential program does two things, it calculates how two bodies affect each other and it uses these calculations to move the bodies. The simulation uses steps over  $dt$  rather than continuous calculations.

#### 2.1.1 Calculating Forces

In order to calculate the forces acting on every particle, for every step and every particle  $i$  the distance  $p(ij)$  between that particle and every other particle  $j$  is calculated using Pythagora's Theorem is applied to find the shortest distance between the two particles.

$$p(ij) = \sqrt{(p(i)_x - p(j)_x)^2 + (p(i)_y - p(j)_y)^2} \quad (1)$$

The exponentially diminishing effect of distance  $p(ij)$  to the gravitational force is then applied to Newton's gravitational constant  $G = 6.67 * 10^{-11}$  and the mass of the two particles using

$$m(ij) = \frac{G * m(i) * m(j)}{p(ij)^2} \quad (2)$$

The direction of the movement is calculated for the  $x$  and  $y$  axis

$$d(ij)_x = p(j)_x - p(i)_x \quad (3)$$

$$d(ij)_y = p(j)_y - p(i)_y \quad (4)$$

Finally, these calculations are applied on the current force in both directions of the particle  $i$

$$f(i)_x = f(i)_x + \frac{m(ij) * d(ij)_x}{p(ij)} \quad (5)$$

$$f(i)_y = f(i)_y + \frac{m(ij) * d(ij)_y}{p(ij)} \quad (6)$$

Furthermore, as the forces  $f(i)_x$  and  $f(j)_x$  are each other's opposites the forces  $f(j)_{xy}$  can be calculated with subtraction of the force rather than addition of it in order to avoid having to calculate both  $p(ij)$  and  $p(ji)$  (which are equal),  $m(ij)$  and  $m(ji)$  (also equal), and  $d(ij)$  and  $d(ji)$  (which are opposites).

$$f(j)_x = f(i)_x - \frac{m(ij) * d(ij)_x}{p(ij)} \quad (7)$$

$$f(j)_y = f(i)_y - \frac{m(ij) * d(ij)_y}{p(ij)} \quad (8)$$

This operation is  $O(n^2)$  for  $n$  particles.

### 2.1.2 Applying Forces

Applying the forces is an  $O(n)$  operation once the calculations in section 2.1.1 are applied. For every particle  $i$  the following equation is ran to get the new velocity and position of the particle.

$$\delta v(i)_x = \frac{f(i)_x}{m(i)} * dt \quad (9)$$

$$\delta v(i)_y = \frac{f(i)_y}{m(i)} * dt \quad (10)$$

$$\delta p(i)_x = (v(i)_x + \frac{\delta v(i)_x}{2}) * dt \quad (11)$$

$$\delta p(i)_y = (v(i)_y + \frac{\delta v(i)_y}{2}) * dt \quad (12)$$

$$v(i)_x = v(i)_x + \delta v(i)_x \quad (13)$$

$$v(i)_y = v(i)_y + \delta v(i)_y \quad (14)$$

$$p(i)_x = p(i)_x + \delta p(i)_x \quad (15)$$

$$p(i)_y = p(i)_y + \delta p(i)_y \quad (16)$$

And finally resetting the forces in both directions  $f(i)$  to 0.

## 2.2 Naïve Parallel Program

Parallelising the sequential application described in section 2.1 is primarily a matter of changing the for loops in such a manner so that every thread will take care of individual particles  $i$ . In order to keep the performance gain by applying the forces bi-directionally the force  $f(i)$  is stored as  $f(i(t))$  which are then summed together  $f(i) = f(i(t_0)) + f(i(t_1)) + \dots + f(i(t_n))$  for  $n$  threads before applying the forces.

A “barrier” between the calculation and application of the forces is necessary to avoid threads from applying unfinished data and corrupting the output. This barrier was implemented using a shared counter  $b(c)$  and a conditional variable  $b(v)$  where a barrier is initialised by setting a value  $b(c) \in \mathbb{Z}^+$  for the  $c$  threads which must meet at the barrier in order for the conditional variable  $b(v)$  to signal all threads to continue.

## 2.3 Barnes-Hut Sequential Program

The Barnes-Hut approximation of a gravitational n-body system uses the assumption that any particle far away enough may be calculated as a sum of the particles in that particle’s proximity. This assumption works in the same way as if someone would answer “1067” on the question “Which year did the Battle of Hastings break out?” that answer would be close to the correct answer, whereas if someone were to pay their rent one year overdue,

their landlord would probably not be as forgiving as a history teacher might be about misplacing the Battle of Hastings.

The implementation of the approximation was made with a tree of quadrants where every quadrant leaf represents  $0 \leq n \leq 5$  particles. If a quadrant leaf is filled with more than 5 particles, it would split into four new quadrants and thus become a quadrant branch. When the entire tree is composed from the particles in the particle system the sum of the mass of every particle within every quadrant branch and leaf is combined with the position of it's children (i.e. quadrant branches or leaves for a quadrant branch and particles for a quadrant leaf) to create the center of mass for the quadrant.

The center of mass calculation used is

$$com(q) = \frac{m(1) * p(1) + m(2) * p(2) + \dots + m(n) * p(n)}{m(1) + m(2) + \dots + m(n)} \quad (17)$$

The force calculations are based on a further recursive algorithm where the distance between a quadrant and every particle  $i$  is calculated with Pythagora's theorem to the nearest point of the quadrant. The nearest point calculation is based on the algorithm below.

```

if  $p(i)_x > q_e$  then
     $\delta x \leftarrow p(i)_x - q_e$ 
else if  $p(i)_x < q_w$  then
     $\delta x \leftarrow q_w - p(i)_x$ 
else
     $\delta x \leftarrow 0$ 
end if
if  $p(i)_y < q_s$  then
     $\delta y \leftarrow q_s - p(i)_y$ 
else if  $p(i)_y > q_n$  then
     $\delta y \leftarrow p(i)_y - q_n$ 
else
     $\delta y \leftarrow 0$ 
end if

```

If the distance between the quadrant and the particle is greater than the cutoff distance defined either at runtime or as a compile time constant the particle's force calculation is based on that quadrant's center of mass and mass sum. Otherwise, if the distance is less than the cutoff distance but the quadrant is a quadrant branch (and has more than 5 particles), the function recursively calls all the containing quadrants within the quadrant branch.

When the recursive function reaches a quadrant leaf and the distance is within the cutoff distance the regular function for applying forces on the in-

dividual particles is used, with the modification that the “equal but opposing force”- optimization isn’t used.

The opposing force optimization from sections 2.1 and 2.2 isn’t used in this context as the risk that the assumption would insert errors into calculations were prominent. In a real-world implementation, proving that this kind of optimization was possible without compromising the integrity of the simulation could prove useful.

## 2.4 Barnes-Hut Parallell Program

Parallellizing the Barnes-Hut application from section 2.3 proved to be a complicated problem. Initially, the same methods that were applied to the parallellization in section 2.2 was applied to the calculation and application of forces and then an attempt at parallelizing the division of quadrants was made but unsuccessfully due to segmentation faults and introducing *NaN*-values during the floating point calculations. Two distinct attempts were made, one which is presented as a diff format in appendix G.

The theory for further parallellizing the program is that the quadrants can be independently calculated without interference from any parent quadrant. The problem showed to be to deterministically decide if a quadrant is calculated or not and to avoid dead-locks where the different processes depend on each other when adding up the quadrant branches.

The performance gain from simply parallelizing the calculating and applying parts of the application were minor, but still existing and useful (especially when testing with 1000+ particles, before which the synchronizations caused the application to not show any major gains).

## 3 Evaluation

The evaluation of the programs developed was done using the `performance.py` script from appendix I which runs every test value/program pair five times and picks the median time from those to present to `stdout`.

Beginning with 120 particles (over 50 000 steps of time) the performance presented in table 1 and figure 1 was not very surprising, the parallell applications on single cores performed slightly worse than the sequential did and the parallell naïve program got almost 2X performance improvement with 3 cores (and no further with 4 cores). The most surprising part was that the performance gains with parallelisation for the Barnes-Hut model was negative in all cases and performed *worse* when adding more cores. Synchronization

overhead for the model is prominent and since  $n \log n$  for 120 is only a few hundred comparisons, this result isn't entirely unexpected.

The real differences between the Barnes-Hut model and the naïve model is much more obvious the more particles that are added, as seen in figure 2. The performance gain from parallelization of the naïve implementation are clear in this graph being twice as good in the beginning and delivering three times the performance on 240 elements. As time spent with thread overhead is *near* constant no matter the amount of particles, the parallel programs will show closer to optimal performance gain the more particles are added. The same property is true for the Barnes-Hut model where the development is logarithmic.

To test this theory, I ran the program with 2000 particles for 2000 time steps on my laptop and got the result showed in table 2 where the results for the Barnes-Hut model is both vastly better and there is real performance gain in the parallelization.

## 4 Conclusion

The gravitational n-body problem proved to be an interesting problem to try to optimize as two vastly different methods proved to be useful in improving the performance of the problem. If I had more time to debug the solution presented to the parallelization of the Barnes-Hut approximation I am fairly sure that results that would've ranged from  $O(n^2)t \rightarrow \frac{O(n \log n)t}{3}$  could've appeared in table 2 when running on 4 cores rather than the current range from  $O(n^2)t \rightarrow \frac{O(n \log n)t}{2}$ .

During the course of this project I've learned more about architecture of parallel programs in order to gain maximal performance and visualizations of  $O(n^2)$  vs  $O(n \log n)$ .



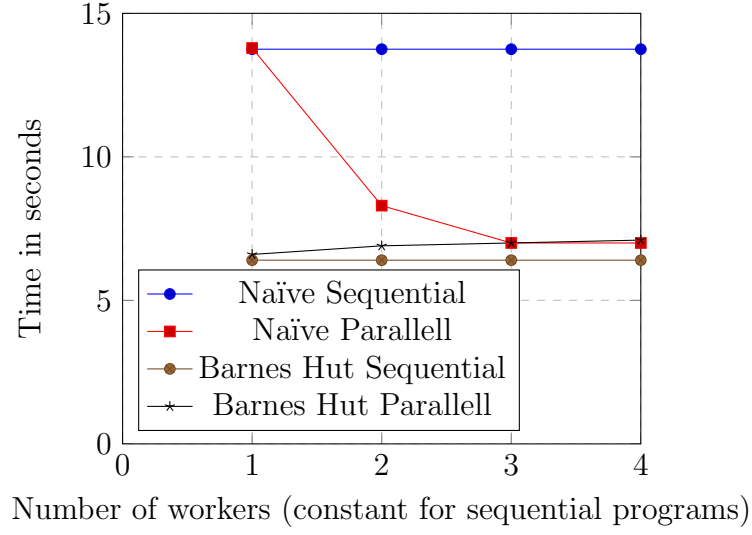


Figure 1: Performance with 120 particles

Name	Workers	$t(120)$	$t(180)$	$t(240)$
Naïve Sequential	N/A	13.76	30.85	54.81
Naïve Parallell	1	13.79	30.91	54.97
Naïve Parallell	2	8.3	16.68	29.19
Naïve Parallell	3	7.0	12.51	27.99
Naïve Parallell	4	7.0	10.29	18.67
Barnes-Hut Sequential	N/A	6.4	10.45	15.22
Barnes-Hut Parallell	1	6.6	10.76	15.74
Barnes-Hut Parallell	2	6.9	10.58	14.89
Barnes-Hut Parallell	3	7.0	9.13	13.70
Barnes-Hut Parallell	4	7.1	9.93	13.24

Table 1:  $t(n)$  s performance with  $n$  particles over 50 000 time steps

Name	Workers	$t(2000)$
Naïve Sequential	N/A	51.29
Naïve Parallell	1	51.29
Naïve Parallell	4	24.47
Barnes-Hut Sequential	N/A	3.94
Barnes-Hut Parallell	1	4.34
Barnes-Hut Parallell	4	2.96

Table 2:  $t(n)$  s performance with  $n$  particles over 2000 time steps

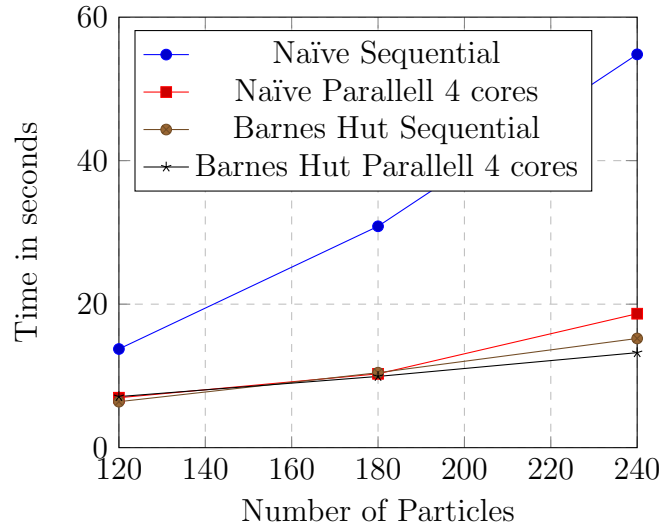


Figure 2: Performance Development with more Particles

# Appendices

## A Performance.py Time Results

```
1 <KTH Shell>
2
3 ('./par_nlg.out', 120, 6.6356520000000003, 1)
4 ('./par_nlg.out', 120, 6.906879, 2)
5 ('./par_nlg.out', 120, 7.0215680000000003, 3)
6 ('./par_nlg.out', 120, 7.1209990000000003, 4)
7 ('./par_nlg.out', 180, 10.765097000000001, 1)
8 ('./par_nlg.out', 180, 10.588206, 2)
9 ('./par_nlg.out', 180, 9.1330360000000006, 3)
10 ('./par_nlg.out', 180, 9.9337389999999992, 4)
11 ('./par_nlg.out', 240, 15.738244, 1)
12 ('./par_nlg.out', 240, 14.88978, 2)
13 ('./par_nlg.out', 240, 13.702691, 3)
14 ('./par_nlg.out', 240, 13.239893, 4)
15 ('./par_sq.out', 120, 13.792294999999999, 1)
16 ('./par_sq.out', 120, 8.3339909999999993, 2)
17 ('./par_sq.out', 120, 7.0577490000000003, 3)
18 ('./par_sq.out', 120, 6.9642720000000002, 4)
19 ('./par_sq.out', 180, 30.919129999999999, 1)
20 ('./par_sq.out', 180, 16.681785999999999, 2)
21 ('./par_sq.out', 180, 12.509850999999999, 3)
22 ('./par_sq.out', 180, 10.286918, 4)
23 ('./par_sq.out', 240, 54.972634999999997, 1)
24 ('./par_sq.out', 240, 29.186340999999999, 2)
25 ('./par_sq.out', 240, 27.993697999999998, 3)
26 ('./par_sq.out', 240, 18.670895000000002, 4)
27 ('./seq_sq.out', 120, 13.759906000000001, 1)
28 ('./seq_sq.out', 180, 30.846284000000001, 1)
29 ('./seq_sq.out', 240, 54.812840000000001, 1)
30 ('./seq_nlg.out', 120, 6.4147290000000003, 1)
31 ('./seq_nlg.out', 180, 10.448035000000001, 1)
32 ('./seq_nlg.out', 240, 15.222167000000001, 1)
33
34 <Lenovo X250 w/ dual-core i3 + hyper-threading>
35 ('./par_nlg.out', 120, 2.386097, 1)
36 ('./par_nlg.out', 120, 2.720354, 2)
37 ('./par_nlg.out', 120, 3.174221, 3)
38 ('./par_nlg.out', 120, 3.484154, 4)
39 ('./par_nlg.out', 180, 4.270851, 1)
40 ('./par_nlg.out', 180, 4.490971, 2)
41 ('./par_nlg.out', 180, 5.031048, 3)
42 ('./par_nlg.out', 180, 6.069867, 4)
43 ('./par_nlg.out', 240, 6.802512, 1)
```

```

44 ('./par_nlg.out', 240, 6.301626, 2)
45 ('./par_nlg.out', 240, 8.207197, 3)
46 ('./par_nlg.out', 240, 6.441181, 4)
47 ('./par_sq.out', 120, 4.556379, 1)
48 ('./par_sq.out', 120, 3.33266, 2)
49 ('./par_sq.out', 120, 3.323491, 3)
50 ('./par_sq.out', 120, 3.296448, 4)
51 ('./par_sq.out', 180, 10.302906, 1)
52 ('./par_sq.out', 180, 8.160446, 2)
53 ('./par_sq.out', 180, 7.060408, 3)
54 ('./par_sq.out', 180, 6.107389, 4)
55 ('./par_sq.out', 240, 18.357986, 1)
56 ('./par_sq.out', 240, 12.030068, 2)
57 ('./par_sq.out', 240, 11.911544, 3)
58 ('./par_sq.out', 240, 9.87963, 4)
59 ('./seq_sq.out', 120, 4.523169, 1)
60 ('./seq_sq.out', 180, 10.363997, 1)
61 ('./seq_sq.out', 240, 18.106595, 1)
62 ('./seq_nlg.out', 120, 2.01212, 1)
63 ('./seq_nlg.out', 180, 3.83345, 1)
64 ('./seq_nlg.out', 240, 5.894773, 1)

```

## B Common Listings for all applications

```

1 #include "gravn.h"
2 #include <sys/time.h>
3 #include <stdio.h>
4 #include <pthread.h>
5
6 void row_of_twenty(body* o, int64_t i) {
7     /* Simple initializer for the bodies where the bodies are
8        stacked up in
9        * columns containing 20 elements each */
10    o->id = i;
11    o->position.x = i / 20;
12    o->position.y = i % 20;
13    o->mass = 100000;
14    o->force.x = 0;
15    o->force.y = 0;
16 }
17
18 struct timeval start_timer() {
19     /* Return the current time. Wrapper for gettimeofday */
20     struct timeval time;
21     gettimeofday(&time, NULL);
22     return time;
23 }

```

```

24 void stop_timer(struct timeval start_time) {
25     /* Calculate the difference between start_time and current
        time and print
26     * it to stdout */
27     struct timeval stop_time;
28     gettimeofday(&stop_time, NULL);
29
30     int64_t seconds_total = stop_time.tv_sec - start_time.
        tv_sec;
31     int64_t microseconds_total = stop_time.tv_usec - start_time
        .tv_usec;
32     if (microseconds_total < 0) {
33         seconds_total--;
34         microseconds_total = 1000000 + microseconds_total;
35     }
36
37     printf("[simulation_time] %ld.%06ld seconds\n",
        seconds_total, microseconds_total);
38 }
39
40 void apply_deltav(body* o) {
41     /* Move the bodies according to deltav and adjust their
        velocity property */
42     point deltav, deltap;
43     deltav.x = o->force.x/o->mass * DELTA_T;
44     deltav.y = o->force.y/o->mass * DELTA_T;
45
46     deltav.x = (o->velocity.x + deltav.x/2) * DELTA_T;
47     deltav.y = (o->velocity.y + deltav.y/2) * DELTA_T;
48
49     o->velocity.x = o->velocity.x + deltav.x;
50     o->velocity.y = o->velocity.y + deltav.y;
51     o->position.x = o->position.x + deltav.x;
52     o->position.y = o->position.y + deltav.y;
53     o->force.x = 0;
54     o->force.y = 0;
55 }
56
57 int64_t num_arrived = 0;
58 pthread_mutex_t barrier_mutex = PTHREAD_MUTEX_INITIALIZER;
59 pthread_cond_t go = PTHREAD_COND_INITIALIZER;
60
61 /*
62  * from the matrix sum code from homework 1
63  */
64 void barrier(int64_t total_workers) {
65     pthread_mutex_lock(&barrier_mutex);
66     num_arrived++;
67     if (num_arrived == total_workers) {

```

```
68     num_arrived = 0;
69     pthread_cond_broadcast(&go);
70 } else {
71     pthread_cond_wait(&go, &barrier_mutex);
72 }
73 pthread_mutex_unlock(&barrier_mutex);
74 }
75
76 double max(double a, double b) {
77     if (a > b)
78         return a;
79     else
80         return b;
81 }
82
83 double min(double a, double b) {
84     if (a < b)
85         return a;
86     else
87         return b;
88 }
```

## C Listings for section 2.1

```
1 #include "gravn.h"
2 #include <math.h>
3 #include <stdio.h>
4 #include <string.h>
5 #include <stdbool.h>
6 #include <stdlib.h>
7 #include <sys/time.h>
8
9 void calculate_forces(int64_t count, body* vec) {
10     /* Calculate the graviational forces between the bodies in
       the 'verse */
11     for (int64_t i = 0; i < count; i++) {
12         for (int64_t j = i + 1; j < count; j++) {
13             double distance = sqrt(pow(vec[i].position.x - vec[j].
               position.x, 2) +
14               pow(vec[i].position.y - vec[j].position.y, 2));
15             double magnitude = (NEWTON_G*vec[i].mass*vec[j].mass) /
16               (pow(distance, 2));
17             point direction;
18             direction.x = vec[j].position.x - vec[i].position.x;
19             direction.y = vec[j].position.y - vec[i].position.y;
20
21             vec[i].force.x = vec[i].force.x + magnitude*direction.x
               /distance;
22             vec[j].force.x = vec[j].force.x - magnitude*direction.x
               /distance;
23             vec[i].force.y = vec[i].force.y + magnitude*direction.y
               /distance;
24             vec[j].force.y = vec[j].force.y - magnitude*direction.y
               /distance;
25         }
26     }
27 }
28
29 void move_bodies(int64_t count, body* vec) {
30     /* Apply the forces of the bodies using the common
       apply_deltav function */
31     for (int64_t i = 0; i < count; i++){
32         apply_deltav(&vec[i]);
33     }
34 }
35
36 int main (int argc, char* argv[]) {
37     /* Run the simulation */
38     int time_limit = TIME_DEFAULT;
39     int n_bodies = BODIES_DEFAULT;
40 }
```

```

41  /* Command line arguments */
42  if (argc > 1) {
43      n_bodies = atoi(argv[1]);
44  }
45  if (argc > 2) {
46      time_limit = atoi(argv[2]);
47  }
48
49  /* Initialize the bodies at their first position */
50  body bodies[n_bodies];
51  memset(bodies, 0, sizeof(body) * n_bodies);
52  for (int i = 0; i < n_bodies; i++) {
53      row_of_twenty(&bodies[i], i);
54  }
55
56  printf("[simulation] %d bodies over %d time steps\n",
57         n_bodies, time_limit);
58  struct timeval start = start_timer();
59  #ifdef DEBUG_MODE
60      FILE* output = fopen("output", "w");
61  #endif
62  /* Do simulation */
63  for (int64_t t = 0; t < time_limit; t++) {
64      calculate_forces(n_bodies, bodies);
65      move_bodies(n_bodies, bodies);
66  #ifdef DEBUG_MODE
67      /* Avoid I/O unless debug-mode is activated */
68      for (int64_t i = 0; i < n_bodies; i++) {
69          fprintf(output, "%ld %ld %lf %lf\n", t, i, bodies[i].
70                  position.x,
71                  bodies[i].position.y);
72      }
73  #endif
74  #ifdef DEBUG_MODE
75      fclose(output);
76  #endif
77  stop_timer(start);
78  }

```



## D Listings for section 2.2

```
1 #include "gravn.h"
2 #include <math.h>
3 #include <stdio.h>
4 #include <stdlib.h>
5 #include <string.h>
6 #include <pthread.h>
7
8 typedef struct worker_info {
9     int64_t worker_id;
10    int64_t total_workers;
11    int64_t count;
12    int64_t time_limit;
13    FILE* output;
14    body* bodies;
15    point** forces;
16 } worker_info;
17
18 void* Worker (void* d);
19
20 void calculate_forces(worker_info data) {
21     /* Calculate how the forces apply to the different bodies
22        */
23     int64_t worker_id, total_workers, count;
24     worker_id = data.worker_id;
25     total_workers = data.total_workers;
26     count = data.count;
27     body* vec = data.bodies;
28     point* force = data.forces[worker_id];
29     /* Decide what bodies this worker applies to. Pattern
30        ABCABCABC is mostly
31        * fair and simple and good 'nuf */
32     for (int64_t i = worker_id; i < count; i+=total_workers) {
33         for (int64_t j = i + 1; j < count; j++) {
34             double distance = sqrt(pow(vec[i].position.x - vec[j].
35                 position.x, 2) +
36                 pow(vec[i].position.y - vec[j].position.y, 2));
37             double magnitude = (NEWTON_G*vec[i].mass*vec[j].mass) /
38                 (pow(distance, 2));
39             point direction;
40             direction.x = vec[j].position.x - vec[i].position.x;
41             direction.y = vec[j].position.y - vec[i].position.y;
42
43             force[i].x = force[i].x + magnitude*direction.x/
44                 distance;
45             force[j].x = force[j].x - magnitude*direction.x/
46                 distance;
47             force[i].y = force[i].y + magnitude*direction.y/
```

```

43         distance;
44         force[j].y = force[j].y - magnitude*direction.y/
45         distance;
46     }
47 }
48 void move_bodies(worker_info data) {
49     /* Apply the forces calculated in calculate_forces on the
50     bodies */
51     point force;
52     force.x = 0.0;
53     force.y = 0.0;
54     for (int64_t i = data.worker_id; i < data.count; i+=data.
55         total_workers) {
56         for (int64_t k = 0; k < data.total_workers; k++) {
57             force.x = force.x + data.forces[k][i].x;
58             force.y = force.y + data.forces[k][i].y;
59             data.forces[k][i].x = 0;
60             data.forces[k][i].y = 0;
61         }
62         data.bodies[i].force.x = force.x;
63         data.bodies[i].force.y = force.y;
64         apply_deltav(&data.bodies[i]);
65         force.x = 0;
66         force.y = 0;
67     }
68 }
69 int main(int argc, char* argv[]) {
70     int time_limit = TIME_DEFAULT;
71     int n_bodies = BODIES_DEFAULT;
72     int n_workers = WORKERS_DEFAULT;
73
74     /* Get command line arguments */
75     if (argc > 1) {
76         n_bodies = atoi(argv[1]);
77     }
78     if (argc > 2) {
79         time_limit = atoi(argv[2]);
80     }
81     if (argc > 3) {
82         n_workers = atoi(argv[3]);
83         if (n_workers > 64) {
84             n_workers = 64;
85         }
86     }
87 }

```

```

88  /* The thread variables and properties for the workers */
89  pthread_t worker_threads[n_workers];
90  worker_info workers_data[n_workers];
91
92  /* set global thread attributes */
93  pthread_attr_t attr;
94  pthread_attr_init(&attr);
95  pthread_attr_setscope(&attr, PTHREAD_SCOPE_SYSTEM);
96
97  FILE* output = fopen("output", "w");
98
99  body* bodies = malloc(sizeof(body) * n_bodies);
100  memset(bodies, 0, sizeof(body) * n_bodies);
101  for (int i = 0; i < n_bodies; i++) {
102      row_of_twenty(&bodies[i], i);
103  }
104
105  /* Allocate space for the pointers to the worker point-
      arrays */
106  point** forces = malloc(sizeof(point*) * n_workers);
107  printf("[simulation] %d bodies over %d time steps with %d\n",
      workers_data[n_bodies, time_limit, n_workers]);
108
109  struct timeval start = start_timer();
110  for (int w = 0; w < n_workers; w++) {
111      /* Iterate over all the workers and create their
          properties */
112      forces[w] = malloc(sizeof(point) * n_bodies);
113      memset(forces[w], 0, sizeof(point) * n_bodies);
114      workers_data[w].worker_id = w;
115      workers_data[w].total_workers = n_workers;
116      workers_data[w].forces = forces;
117      workers_data[w].count = n_bodies;
118      workers_data[w].bodies = bodies;
119      workers_data[w].output = output;
120      workers_data[w].time_limit = time_limit;
121  }
122  for (int w = 0; w < n_workers; w++) {
123      pthread_create(&worker_threads[w], &attr, Worker, (void*)
124          &workers_data[w]);
125  }
126  for (int w = 0; w < n_workers; w++) {
127      pthread_join(worker_threads[w], NULL);
128  }
129  stop_timer(start);
130 }
131
132 void* Worker (void* d) {

```

```

133     /* A worker is a thread which iterates over a predefined
        set of bodies
134     * and calculates their new properties */
135     worker_info* data = (worker_info*) d;
136     for (int64_t t = 0; t < data->time_limit; t++) {
137 #ifdef DEBUG_MODE
138         for (int64_t i = data->worker_id; i < data->count; i+=
            data->total_workers) {
139             fprintf(data->output, "%ld %ld %lf %lf\n", t, i,
140                 data->bodies[i].position.x,
141                 data->bodies[i].position.y);
142         };
143 #endif
144         calculate_forces(*data);
145         barrier(data->total_workers);
146         move_bodies(*data);
147         barrier(data->total_workers);
148     }
149     return NULL;
150 }

```

## E Listings for section 2.3

```

1  #include "gravn.h"
2  #include <stdio.h>
3  #include <stdlib.h>
4  #include <string.h>
5  #include <stdbool.h>
6  #include <math.h>
7
8  #define CUTOFF_DISTANCE_DEFAULT 2
9  #define QUADS_MAX_ELEMENTS 5
10
11 typedef struct body_list {
12     int64_t cnt;
13     body** list;
14 } body_list;
15
16 typedef struct quads {
17     int64_t id;
18     struct quads* children[4];
19     body** bodies;
20     int64_t child_count;
21     double sum_mass;
22     point center_of_mass;
23     /* Corners for the square */
24     point nw;
25     point se;

```

```

26 } quads;
27
28 void divide(int64_t, quads*);
29
30 double point_distance(point a, point b) {
31     double rv = sqrt(pow(a.x - b.x, 2) + pow(a.y - b.y, 2));
32     return rv;
33 }
34
35 point point_direction(point a, point b) {
36     point direction;
37     direction.x = b.x - a.x;
38     direction.y = b.y - a.y;
39     return direction;
40 }
41
42 double point_magnitude(double mass_a, double mass_b, double
    distance) {
43     return ((NEWTON_G*mass_a*mass_b) / pow(distance, 2));
44 }
45
46 void clean_tree(quads* root, int64_t level) {
47     if (level == 0) {
48         for (int i = 0; i < 4; i++) {
49             clean_tree(root->children[i], level+1);
50         }
51     }
52     else if (root->child_count > QUADS_MAX_ELEMENTS) {
53         for (int i = 0; i < 4; i++) {
54             clean_tree(root->children[i], level+1);
55         }
56         free(root->bodies);
57         free(root);
58     } else {
59         free(root->bodies);
60         free(root);
61     }
62 }
63
64 double distance_to_quad(point* origin, quads* target) {
65     /* Calculates the distance to the closes point at the quad
        target from the
        * point origin */
66
67     // FIXME Every point should be within one of the quads
68     double deltax = 0, deltay = 0;
69
70
71     if (origin->x > target->se.x) {
72         deltax = origin->x - target->se.x;

```

```

73     } else if (origin->x < target->nw.x) {
74         deltax = target->nw.x - origin->x;
75     }
76
77     if (origin->y < target->se.y) {
78         deltax = target->se.y - origin->y;
79     } else if (origin->y > target->nw.y) {
80         deltax = origin->y - target->nw.y;
81     }
82
83     if (deltax == 0 && deltax == 0) {
84         return 0.0; // The point is within the quad
85     } else {
86         return sqrt(pow(deltax, 2) + pow(deltax, 2)); //
            Pythagoras
87     }
88 }
89
90 int64_t relevant_forces(body* vec, double cutoff_distance,
    quads* root) {
91     point origin = vec->position;
92     int64_t counter = 0;
93     for (int i = 0; i < 4; i++) {
94         double distance = distance_to_quad(&origin, root->
            children[i]);
95         if (distance > cutoff_distance && root->children[i]->
            child_count) {
96             quads* target = root->children[i];
97             double pdistance = point_distance(vec->position, target
                ->center_of_mass);
98             double magnitude = point_magnitude(vec->mass, target->
                sum_mass, pdistance);
99             point direction = point_direction(vec->position, target
                ->center_of_mass);
100             vec->force.x = vec->force.x + magnitude*direction.x/
                pdistance;
101             vec->force.y = vec->force.y + magnitude*direction.y/
                pdistance;
102             counter++;
103         } else if (root->children[i]->child_count >
            QUADS_MAX_ELEMENTS) {
104             counter += relevant_forces(vec, cutoff_distance, root->
                children[i]);
105         } else if (root->children[i]->child_count == 0) {
106             continue;
107         } else {
108             for (int j = 0; j < root->children[i]->child_count; j
                ++){
109                 body* target = root->children[i]->bodies[j];

```

```

110         double pdistance = point_distance(vec->position,
111                                           target->position);
112         if (pdistance == 0) {
113             continue;
114         }
115         double magnitude = point_magnitude(vec->mass, target
116                                           ->mass, pdistance);
117         point direction = point_direction(vec->position,
118                                           target->position);
119         vec->force.x = vec->force.x + magnitude*direction.x/
120         pdistance;
121         vec->force.y = vec->force.y + magnitude*direction.y/
122         pdistance;
123         counter++;
124     }
125 }
126 return counter;
127 }
128
129 void calculate_forces(int64_t count, quads* root, double
130                       cutoff_distance) {
131     int64_t comparisons = 0;
132     for (int64_t i = 0; i < count; i++) {
133         comparisons += relevant_forces(root->bodies[i],
134                                         cutoff_distance, root);
135     }
136 #ifdef DEBUG_MODE
137     int64_t naive_approx = (count*count)/2;
138     printf("%ld/%ld comparisons, %lf%% saved\n", comparisons,
139           naive_approx,
140           100*(1-((1.0*comparisons)/naive_approx)));
141 #endif
142 }
143
144 void move_bodies(int64_t count, quads* root) {
145     /* Apply the forces of the bodies using the common
146        apply_deltav function */
147     body** vec = root->bodies;
148     for (int64_t i = 0; i < count; i++){
149         apply_deltav(vec[i]);
150     }
151 }
152
153 void insert_body(quads* quad, body* o) {
154     quad->bodies[quad->child_count] = o;
155     quad->child_count++;
156
157     quad->nw.x = min(quad->nw.x, o->position.x);

```

```

150     quad->nw.y = max(quad->nw.y, o->position.y);
151     quad->se.x = max(quad->se.x, o->position.x);
152     quad->se.y = min(quad->se.y, o->position.y);
153 }
154
155 void inner_divide(quads* quad) {
156     if (quad->child_count > QUADS_MAX_ELEMENTS) {
157         divide(quad->child_count, quad);
158
159         point mass_position_sum;
160         mass_position_sum.x = 0;
161         mass_position_sum.y = 0;
162
163         for (int i = 0; i < 4; i++) {
164             mass_position_sum.x += quad->children[i]->
                center_of_mass.x
165             * quad->children[i]->sum_mass;
166             mass_position_sum.y += quad->children[i]->
                center_of_mass.y
167             * quad->children[i]->sum_mass;
168         }
169
170         if (quad->sum_mass != 0) {
171             quad->center_of_mass.x = mass_position_sum.x / quad->
                sum_mass;
172             quad->center_of_mass.y = mass_position_sum.y / quad->
                sum_mass;
173         }
174     } else {
175         point mass_position_sum;
176         mass_position_sum.x = 0;
177         mass_position_sum.y = 0;
178
179         for (int child = 0; child < quad->child_count; child++) {
180             mass_position_sum.x += quad->bodies[child]->position.x
                *
181             quad->bodies[child]->mass;
182             mass_position_sum.y += quad->bodies[child]->position.y
                *
183             quad->bodies[child]->mass;
184             quad->sum_mass += quad->bodies[child]->mass;
185         }
186
187         if (quad->sum_mass != 0) {
188             quad->center_of_mass.x = mass_position_sum.x / quad->
                sum_mass;
189             quad->center_of_mass.y = mass_position_sum.y / quad->
                sum_mass;
190         }

```



```

191     }
192 }
193
194 quads* init_child(int id, point middle, int64_t parent_count)
195 {
196     quads* child = malloc(sizeof(quads));
197     child->id = id;
198     child->child_count = 0;
199     child->sum_mass = 0;
200     child->center_of_mass.x = 0;
201     child->center_of_mass.y = 0;
202     child->bodies = malloc(sizeof(body*)*parent_count);
203     child->se.x = middle.x;
204     child->se.y = middle.y;
205     child->nw.x = middle.x;
206     child->nw.y = middle.y;
207     return child;
208 }
209
210 point get_middle(int64_t count, body** vec) {
211     point middle;
212     middle.x = vec[0]->position.x;
213     middle.y = vec[0]->position.y;
214     for (int64_t i = 1; i < count; i++) {
215         middle.x += vec[i]->position.x;
216         middle.y += vec[i]->position.y;
217     }
218     middle.x = middle.x / count;
219     middle.y = middle.y / count;
220     return middle;
221 }
222
223 void divide(int64_t count, quads* root) {
224     body** vec = root->bodies;
225     point middle = get_middle(count, vec);
226     root->sum_mass = 0;
227     for (int i = 0; i < 4; i++) {
228         root->children[i] = init_child(i, middle, count);
229     }
230
231     for (int64_t i = 0; i < count; i++) {
232         root->sum_mass += vec[i]->mass;
233         if (vec[i]->position.y > middle.y) { // N
234             if (vec[i]->position.x > middle.x) { // NE
235                 insert_body(root->children[0], vec[i]);
236             } else if (vec[i]->position.x <= middle.x) { // NW
237                 insert_body(root->children[1], vec[i]);
238             }
239         } else if (vec[i]->position.y <= middle.y) { // S

```

```

239         if (vec[i]->position.x < middle.x) { // SW
240             insert_body(root->children[2], vec[i]);
241         } else if (vec[i]->position.x >= middle.x){ // SE
242             insert_body(root->children[3], vec[i]);
243         }
244     } else {
245         printf("Error! x %lf y %lf\n", vec[i]->position.x, vec[
            i]->position.y);
246     }
247 }
248 for (int i = 0; i < 4; i++) {
249     inner_divide(root->children[i]);
250 }
251 }
252
253 int main(int argc, char* argv[]) {
254     /* Run the simulation */
255     int time_limit = TIME_DEFAULT;
256     int n_bodies = BODIES_DEFAULT;
257     double cutoff_distance = CUTOFF_DISTANCE_DEFAULT;
258
259     /* Command line arguments */
260     if (argc > 1) {
261         n_bodies = atoi(argv[1]);
262     }
263     if (argc > 2) {
264         time_limit = atoi(argv[2]);
265     }
266     if (argc > 3) {
267         cutoff_distance = atof(argv[3]);
268     }
269
270     /* Initialize the bodies at their first position */
271     point origo;
272     origo.x = 0;
273     origo.y = 0;
274     quads* root = init_child(0, origo, n_bodies);
275     for (int i = 0; i < n_bodies; i++) {
276         root->bodies[i] = malloc(sizeof(body));
277         row_of_twenty(root->bodies[i], i);
278         root->nw.x = min(root->nw.x, root->bodies[i]->position.x)
                ;
279         root->nw.y = max(root->nw.y, root->bodies[i]->position.y)
                ;
280         root->se.x = max(root->se.x, root->bodies[i]->position.x)
                ;
281         root->se.y = min(root->se.y, root->bodies[i]->position.y)
                ;
282     }

```

```

283
284     printf("[simulation] %d bodies over %d time steps -- nlogn\
        n", n_bodies, time_limit);
285     struct timeval start = start_timer();
286 #ifdef DEBUG_MODE
287     FILE* output = fopen("output", "w");
288 #endif
289     /* Do simulation */
290     for (int64_t t = 0; t < time_limit; t++) {
291         divide(n_bodies, root);
292         calculate_forces(n_bodies, root, cutoff_distance);
293         move_bodies(n_bodies, root);
294         clean_tree(root, 0);
295 #ifdef DEBUG_MODE
296         /* Avoid I/O unless debug-mode is activated */
297         for (int64_t i = 0; i < n_bodies; i++) {
298             fprintf(output, "%ld %ld %lf %lf\n", t, i, root->bodies
                [i]->position.x,
299                 root->bodies[i]->position.y);
300         };
301 #endif
302     }
303     stop_timer(start);
304 #ifdef DEBUG_MODE
305     fclose(output);
306 #endif
307
308 }

```

## F Listings for section 2.4

```
1 #include "gravn.h"
2 #include <stdio.h>
3 #include <stdlib.h>
4 #include <string.h>
5 #include <stdbool.h>
6 #include <math.h>
7 #include <pthread.h>
8
9 #define CUTOFF_DISTANCE_DEFAULT 2
10 #define QUADS_MAX_ELEMENTS 5
11
12 typedef struct quads {
13     int64_t id;
14     struct quads* children[4];
15     body** bodies;
16     int64_t child_count;
17     double sum_mass;
18     point center_of_mass;
19     /* Corners for the square */
20     point nw;
21     point se;
22     /* Parallelism */
23     bool done;
24     pthread_mutex_t lock;
25 } quads;
26
27 typedef struct worker_info {
28     int64_t worker_id;
29     int64_t total_workers;
30     int64_t count;
31     int64_t time_limit;
32     double cutoff_distance;
33     FILE* output;
34     quads* root;
35 } worker_info;
36
37 void divide(int64_t, quads*);
38 void* Worker (void*);
39
40 double point_distance(point a, point b) {
41     double rv = sqrt(pow(a.x - b.x, 2) + pow(a.y - b.y, 2));
42     return rv;
43 }
44
45 point point_direction(point a, point b) {
46     point direction;
47     direction.x = b.x - a.x;
```

```

48     direction.y = b.y - a.y;
49     return direction;
50 }
51
52 double point_magnitude(double mass_a, double mass_b, double
    distance) {
53     return ((NEWTON_G*mass_a*mass_b) / pow(distance, 2));
54 }
55
56 void clean_tree(quads* root, int64_t level) {
57     if (level == 0) {
58         for (int i = 0; i < 4; i++) {
59             clean_tree(root->children[i], level+1);
60         }
61     }
62     else if (root->child_count > QUADS_MAX_ELEMENTS) {
63         for (int i = 0; i < 4; i++) {
64             clean_tree(root->children[i], level+1);
65         }
66         free(root->bodies);
67         pthread_mutex_destroy(&root->lock);
68         free(root);
69     } else {
70         free(root->bodies);
71         pthread_mutex_destroy(&root->lock);
72         free(root);
73     }
74 }
75
76 double distance_to_quad(point* origin, quads* target) {
77     /* Calculates the distance to the closes point at the quad
78     target from the
79     * point origin */
80
81     // FIXME Every point should be within one of the quads
82     double deltax = 0, deltax = 0;
83
84     if (origin->x > target->se.x) {
85         deltax = origin->x - target->se.x;
86     } else if (origin->x < target->nw.x) {
87         deltax = target->nw.x - origin->x;
88     }
89
90     if (origin->y < target->se.y) {
91         deltax = target->se.y - origin->y;
92     } else if (origin->y > target->nw.y) {
93         deltax = origin->y - target->nw.y;
94     }

```

```

95     if (deltay == 0 && deltax == 0) {
96         return 0.0; // The point is within the quad
97     } else {
98         return sqrt(pow(deltay, 2) + pow(deltax, 2)); //
           Pythagoras
99     }
100 }
101
102 int64_t relevant_forces(body* vec, double cutoff_distance,
           quads* root) {
103     point origin = vec->position;
104     int64_t counter = 0;
105     for (int i = 0; i < 4; i++) {
106         double distance = distance_to_quad(&origin, root->
           children[i]);
107         if (distance > cutoff_distance && root->children[i]->
           child_count) {
108             quads* target = root->children[i];
109             double pdistance = point_distance(vec->position, target
           ->center_of_mass);
110             double magnitude = point_magnitude(vec->mass, target->
           sum_mass, pdistance);
111             point direction = point_direction(vec->position, target
           ->center_of_mass);
112             vec->force.x = vec->force.x + magnitude*direction.x/
           pdistance;
113             vec->force.y = vec->force.y + magnitude*direction.y/
           pdistance;
114             counter++;
115         } else if (root->children[i]->child_count >
           QUADS_MAX_ELEMENTS) {
116             counter += relevant_forces(vec, cutoff_distance, root->
           children[i]);
117         } else if (root->children[i]->child_count == 0) {
118             continue;
119         } else {
120             for (int j = 0; j < root->children[i]->child_count; j
           ++){
121                 body* target = root->children[i]->bodies[j];
122                 double pdistance = point_distance(vec->position,
           target->position);
123                 if (pdistance == 0) {
124                     continue;
125                 }
126                 double magnitude = point_magnitude(vec->mass, target
           ->mass, pdistance);
127                 point direction = point_direction(vec->position,
           target->position);
128                 vec->force.x = vec->force.x + magnitude*direction.x/

```

```

129         pdistance;
        vec->force.y = vec->force.y + magnitude*direction.y/
        pdistance;
130     counter++;
131 }
132 }
133 }
134 return counter;
135 }
136
137 void calculate_forces(worker_info* data) {
138     int64_t count = data->count;
139     quads* root = data->root;
140     double cutoff_distance = data->cutoff_distance;
141     int64_t comparisons = 0;
142     for (int64_t i = data->worker_id; i < count; i+=data->
        total_workers) {
143         comparisons += relevant_forces(root->bodies[i],
            cutoff_distance, root);
144     }
145 #ifdef DEBUG_MODE
146     int64_t naive_approx = (count*count)/2;
147     printf("%ld/%ld comparisons, %lf%% saved\n", comparisons,
        naive_approx,
148         100*(1-((1.0*comparisons)/naive_approx)));
149 #endif
150 }
151
152 void move_bodies(worker_info* data) {
153     /* Apply the forces of the bodies using the common
        apply_deltav function */
154     body** vec = data->root->bodies;
155     for (int64_t i = data->worker_id; i < data->count; i+=data
        ->total_workers){
156         apply_deltav(vec[i]);
157     }
158 }
159
160 void insert_body(quads* quad, body* o) {
161     quad->bodies[quad->child_count] = o;
162     quad->child_count++;
163
164     quad->nw.x = min(quad->nw.x, o->position.x);
165     quad->nw.y = max(quad->nw.y, o->position.y);
166     quad->se.x = max(quad->se.x, o->position.x);
167     quad->se.y = min(quad->se.y, o->position.y);
168 }
169
170 void inner_divide(quads* quad) {

```

```

171     if (quad->child_count > QUADS_MAX_ELEMENTS) {
172         divide(quad->child_count, quad);
173
174         point mass_position_sum;
175         mass_position_sum.x = 0;
176         mass_position_sum.y = 0;
177
178         for (int i = 0; i < 4; i++) {
179             mass_position_sum.x += quad->children[i]->
                center_of_mass.x
180             * quad->children[i]->sum_mass;
181             mass_position_sum.y += quad->children[i]->
                center_of_mass.y
182             * quad->children[i]->sum_mass;
183         }
184
185         if (quad->sum_mass != 0) {
186             quad->center_of_mass.x = mass_position_sum.x / quad->
                sum_mass;
187             quad->center_of_mass.y = mass_position_sum.y / quad->
                sum_mass;
188         }
189     } else {
190         point mass_position_sum;
191         mass_position_sum.x = 0;
192         mass_position_sum.y = 0;
193
194         for (int child = 0; child < quad->child_count; child++) {
195             mass_position_sum.x += quad->bodies[child]->position.x
                *
196             quad->bodies[child]->mass;
197             mass_position_sum.y += quad->bodies[child]->position.y
                *
198             quad->bodies[child]->mass;
199             quad->sum_mass += quad->bodies[child]->mass;
200         }
201
202         if (quad->sum_mass != 0) {
203             quad->center_of_mass.x = mass_position_sum.x / quad->
                sum_mass;
204             quad->center_of_mass.y = mass_position_sum.y / quad->
                sum_mass;
205         }
206     }
207 }
208
209 quads* init_child(int id, point middle, int64_t parent_count)
210 {
    quads* child = malloc(sizeof(quads));

```



```

211     child->id = id;
212     child->child_count = 0;
213     child->sum_mass = 0;
214     child->center_of_mass.x = 0;
215     child->center_of_mass.y = 0;
216     child->bodies = malloc(sizeof(body*)*parent_count);
217     child->se.x = middle.x;
218     child->se.y = middle.y;
219     child->nw.x = middle.x;
220     child->nw.y = middle.y;
221     child->done = false;
222     pthread_mutex_init(&child->lock, NULL);
223     return child;
224 }
225
226 point get_middle(int64_t count, body** vec) {
227     point middle;
228     middle.x = vec[0]->position.x;
229     middle.y = vec[0]->position.y;
230     for (int64_t i = 1; i < count; i++) {
231         middle.x += vec[i]->position.x;
232         middle.y += vec[i]->position.y;
233     }
234     middle.x = middle.x / count;
235     middle.y = middle.y / count;
236     return middle;
237 }
238
239 void divide(int64_t count, quads* root) {
240     body** vec = root->bodies;
241     point middle = get_middle(count, vec);
242     root->sum_mass = 0;
243     for (int i = 0; i < 4; i++) {
244         root->children[i] = init_child(i, middle, count);
245     }
246
247     for (int64_t i = 0; i < count; i++) {
248         root->sum_mass += vec[i]->mass;
249         if (vec[i]->position.y > middle.y) { // N
250             if (vec[i]->position.x > middle.x) { // NE
251                 insert_body(root->children[0], vec[i]);
252             } else if (vec[i]->position.x <= middle.x) { // NW
253                 insert_body(root->children[1], vec[i]);
254             }
255         } else if (vec[i]->position.y <= middle.y) { // S
256             if (vec[i]->position.x < middle.x) { // SW
257                 insert_body(root->children[2], vec[i]);
258             } else if (vec[i]->position.x >= middle.x) { // SE
259                 insert_body(root->children[3], vec[i]);

```

```

260     }
261 } else {
262     printf("Error! x %lf y %lf\n", vec[i]->position.x, vec[
        i]->position.y);
263 }
264 }
265 for (int i = 0; i < 4; i++) {
266     inner_divide(root->children[i]);
267 }
268 }
269
270 int main(int argc, char* argv[]) {
271     /* Run the simulation */
272     int time_limit = TIME_DEFAULT;
273     int n_bodies = BODIES_DEFAULT;
274     double cutoff_distance = CUTOFF_DISTANCE_DEFAULT;
275     int n_workers = WORKERS_DEFAULT;
276
277     /* Command line arguments */
278     if (argc > 1) {
279         n_bodies = atoi(argv[1]);
280     }
281     if (argc > 2) {
282         time_limit = atoi(argv[2]);
283     }
284     if (argc > 3) {
285         n_workers = atoi(argv[3]);
286         if (n_workers > 64) {
287             n_workers = 64;
288         }
289     }
290     if (argc > 4) {
291         cutoff_distance = atof(argv[4]);
292     }
293
294     /* The thread variables and properties for the workers */
295     pthread_t worker_threads[n_workers];
296     worker_info workers_data[n_workers];
297
298     /* set global thread attributes */
299     pthread_attr_t attr;
300     pthread_attr_init(&attr);
301     pthread_attr_setscope(&attr, PTHREAD_SCOPE_SYSTEM);
302
303
304     /* Initialize the bodies at their first position */
305
306     point origo;
307     origo.x = 0;

```

```

308     origo.y = 0;
309     quads* root = init_child(0, origo, n_bodies);
310 #ifdef DEBUG_MODE
311     FILE* output = fopen("output", "w");
312 #endif
313
314     printf("[simulation] %d bodies over %d time steps with %d
        workers -- nlogn\n",
315           n_bodies, time_limit, n_workers);
316     struct timeval start = start_timer();
317     for (int w = 0; w < n_workers; w++) {
318         /* Iterate over all the workers and create their
        properties */
319         workers_data[w].worker_id = w;
320         workers_data[w].total_workers = n_workers;
321         workers_data[w].count = n_bodies;
322         workers_data[w].root = root;
323 #ifdef DEBUG_MODE
324         workers_data[w].output = output;
325 #endif
326         workers_data[w].time_limit = time_limit;
327         workers_data[w].cutoff_distance = cutoff_distance;
328     }
329     for (int w = 0; w < n_workers; w++) {
330         pthread_create(&worker_threads[w], &attr, Worker, (void*)
            &workers_data[w]);
331     }
332     for (int w = 0; w < n_workers; w++) {
333         pthread_join(worker_threads[w], NULL);
334     }
335     stop_timer(start);
336 #ifdef DEBUG_MODE
337     fclose(output);
338 #endif
339
340 }
341
342 void* Worker (void* in) {
343     worker_info* data = (worker_info*) in;
344     quads* root = data->root;
345
346     for (int i = data->worker_id; i < data->count; i+=data->
        total_workers) {
347         root->bodies[i] = malloc(sizeof(body));
348         row_of_twenty(root->bodies[i], i);
349         root->nw.x = min(root->nw.x, root->bodies[i]->position.x)
            ;
350         root->nw.y = max(root->nw.y, root->bodies[i]->position.y)
            ;

```

```

351     root->se.x = max(root->se.x, root->bodies[i]->position.x)
352     ;
353     root->se.y = min(root->se.y, root->bodies[i]->position.y)
354     ;
355 }
356 barrier(data->total_workers);
357
358 /* Do simulation */
359 for (int64_t t = 0; t < data->time_limit; t++) {
360     if (data->worker_id == 0)
361         divide(data->count, root);
362     barrier(data->total_workers);
363     calculate_forces(data);
364     barrier(data->total_workers);
365     move_bodies(data);
366     barrier(data->total_workers);
367     if (data->worker_id == 0)
368         clean_tree(root, 0);
369     barrier(data->total_workers);
370 #ifdef DEBUG_MODE
371     /* Avoid I/O unless debug-mode is activated */
372     for (int64_t i = 0; i < data->count; i++) {
373         fprintf(data->output, "%ld %ld %lf %lf\n", t, i, root->
374             bodies[i]->position.x,
375             root->bodies[i]->position.y);
376     };
377 #endif
378 }
379 return NULL;
380 }

```

## G Broken parallellization of 2.4

```
1 | diff --git a/project/par_nlg.c b/project/par_nlg.c
2 | index cfabd39..f5146a6 100644
3 | --- a/project/par_nlg.c
4 | +++ b/project/par_nlg.c
5 | @@ -21,6 +21,7 @@ typedef struct quads {
6 |     point se;
7 |     /* Parallelism */
8 |     bool done;
9 | +   bool initd;
10 |     pthread_mutex_t lock;
11 | } quads;
12 |
13 | @@ -34,7 +35,7 @@ typedef struct worker_info {
14 |     quads* root;
15 | } worker_info;
16 |
17 | -void divide(int64_t, quads*);
18 | +void divide(int64_t, quads*, worker_info*);
19 | void* Worker (void*);
20 |
21 | double point_distance(point a, point b) {
22 | @@ -58,6 +59,8 @@ void clean_tree(quads* root, int64_t level)
23 |     {
24 |         for (int i = 0; i < 4; i++) {
25 |             clean_tree(root->children[i], level+1);
26 |         }
27 | +   root->initd = false;
28 | +   root->done = false;
29 |     }
30 |     else if (root->child_count > QUADS_MAX_ELEMENTS) {
31 |         for (int i = 0; i < 4; i++) {
32 | @@ -77,7 +80,6 @@ double distance_to_quad(point* origin,
33 |         quads* target) {
34 |             /* Calculates the distance to the closes point at the quad
35 |              * target from the
36 |              * point origin */
37 |
38 | -   // FIXME Every point should be within one of the quads
39 |             double deltax = 0, deltax = 0;
40 |
41 |             if (origin->x > target->se.x) {
42 | @@ -158,6 +160,7 @@ void move_bodies(worker_info* data) {
43 |             }
44 |
45 | void insert_body(quads* quad, body* o) {
46 | +   pthread_mutex_lock(&quad->lock);
47 |     quad->bodies[quad->child_count] = o;
```

```

45     quad->child_count++;
46
47 @@ -165,12 +168,14 @@ void insert_body(quads* quad, body* o)
48     {
49     quad->nw.y = max(quad->nw.y, o->position.y);
50     quad->se.x = max(quad->se.x, o->position.x);
51     quad->se.y = min(quad->se.y, o->position.y);
52 +   pthread_mutex_unlock(&quad->lock);
53   }
54
55 -void inner_divide(quads* quad) {
56 +void inner_divide(quads* quad, worker_info* data) {
57     if (quad->child_count > QUADS_MAX_ELEMENTS) {
58     -   divide(quad->child_count, quad);
59     +   divide(quad->child_count, quad, data);
60
61     +   pthread_mutex_lock(&quad->lock);
62     point mass_position_sum;
63     mass_position_sum.x = 0;
64     mass_position_sum.y = 0;
65 @@ -186,7 +191,9 @@ void inner_divide(quads* quad) {
66     quad->center_of_mass.x = mass_position_sum.x / quad->
67         sum_mass;
68     quad->center_of_mass.y = mass_position_sum.y / quad->
69         sum_mass;
70   }
71 +   pthread_mutex_unlock(&quad->lock);
72   } else {
73   +   pthread_mutex_lock(&quad->lock);
74   point mass_position_sum;
75   mass_position_sum.x = 0;
76   mass_position_sum.y = 0;
77 @@ -203,11 +210,14 @@ void inner_divide(quads* quad) {
78     quad->center_of_mass.x = mass_position_sum.x / quad->
79         sum_mass;
80     quad->center_of_mass.y = mass_position_sum.y / quad->
81         sum_mass;
82   }
83 +   pthread_mutex_unlock(&quad->lock);
84   }
85 +   quad->done = true;
86   }
87
88 quads* init_child(int id, point middle, int64_t parent_count
89 ) {
90     quads* child = malloc(sizeof(quads));
91     +   child->initd = false;
92     child->id = id;
93     child->child_count = 0;

```

```

88     child->sum_mass = 0;
89 @@ -236,34 +246,44 @@ point get_middle(int64_t count, body**
    vec) {
90     return middle;
91 }
92
93 -void divide(int64_t count, quads* root) {
94 +void divide(int64_t count, quads* root, worker_info* data) {
95     body** vec = root->bodies;
96     point middle = get_middle(count, vec);
97     root->sum_mass = 0;
98     for (int i = 0; i < 4; i++) {
99         root->children[i] = init_child(i, middle, count);
100     }
101
102     for (int64_t i = 0; i < count; i++) {
103         root->sum_mass += vec[i]->mass;
104         if (vec[i]->position.y > middle.y) { // N
105             if (vec[i]->position.x > middle.x) { // NE
106                 insert_body(root->children[0], vec[i]);
107             } else if (vec[i]->position.x <= middle.x) { // NW
108                 insert_body(root->children[1], vec[i]);
109             }
110             point middle;
111             while (root->initd == false) {
112                 if (pthread_mutex_trylock(&root->lock) == 0) {
113                     if (root->initd == true)
114                         continue;
115                     middle = get_middle(count, vec);
116                     for (int i = 0; i < 4; i++) {
117                         root->children[i] = init_child(i, middle, count);
118                     }
119                 } else if (vec[i]->position.y <= middle.y) { // S
120                     if (vec[i]->position.x < middle.x) { // SW
121                         insert_body(root->children[2], vec[i]);
122                     } else if (vec[i]->position.x >= middle.x) { // SE
123                         insert_body(root->children[3], vec[i]);
124                     }
125                     root->initd = true;
126                 }
127                 for (int64_t i = 0; i < count; i++) {
128                     root->sum_mass += vec[i]->mass;
129                     if (vec[i]->position.y > middle.y) { // N
130                         if (vec[i]->position.x > middle.x) { // NE
131                             insert_body(root->children[0], vec[i]);
132                         } else if (vec[i]->position.x <= middle.x) { // NW
133                             insert_body(root->children[1], vec[i]);
134                         }
135                     } else if (vec[i]->position.y <= middle.y) { // S
136                         if (vec[i]->position.x < middle.x) { // SW
137                             insert_body(root->children[2], vec[i]);
138                         } else if (vec[i]->position.x >= middle.x) { // SE
139                             insert_body(root->children[3], vec[i]);
140                         }
141                     }
142                     root->initd = true;
143                 }
144             }
145         }
146     }
147 }

```

```

136 +         } else if (vec[i]->position.x >= middle.x){ // SE
137 +             insert_body(root->children[3], vec[i]);
138 +         }
139 +     } else {
140 +         printf("Error! x %lf y %lf\n", vec[i]->position.x,
141 +             vec[i]->position.y);
142 +     }
143 -     } else {
144 -         printf("Error! x %lf y %lf\n", vec[i]->position.x, vec
145 -             [i]->position.y);
146 -     }
147 + pthread_mutex_unlock(&root->lock);
148 for (int i = 0; i < 4; i++) {
149 -     inner_divide(root->children[i]);
150 +     if (!root->children[i]->done) {
151 +         inner_divide(root->children[i], data);
152 +     }
153 }
154 }
155
156 @@ -355,8 +375,7 @@ void* Worker (void* in) {
157
158     /* Do simulation */
159     for (int64_t t = 0; t < data->time_limit; t++) {
160 -         if (data->worker_id == 0)
161 -             divide(data->count, root);
162 +         divide(data->count, root, data);
163         barrier(data->total_workers);
164         calculate_forces(data);
165         barrier(data->total_workers);

```



## H Listings for help script plot.py

```
1 from matplotlib import pyplot as plt
2 from matplotlib import animation
3 import json
4
5 data = []
6 t = 0;
7 with open('output') as in_f:
8     for line in in_f:
9         line = line.strip()
10        line = line.split(' ')
11        line[0] = int(line[0])
12        line[2] = float(line[2])
13        line[3] = float(line[3])
14        if line[0] > t:
15            t = line[0]
16        data.append(line)
17
18 figure = plt.figure()
19 plt.xkcd()
20 plt.title('N-body Simulation')
21 frames = []
22 for i in range(0, t, 5):
23     frame = filter(lambda datum: datum[0] == i, data)
24     xs = list(map(lambda f: f[2], frame))
25     ys = list(map(lambda f: f[3], frame))
26     frames.append((plt.scatter(xs, ys),))
27     print("added frame %d" % i)
28
29 gif = animation.ArtistAnimation(figure, frames, interval=50,
30     repeat_delay=3000, blit=True)
31
32 gif.save('nbody.mp4', fps=24, extra_args=['-vcodec', 'libx264
33     '])
34
35 print("Printed animation to 'nbody.mp4'")
36
37 plt.show()
```

## I Listings for help script performance.py

```
1  #!/usr/bin/env python3
2  import subprocess
3
4  processes = ['./par_nlg.out', './par_sq.out', './seq_sq.out',
5              './seq_nlg.out']
6  sizes = ['120', '180', '240']
7  cores = ['1', '2', '3', '4']
8
9  def run_process(run):
10     rv = subprocess.check_output(run).decode('utf-8')
11     rv = rv.strip()
12     rv = rv.split('\n')
13     for i, line in enumerate(rv):
14         rv[i] = line.split(' ')
15     if 'workers' in rv[0]:
16         cpu_count = int(rv[0][8])
17     else:
18         cpu_count = 1
19     bodies = int(rv[0][1])
20     time = float(rv[1][1])
21     return (p, bodies, time, cpu_count)
22
23  for p in processes:
24     runs = []
25     for size in sizes:
26         if 'par' in p:
27             for c in cores:
28                 runs.append([p, size, '50000', c])
29         else:
30             runs.append([p, size, '50000'])
31     if 'nlg' in p:
32         for run in runs:
33             run.append('0.7')
34
35
36     for run in runs:
37         results = []
38         for i in range(5):
39             results.append(run_process(run))
40         for i in range(2):
41             results.remove(min(results, key=lambda rvec: rvec[2]))
42             results.remove(max(results, key=lambda rvec: rvec[2]))
43     print(results[0])
```

## J Listings for Makefile

```
1 CC=gcc
2 CFLAGS=-std=c99 -g -Wall -D_XOPEN_SOURCE=600 -O3 -lm
3
4 all: seq_sq.out seq_sq.debug.out par_sq.out par_sq.debug.out
5     seq_nlg.out \
6         seq_nlg.debug.out par_nlg.out par_nlg.debug.out
7
8 seq_sq.out: seq_sq.c gravn_common.o
9     $(CC) $(CFLAGS) -o $@ $< gravn_common.o
10
11 seq_sq.debug.out: seq_sq.c gravn_common.o
12     $(CC) $(CFLAGS) -DDEBUG_MODE -o $@ $< gravn_common.o
13
14 par_sq.out: par_sq.c gravn_common.o
15     $(CC) $(CFLAGS) -lpthread -o $@ $< gravn_common.o
16
17 par_sq.debug.out: par_sq.c gravn_common.o
18     $(CC) $(CFLAGS) -DDEBUG_MODE -lpthread -o $@ $<
19         gravn_common.o
20
21 seq_nlg.out: seq_nlg.c gravn_common.o
22     $(CC) $(CFLAGS) -o $@ $< gravn_common.o
23
24 seq_nlg.debug.out: seq_nlg.c gravn_common.o
25     $(CC) $(CFLAGS) -DDEBUG_MODE -o $@ $< gravn_common.o
26
27 par_nlg.out: par_nlg.c gravn_common.o
28     $(CC) $(CFLAGS) -lpthread -o $@ $< gravn_common.o
29
30 par_nlg.debug.out: par_nlg.c gravn_common.o
31     $(CC) $(CFLAGS) -DDEBUG_MODE -lpthread -o $@ $<
32         gravn_common.o
33
34 gravn_common.o: gravn_common.c
35     $(CC) $(CFLAGS) -c -o $@ $<
36
37 clean:
38     $(RM) ./*.out
39     $(RM) ./*.o
```