The Gravitational N-body System

Emil Tullstedt

2016-03-25

Contents

1 Introduction **2**

2 Programs **2**

2.1 Naïve Sequential Program 2

2.1.1 Calculating Forces 2

2.1.2 Applying Forces 3

2.2 Naïve Parallell Program 4

2.3 Barnes-Hut Sequential Program 4

2.4 Barnes-Hut Parallell Program 6

3 Evaluation **6**

4 Conclusion **7**

Appendices **10**

A Performance.py Time Results **10**

B Common Listings for all applications **11**

C Listings for section 2.1 **14**

D Listings for section 2.2 **16**

E Listings for section 2.3 **19**

F Listings for section 2.4 **27**

G Broken parallellization of 2.4 **36**

H Listings for help script plot.py **40**

I Listings for help script performance.py **41**

J Listings for Makefile **42**

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 astronomics 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=AHOPfpAryfc` 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 continous 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.

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

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

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

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

$$d(ij)_x = p(j)_x - p(i)_x \tag{3}$$
$$d(ij)_y = p(j)_y - p(j)_y \tag{4}$$

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

$$f(i)_x = f(i)_x + \frac{m(ij) * d(ij)_x}{p(ij)} \tag{5}$$
$$f(i)_y = f(i)_y + \frac{m(ij) * d(ij)_y}{p(ij)} \tag{6}$$

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 calculated 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).

$$f(j)_x = f(i)_x - \frac{m(ij) * d(ij)_x}{p(ij)} \tag{7}$$
$$f(j)_y = f(i)_y - \frac{m(ij) * d(ij)_y}{p(ij)} \tag{8}$$

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.

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

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

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

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

$$v(i)_x = v(i)_x + \delta v(i)_x \tag{13}$$

$$v(i)_y = v(i)_y + \delta v(i)_y \tag{14}$$

$$p(i)_x = p(i)_x + \delta p(i)_x \tag{15}$$

$$p(i)_y = p(i)_y + \delta p(i)_y \tag{16}$$

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

2.2 Naïve Parallell 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 $f(i) = f(i(t_0)) + f(i(t_1)) + \cdots + f(i(t_n))$ 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

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

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 individual particles is used, with the modification that the “equal but opposing force”- optimiziation 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 parallellization 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 parallell 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 parallellization.

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 parallellization of the Barnes-Hut approximation I am fairly sure that results that would've ranged from $O(n^2)t \rightarrow \dfrac{O(n \log n)t}{3}$ could've appeared in table 2 when running on 4 cores rather than the current range from $O(n^2)t \rightarrow \dfrac{O(n \log n)t}{2}$.

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

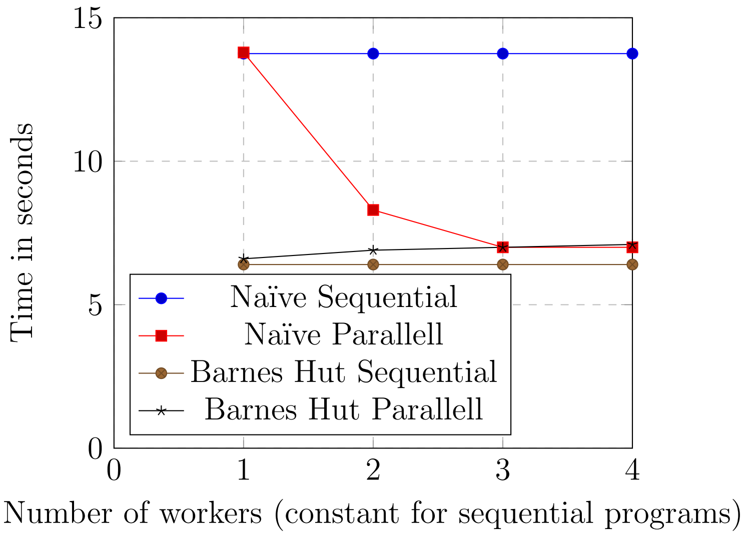

Figure 1: Performance with 120 particles

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

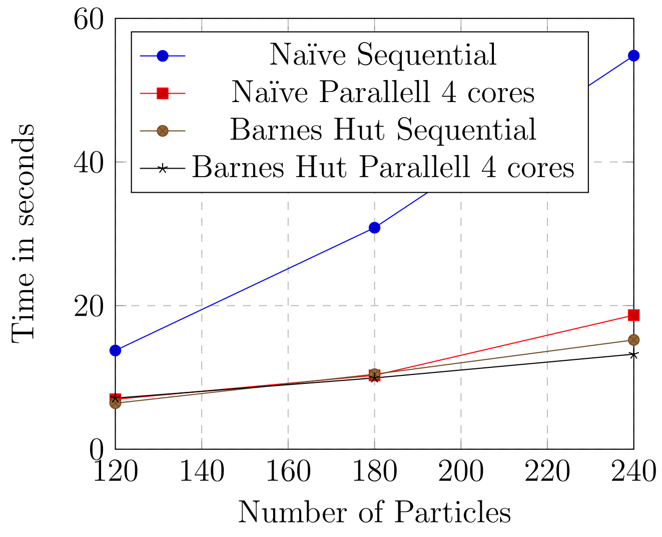

Figure 2: Performance Development with more Particles

Appendices

A Performance.py Time Results

```
<KTH Shell>

('./par_nlg.out', 120, 6.6356520000000003, 1)
('./par_nlg.out', 120, 6.906879, 2)
('./par_nlg.out', 120, 7.0215680000000003, 3)
('./par_nlg.out', 120, 7.1209990000000003, 4)
('./par_nlg.out', 180, 10.765097000000001, 1)
('./par_nlg.out', 180, 10.588206, 2)
('./par_nlg.out', 180, 9.1330360000000006, 3)
('./par_nlg.out', 180, 9.9337389999999992, 4)
('./par_nlg.out', 240, 15.738244, 1)
('./par_nlg.out', 240, 14.88978, 2)
('./par_nlg.out', 240, 13.702691, 3)
('./par_nlg.out', 240, 13.239893, 4)
('./par_sq.out', 120, 13.792294999999999, 1)
('./par_sq.out', 120, 8.3339909999999993, 2)
('./par_sq.out', 120, 7.0577490000000003, 3)
('./par_sq.out', 120, 6.9642720000000002, 4)
('./par_sq.out', 180, 30.919129999999999, 1)
('./par_sq.out', 180, 16.681785999999999, 2)
('./par_sq.out', 180, 12.509850999999999, 3)
('./par_sq.out', 180, 10.286918, 4)
('./par_sq.out', 240, 54.972634999999997, 1)
('./par_sq.out', 240, 29.186340999999999, 2)
('./par_sq.out', 240, 27.993697999999998, 3)
('./par_sq.out', 240, 18.670895000000002, 4)
('./seq_sq.out', 120, 13.759906000000001, 1)
('./seq_sq.out', 180, 30.846284000000001, 1)
('./seq_sq.out', 240, 54.812840000000001, 1)
('./seq_nlg.out', 120, 6.4147290000000003, 1)
('./seq_nlg.out', 180, 10.448035000000001, 1)
('./seq_nlg.out', 240, 15.222167000000001, 1)

<Lenovo X250 w/ dual-core i3 + hyper-threading>
('./par_nlg.out', 120, 2.386097, 1)
('./par_nlg.out', 120, 2.720354, 2)
('./par_nlg.out', 120, 3.174221, 3)
('./par_nlg.out', 120, 3.484154, 4)
('./par_nlg.out', 180, 4.270851, 1)
('./par_nlg.out', 180, 4.490971, 2)
('./par_nlg.out', 180, 5.031048, 3)
('./par_nlg.out', 180, 6.069867, 4)
('./par_nlg.out', 240, 6.802512, 1)
```

```
('./par_nlg.out', 240, 6.301626, 2)
('./par_nlg.out', 240, 8.207197, 3)
('./par_nlg.out', 240, 6.441181, 4)
('./par_sq.out', 120, 4.556379, 1)
('./par_sq.out', 120, 3.33266, 2)
('./par_sq.out', 120, 3.323491, 3)
('./par_sq.out', 120, 3.296448, 4)
('./par_sq.out', 180, 10.302906, 1)
('./par_sq.out', 180, 8.160446, 2)
('./par_sq.out', 180, 7.060408, 3)
('./par_sq.out', 180, 6.107389, 4)
('./par_sq.out', 240, 18.357986, 1)
('./par_sq.out', 240, 12.030068, 2)
('./par_sq.out', 240, 11.911544, 3)
('./par_sq.out', 240, 9.87963, 4)
('./seq_sq.out', 120, 4.523169, 1)
('./seq_sq.out', 180, 10.363997, 1)
('./seq_sq.out', 240, 18.106595, 1)
('./seq_nlg.out', 120, 2.01212, 1)
('./seq_nlg.out', 180, 3.83345, 1)
('./seq_nlg.out', 240, 5.894773, 1)
```

B Common Listings for all applications

```c
#include "gravn.h"
#include <sys/time.h>
#include <stdio.h>
#include <pthread.h>

void row_of_twenty(body* o, int64_t i) {
  /* Simple initializer for the bodies where the bodies are
      stacked up in
   * columns containing 20 elemens each */
  o->id = i;
  o->position.x = i / 20;
  o->position.y = i % 20;
  o->mass = 100000;
  o->force.x = 0;
  o->force.y = 0;
}

struct timeval start_timer() {
  /* Return the current time. Wrapper for gettimeofday */
  struct timeval time;
  gettimeofday(&time, NULL);
  return time;
}

```

```
void stop_timer(struct timeval start_time) {
  /* Calculate the difference between start_time and current time and print
   * it to stdout */
  struct timeval stop_time;
  gettimeofday(&stop_time, NULL);

  int64_t seconds_total = stop_time.tv_sec - start_time.tv_sec;
  int64_t microseconds_total = stop_time.tv_usec - start_time.tv_usec;
  if (microseconds_total < 0) {
    seconds_total--;
    microseconds_total = 1000000 + microseconds_total;
  }

  printf("[simulation_time] %ld.%06ld seconds\n", seconds_total, microseconds_total);
}

void apply_deltav(body* o) {
  /* Move the bodies according to deltav and adjust their velocity property */
  point deltav, deltap;
  deltav.x = o->force.x/o->mass * DELTA_T;
  deltav.y = o->force.y/o->mass * DELTA_T;

  deltap.x = (o->velocity.x + deltav.x/2) * DELTA_T;
  deltap.y = (o->velocity.y + deltav.y/2) * DELTA_T;

  o->velocity.x = o->velocity.x + deltav.x;
  o->velocity.y = o->velocity.y + deltav.y;
  o->position.x = o->position.x + deltap.x;
  o->position.y = o->position.y + deltap.y;
  o->force.x = 0;
  o->force.y = 0;
}

int64_t num_arrived = 0;
pthread_mutex_t barrier_mutex = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t go = PTHREAD_COND_INITIALIZER;

/*
 * from the matrix sum code from homework 1
 */
void barrier(int64_t total_workers) {
  pthread_mutex_lock(&barrier_mutex);
  num_arrived++;
  if (num_arrived == total_workers) {
    num_arrived = 0;
    pthread_cond_broadcast(&go);
  } else {
    pthread_cond_wait(&go, &barrier_mutex);
  }
  pthread_mutex_unlock(&barrier_mutex);
}

double max(double a, double b) {
  if (a > b)
    return a;
  else
    return b;
}

double min(double a, double b) {
  if (a < b)
    return a;
  else
    return b;
}
```

C Listings for section 2.1

```c
#include "gravn.h"
#include <math.h>
#include <stdio.h>
#include <string.h>
#include <stdbool.h>
#include <stdlib.h>
#include <sys/time.h>

void calculate_forces(int64_t count, body* vec) {
  /* Calculate the graviational forces between the bodies in the 'verse */
  for (int64_t i = 0; i < count; i++) {
    for (int64_t j = i + 1; j < count; j++) {
      double distance = sqrt(pow(vec[i].position.x - vec[j].position.x, 2) +
           pow(vec[i].position.y - vec[j].position.y, 2));
      double magnitude = (NEWTON_G*vec[i].mass*vec[j].mass) /
        (pow(distance, 2));
      point direction;
      direction.x = vec[j].position.x - vec[i].position.x;
      direction.y = vec[j].position.y - vec[i].position.y;

      vec[i].force.x = vec[i].force.x + magnitude*direction.x/distance;
      vec[j].force.x = vec[j].force.x - magnitude*direction.x/distance;
      vec[i].force.y = vec[i].force.y + magnitude*direction.y/distance;
      vec[j].force.y = vec[j].force.y - magnitude*direction.y/distance;
    }
  }
}

void move_bodies(int64_t count, body* vec) {
  /* Apply the forces of the bodies using the common apply_deltav function */
  for (int64_t i = 0; i < count; i++){
    apply_deltav(&vec[i]);
  }
}

int main (int argc, char* argv[]) {
  /* Run the simulation */
  int time_limit = TIME_DEFAULT;
  int n_bodies = BODIES_DEFAULT;

  /* Command line arguments */
  if (argc > 1) {
    n_bodies = atoi(argv[1]);
  }
  if (argc > 2) {
    time_limit = atoi(argv[2]);
  }

  /* Initialize the bodies at their first position */
  body bodies[n_bodies];
  memset(bodies, 0, sizeof(body) * n_bodies);
  for (int i = 0; i < n_bodies; i++) {
    row_of_twenty(&bodies[i], i);
  }

  printf("[simulation] %d bodies over %d time steps\n",
      n_bodies, time_limit);
  struct timeval start = start_timer();
#ifdef DEBUG_MODE
  FILE* output = fopen("output", "w");
#endif
  /* Do simulation */
  for (int64_t t = 0; t < time_limit; t++) {
    calculate_forces(n_bodies, bodies);
    move_bodies(n_bodies, bodies);
#ifdef DEBUG_MODE
    /* Avoid I/O unless debug-mode is activated */
    for (int64_t i = 0; i < n_bodies; i++) {
      fprintf(output, "%ld %ld %lf %lf\n", t, i, bodies[i].
          position.x,
        bodies[i].position.y);
    };
#endif
  }
#ifdef DEBUG_MODE
  fclose(output);
#endif
  stop_timer(start);
}
```

D Listings for section 2.2

```c
#include "gravn.h"
#include <math.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <pthread.h>

typedef struct worker_info {
  int64_t worker_id;
  int64_t total_workers;
  int64_t count;
  int64_t time_limit;
  FILE* output;
  body* bodies;
  point** forces;
} worker_info;

void* Worker (void* d);

void calculate_forces(worker_info data) {
  /* Calculate how the forces apply to the different bodies */
  int64_t worker_id, total_workers, count;
  worker_id = data.worker_id;
  total_workers = data.total_workers;
  count = data.count;
  body* vec = data.bodies;
  point* force = data.forces[worker_id];
  /* Decide what bodies this worker applies to. Pattern ABCABCABC is mostly
   * fair and simple and good 'nuf */
  for (int64_t i = worker_id; i < count; i+=total_workers) {
    for (int64_t j = i + 1; j < count; j++) {
      double distance = sqrt(pow(vec[i].position.x - vec[j].position.x, 2) +
          pow(vec[i].position.y - vec[j].position.y, 2));
      double magnitude = (NEWTON_G*vec[i].mass*vec[j].mass) /
        (pow(distance, 2));
      point direction;
      direction.x = vec[j].position.x - vec[i].position.x;
      direction.y = vec[j].position.y - vec[i].position.y;

      force[i].x = force[i].x + magnitude*direction.x/distance;
      force[j].x = force[j].x - magnitude*direction.x/distance;
      force[i].y = force[i].y + magnitude*direction.y/
```

```
          distance;
      force[j].y = force[j].y - magnitude*direction.y/distance;
    }
  }
}

void move_bodies(worker_info data) {
  /* Apply the forces calculated in calculate_forces on the bodies */
  point force;
  force.x = 0.0;
  force.y = 0.0;

  for (int64_t i = data.worker_id; i < data.count; i+=data.total_workers) {
    for (int64_t k = 0; k < data.total_workers; k++) {
      force.x = force.x + data.forces[k][i].x;
      force.y = force.y + data.forces[k][i].y;
      data.forces[k][i].x = 0;
      data.forces[k][i].y = 0;
    }
    data.bodies[i].force.x = force.x;
    data.bodies[i].force.y = force.y;
    apply_deltav(&data.bodies[i]);
    force.x = 0;
    force.y = 0;
  }
}

int main(int argc, char* argv[]) {
  int time_limit = TIME_DEFAULT;
  int n_bodies = BODIES_DEFAULT;
  int n_workers = WORKERS_DEFAULT;

  /* Get command line arguments */
  if (argc > 1) {
    n_bodies = atoi(argv[1]);
  }
  if (argc > 2) {
    time_limit = atoi(argv[2]);
  }
  if (argc > 3) {
    n_workers = atoi(argv[3]);
    if (n_workers > 64) {
      n_workers = 64;
    }
  }

```

```c
  /* The thread variables and properties for the workers */
  pthread_t worker_threads[n_workers];
  worker_info workers_data[n_workers];

  /* set global thread attributes */
  pthread_attr_t attr;
  pthread_attr_init(&attr);
  pthread_attr_setscope(&attr, PTHREAD_SCOPE_SYSTEM);

  FILE* output = fopen("output", "w");

  body* bodies = malloc(sizeof(body) * n_bodies);
  memset(bodies, 0, sizeof(body) * n_bodies);
  for (int i = 0; i < n_bodies; i++) {
    row_of_twenty(&bodies[i], i);
  }

  /* Allocate space for the pointers to the worker point-arrays */
  point** forces = malloc(sizeof(point*) * n_workers);
  printf("[simulation] %d bodies over %d time steps with %d workers\n",
      n_bodies, time_limit, n_workers);

  struct timeval start = start_timer();
  for (int w = 0; w < n_workers; w++) {
    /* Iterate over all the workers and create their properties */
    forces[w] = malloc(sizeof(point) * n_bodies);
    memset(forces[w], 0, sizeof(point) * n_bodies);
    workers_data[w].worker_id = w;
    workers_data[w].total_workers = n_workers;
    workers_data[w].forces = forces;
    workers_data[w].count = n_bodies;
    workers_data[w].bodies = bodies;
    workers_data[w].output = output;
    workers_data[w].time_limit = time_limit;
  }
  for (int w = 0; w < n_workers; w++) {
    pthread_create(&worker_threads[w], &attr, Worker, (void*) &workers_data[w]);
  }
  for (int w = 0; w < n_workers; w++) {
    pthread_join(worker_threads[w], NULL);
  }
  stop_timer(start);
}

void* Worker (void* d) {
  /* A worker is a thread which iterates over a predefined
      set of bodies
   * and calculates their new properties */
  worker_info* data = (worker_info*) d;
  for (int64_t t = 0; t < data->time_limit; t++) {
#ifdef DEBUG_MODE
    for (int64_t i = data->worker_id; i < data->count; i+=
        data->total_workers) {
      fprintf(data->output, "%ld %ld %lf %lf\n", t, i,
          data->bodies[i].position.x,
          data->bodies[i].position.y);
    };
#endif
    calculate_forces(*data);
    barrier(data->total_workers);
    move_bodies(*data);
    barrier(data->total_workers);
  }
  return NULL;
}
```

E Listings for section 2.3

```c
#include "gravn.h"
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <stdbool.h>
#include <math.h>

#define CUTOFF_DISTANCE_DEFAULT 2
#define QUADS_MAX_ELEMENTS 5

typedef struct body_list {
  int64_t cnt;
  body** list;
} body_list;

typedef struct quads {
  int64_t id;
  struct quads* children[4];
  body** bodies;
  int64_t child_count;
  double sum_mass;
  point center_of_mass;
  /* Corners for the square */
  point nw;
  point se;
```

```
} quads;

void divide(int64_t, quads*);

double point_distance(point a, point b) {
  double rv = sqrt(pow(a.x - b.x, 2) + pow(a.y - b.y, 2));
  return rv;
}

point point_direction(point a, point b) {
  point direction;
  direction.x = b.x - a.x;
  direction.y = b.y - a.y;
  return direction;
}

double point_magnitude(double mass_a, double mass_b, double
    distance) {
  return ((NEWTON_G*mass_a*mass_b) / pow(distance, 2));
}

void clean_tree(quads* root, int64_t level) {
  if (level == 0) {
    for (int i = 0; i < 4; i++) {
      clean_tree(root->children[i], level+1);
    }
  }
  else if (root->child_count > QUADS_MAX_ELEMENTS) {
    for (int i = 0; i < 4; i++) {
      clean_tree(root->children[i], level+1);
    }
    free(root->bodies);
    free(root);
  } else {
    free(root->bodies);
    free(root);
  }
}

double distance_to_quad(point* origin, quads* target) {
  /* Calculates the distance to the closes point at the quad
      target from the
   * point origin */

  // FIXME Every point should be within one of the quads
  double deltax = 0, deltay = 0;

  if (origin->x > target->se.x) {
    deltax = origin->x - target->se.x;
```

```c
  } else if (origin->x < target->nw.x) {
    deltax = target->nw.x - origin->x;
  }

  if (origin->y < target->se.y) {
    deltay = target->se.y - origin->y;
  } else if (origin->y > target->nw.y) {
    deltay = origin->y - target->nw.y;
  }

  if (deltay == 0 && deltax == 0) {
    return 0.0; // The point is within the quad
  } else {
    return sqrt(pow(deltay, 2) + pow(deltax, 2)); // Pythagoras
  }
}

int64_t relevant_forces(body* vec, double cutoff_distance, quads* root) {
  point origin = vec->position;
  int64_t counter = 0;
  for (int i = 0; i < 4; i++) {
    double distance = distance_to_quad(&origin, root->children[i]);
    if (distance > cutoff_distance && root->children[i]->child_count) {
      quads* target = root->children[i];
      double pdistance = point_distance(vec->position, target->center_of_mass);
      double magnitude = point_magnitude(vec->mass, target->sum_mass, pdistance);
      point direction = point_direction(vec->position, target->center_of_mass);
      vec->force.x = vec->force.x + magnitude*direction.x/pdistance;
      vec->force.y = vec->force.y + magnitude*direction.y/pdistance;
      counter++;
    } else if (root->children[i]->child_count > QUADS_MAX_ELEMENTS) {
      counter += relevant_forces(vec, cutoff_distance, root->children[i]);
    } else if (root->children[i]->child_count == 0) {
      continue;
    } else {
      for (int j = 0; j < root->children[i]->child_count; j++) {
        body* target = root->children[i]->bodies[j];
        double pdistance = point_distance(vec->position, target->position);
        if (pdistance == 0) {
          continue;
        }
        double magnitude = point_magnitude(vec->mass, target->mass, pdistance);
        point direction = point_direction(vec->position, target->position);
        vec->force.x = vec->force.x + magnitude*direction.x/pdistance;
        vec->force.y = vec->force.y + magnitude*direction.y/pdistance;
        counter++;
      }
    }
  }
  return counter;
}

void calculate_forces(int64_t count, quads* root, double cutoff_distance) {
  int64_t comparisons = 0;
  for (int64_t i = 0; i < count; i++) {
    comparisons += relevant_forces(root->bodies[i], cutoff_distance, root);
  }
#ifdef DEBUG_MODE
  int64_t naive_approx = (count*count)/2;
  printf("%ld/%ld comparisons, %lf%% saved\n", comparisons, naive_approx,
     100*(1-((1.0*comparisons)/naive_approx)));
#endif
}

void move_bodies(int64_t count, quads* root) {
  /* Apply the forces of the bodies using the common apply_deltav function */
  body** vec = root->bodies;
  for (int64_t i = 0; i < count; i++){
    apply_deltav(vec[i]);
  }
}

void insert_body(quads* quad, body* o) {
  quad->bodies[quad->child_count] = o;
  quad->child_count++;

  quad->nw.x = min(quad->nw.x, o->position.x);
  quad->nw.y = max(quad->nw.y, o->position.y);
  quad->se.x = max(quad->se.x, o->position.x);
  quad->se.y = min(quad->se.y, o->position.y);
}

void inner_divide(quads* quad) {
  if (quad->child_count > QUADS_MAX_ELEMENTS) {
    divide(quad->child_count, quad);

    point mass_position_sum;
    mass_position_sum.x = 0;
    mass_position_sum.y = 0;

    for (int i = 0; i < 4; i++) {
      mass_position_sum.x += quad->children[i]->
          center_of_mass.x
        * quad->children[i]->sum_mass;
      mass_position_sum.y += quad->children[i]->
          center_of_mass.y
        * quad->children[i]->sum_mass;
    }

    if (quad->sum_mass != 0) {
      quad->center_of_mass.x = mass_position_sum.x / quad->
          sum_mass;
      quad->center_of_mass.y = mass_position_sum.y / quad->
          sum_mass;
    }
  } else {
    point mass_position_sum;
    mass_position_sum.x = 0;
    mass_position_sum.y = 0;

    for (int child = 0; child < quad->child_count; child++) {
      mass_position_sum.x += quad->bodies[child]->position.x
          *
        quad->bodies[child]->mass;
      mass_position_sum.y += quad->bodies[child]->position.y
          *
        quad->bodies[child]->mass;
      quad->sum_mass += quad->bodies[child]->mass;
    }

    if (quad->sum_mass != 0) {
      quad->center_of_mass.x = mass_position_sum.x / quad->
          sum_mass;
      quad->center_of_mass.y = mass_position_sum.y / quad->
          sum_mass;
    }
  }
}

quads* init_child(int id, point middle, int64_t parent_count)
    {
    quads* child = malloc(sizeof(quads));
    child->id = id;
    child->child_count = 0;
    child->sum_mass = 0;
    child->center_of_mass.x = 0;
    child->center_of_mass.y = 0;
    child->bodies = malloc(sizeof(body*)*parent_count);
    child->se.x = middle.x;
    child->se.y = middle.y;
    child->nw.x = middle.x;
    child->nw.y = middle.y;
    return child;
}

point get_middle(int64_t count, body** vec) {
  point middle;
  middle.x = vec[0]->position.x;
  middle.y = vec[0]->position.y;
  for (int64_t i = 1; i < count; i++) {
    middle.x += vec[i]->position.x;
    middle.y += vec[i]->position.y;
  }
  middle.x = middle.x / count;
  middle.y = middle.y / count;
  return middle;
}

void divide(int64_t count, quads* root) {
  body** vec = root->bodies;
  point middle = get_middle(count, vec);
  root->sum_mass = 0;
  for (int i = 0; i < 4; i++) {
    root->children[i] = init_child(i, middle, count);
  }

  for (int64_t i = 0; i < count; i++) {
    root->sum_mass += vec[i]->mass;
    if (vec[i]->position.y > middle.y) { // N
      if (vec[i]->position.x > middle.x) { // NE
        insert_body(root->children[0], vec[i]);
      } else if (vec[i]->position.x <= middle.x) { // NW
        insert_body(root->children[1], vec[i]);
      }
    } else if (vec[i]->position.y <= middle.y) { // S
```

```c
      if (vec[i]->position.x < middle.x) { // SW
        insert_body(root->children[2], vec[i]);
      } else if (vec[i]->position.x >= middle.x){ // SE
        insert_body(root->children[3], vec[i]);
      }
    } else {
      printf("Error! x %lf y %lf\n", vec[i]->position.x, vec[i]->position.y);
    }
  }
  for (int i = 0; i < 4; i++) {
    inner_divide(root->children[i]);
  }
}

int main(int argc, char* argv[]) {
  /* Run the simulation */
  int time_limit = TIME_DEFAULT;
  int n_bodies = BODIES_DEFAULT;
  double cutoff_distance = CUTOFF_DISTANCE_DEFAULT;

  /* Command line arguments */
  if (argc > 1) {
    n_bodies = atoi(argv[1]);
  }
  if (argc > 2) {
    time_limit = atoi(argv[2]);
  }
  if (argc > 3) {
    cutoff_distance = atof(argv[3]);
  }

  /* Initialize the bodies at their first position */
  point origo;
  origo.x = 0;
  origo.y = 0;
  quads* root = init_child(0, origo, n_bodies);
  for (int i = 0; i < n_bodies; i++) {
    root->bodies[i] = malloc(sizeof(body));
    row_of_twenty(root->bodies[i], i);
    root->nw.x = min(root->nw.x, root->bodies[i]->position.x);
    root->nw.y = max(root->nw.y, root->bodies[i]->position.y);
    root->se.x = max(root->se.x, root->bodies[i]->position.x);
    root->se.y = min(root->se.y, root->bodies[i]->position.y);
  }

  printf("[simulation] %d bodies over %d time steps -- nlogn\
     n", n_bodies, time_limit);
  struct timeval start = start_timer();
#ifdef DEBUG_MODE
  FILE* output = fopen("output", "w");
#endif
  /* Do simulation */
  for (int64_t t = 0; t < time_limit; t++) {
    divide(n_bodies, root);
    calculate_forces(n_bodies, root, cutoff_distance);
    move_bodies(n_bodies, root);
    clean_tree(root, 0);
#ifdef DEBUG_MODE
    /* Avoid I/O unless debug-mode is activated */
    for (int64_t i = 0; i < n_bodies; i++) {
      fprintf(output, "%ld %ld %lf %lf\n", t, i, root->bodies
          [i]->position.x,
          root->bodies[i]->position.y);
    };
#endif
  }
  stop_timer(start);
#ifdef DEBUG_MODE
  fclose(output);
#endif

}
```

F Listings for section 2.4

```c
#include "gravn.h"
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <stdbool.h>
#include <math.h>
#include <pthread.h>

#define CUTOFF_DISTANCE_DEFAULT 2
#define QUADS_MAX_ELEMENTS 5

typedef struct quads {
  int64_t id;
  struct quads* children[4];
  body** bodies;
  int64_t child_count;
  double sum_mass;
  point center_of_mass;
  /* Corners for the square */
  point nw;
  point se;
  /* Parallelism */
  bool done;
  pthread_mutex_t lock;
} quads;

typedef struct worker_info {
  int64_t worker_id;
  int64_t total_workers;
  int64_t count;
  int64_t time_limit;
  double cutoff_distance;
  FILE* output;
  quads* root;
} worker_info;

void divide(int64_t, quads*);
void* Worker (void*);

double point_distance(point a, point b) {
  double rv = sqrt(pow(a.x - b.x, 2) + pow(a.y - b.y, 2));
  return rv;
}

point point_direction(point a, point b) {
  point direction;
  direction.x = b.x - a.x;
  direction.y = b.y - a.y;
  return direction;
}

double point_magnitude(double mass_a, double mass_b, double
    distance) {
  return ((NEWTON_G*mass_a*mass_b) / pow(distance, 2));
}

void clean_tree(quads* root, int64_t level) {
  if (level == 0) {
    for (int i = 0; i < 4; i++) {
      clean_tree(root->children[i], level+1);
    }
  }
  else if (root->child_count > QUADS_MAX_ELEMENTS) {
    for (int i = 0; i < 4; i++) {
      clean_tree(root->children[i], level+1);
    }
    free(root->bodies);
    pthread_mutex_destroy(&root->lock);
    free(root);
  } else {
    free(root->bodies);
    pthread_mutex_destroy(&root->lock);
    free(root);
  }
}

double distance_to_quad(point* origin, quads* target) {
  /* Calculates the distance to the closes point at the quad
      target from the
   * point origin */

  // FIXME Every point should be within one of the quads
  double deltax = 0, deltay = 0;

  if (origin->x > target->se.x) {
    deltax = origin->x - target->se.x;
  } else if (origin->x < target->nw.x) {
    deltax = target->nw.x - origin->x;
  }

  if (origin->y < target->se.y) {
    deltay = target->se.y - origin->y;
  } else if (origin->y > target->nw.y) {
    deltay = origin->y - target->nw.y;
  }

  if (deltay == 0 && deltax == 0) {
    return 0.0; // The point is within the quad
  } else {
    return sqrt(pow(deltay, 2) + pow(deltax, 2)); // Pythagoras
  }
}

int64_t relevant_forces(body* vec, double cutoff_distance, quads* root) {
  point origin = vec->position;
  int64_t counter = 0;
  for (int i = 0; i < 4; i++) {
    double distance = distance_to_quad(&origin, root->children[i]);
    if (distance > cutoff_distance && root->children[i]->child_count) {
      quads* target = root->children[i];
      double pdistance = point_distance(vec->position, target->center_of_mass);
      double magnitude = point_magnitude(vec->mass, target->sum_mass, pdistance);
      point direction = point_direction(vec->position, target->center_of_mass);
      vec->force.x = vec->force.x + magnitude*direction.x/pdistance;
      vec->force.y = vec->force.y + magnitude*direction.y/pdistance;
      counter++;
    } else if (root->children[i]->child_count > QUADS_MAX_ELEMENTS) {
      counter += relevant_forces(vec, cutoff_distance, root->children[i]);
    } else if (root->children[i]->child_count == 0) {
      continue;
    } else {
      for (int j = 0; j < root->children[i]->child_count; j++) {
        body* target = root->children[i]->bodies[j];
        double pdistance = point_distance(vec->position, target->position);
        if (pdistance == 0) {
          continue;
        }
        double magnitude = point_magnitude(vec->mass, target->mass, pdistance);
        point direction = point_direction(vec->position, target->position);
        vec->force.x = vec->force.x + magnitude*direction.x/
```

```
            pdistance;
        vec->force.y = vec->force.y + magnitude*direction.y/
            pdistance;
        counter++;
      }
    }
  }
  return counter;
}

void calculate_forces(worker_info* data) {
  int64_t count = data->count;
  quads* root = data->root;
  double cutoff_distance = data->cutoff_distance;
  int64_t comparisons = 0;
  for (int64_t i = data->worker_id; i < count; i+=data->
      total_workers) {
    comparisons += relevant_forces(root->bodies[i],
        cutoff_distance, root);
  }
#ifdef DEBUG_MODE
  int64_t naive_approx = (count*count)/2;
  printf("%ld/%ld comparisons, %lf%% saved\n", comparisons,
      naive_approx,
       100*(1-((1.0*comparisons)/naive_approx)));
#endif
}

void move_bodies(worker_info* data) {
  /* Apply the forces of the bodies using the common
      apply_deltav function */
  body** vec = data->root->bodies;
  for (int64_t i = data->worker_id; i < data->count; i+=data
      ->total_workers){
    apply_deltav(vec[i]);
  }
}

void insert_body(quads* quad, body* o) {
  quad->bodies[quad->child_count] = o;
  quad->child_count++;

  quad->nw.x = min(quad->nw.x, o->position.x);
  quad->nw.y = max(quad->nw.y, o->position.y);
  quad->se.x = max(quad->se.x, o->position.x);
  quad->se.y = min(quad->se.y, o->position.y);
}

void inner_divide(quads* quad) {
```

```c
  if (quad->child_count > QUADS_MAX_ELEMENTS) {
    divide(quad->child_count, quad);

    point mass_position_sum;
    mass_position_sum.x = 0;
    mass_position_sum.y = 0;

    for (int i = 0; i < 4; i++) {
      mass_position_sum.x += quad->children[i]->center_of_mass.x
        * quad->children[i]->sum_mass;
      mass_position_sum.y += quad->children[i]->center_of_mass.y
        * quad->children[i]->sum_mass;
    }

    if (quad->sum_mass != 0) {
      quad->center_of_mass.x = mass_position_sum.x / quad->sum_mass;
      quad->center_of_mass.y = mass_position_sum.y / quad->sum_mass;
    }
  } else {
    point mass_position_sum;
    mass_position_sum.x = 0;
    mass_position_sum.y = 0;

    for (int child = 0; child < quad->child_count; child++) {
      mass_position_sum.x += quad->bodies[child]->position.x *
        quad->bodies[child]->mass;
      mass_position_sum.y += quad->bodies[child]->position.y *
        quad->bodies[child]->mass;
      quad->sum_mass += quad->bodies[child]->mass;
    }

    if (quad->sum_mass != 0) {
      quad->center_of_mass.x = mass_position_sum.x / quad->sum_mass;
      quad->center_of_mass.y = mass_position_sum.y / quad->sum_mass;
    }
  }
}

quads* init_child(int id, point middle, int64_t parent_count) {
    quads* child = malloc(sizeof(quads));
    child->id = id;
    child->child_count = 0;
    child->sum_mass = 0;
    child->center_of_mass.x = 0;
    child->center_of_mass.y = 0;
    child->bodies = malloc(sizeof(body*)*parent_count);
    child->se.x = middle.x;
    child->se.y = middle.y;
    child->nw.x = middle.x;
    child->nw.y = middle.y;
    child->done = false;
    pthread_mutex_init(&child->lock, NULL);
    return child;
}

point get_middle(int64_t count, body** vec) {
  point middle;
  middle.x = vec[0]->position.x;
  middle.y = vec[0]->position.y;
  for (int64_t i = 1; i < count; i++) {
    middle.x += vec[i]->position.x;
    middle.y += vec[i]->position.y;
  }
  middle.x = middle.x / count;
  middle.y = middle.y / count;
  return middle;
}

void divide(int64_t count, quads* root) {
  body** vec = root->bodies;
  point middle = get_middle(count, vec);
  root->sum_mass = 0;
  for (int i = 0; i < 4; i++) {
    root->children[i] = init_child(i, middle, count);
  }

  for (int64_t i = 0; i < count; i++) {
    root->sum_mass += vec[i]->mass;
    if (vec[i]->position.y > middle.y) { // N
      if (vec[i]->position.x > middle.x) { // NE
        insert_body(root->children[0], vec[i]);
      } else if (vec[i]->position.x <= middle.x) { // NW
        insert_body(root->children[1], vec[i]);
      }
    } else if (vec[i]->position.y <= middle.y) { // S
      if (vec[i]->position.x < middle.x) { // SW
        insert_body(root->children[2], vec[i]);
      } else if (vec[i]->position.x >= middle.x){ // SE
        insert_body(root->children[3], vec[i]);
      }
    } else {
      printf("Error! x %lf y %lf\n", vec[i]->position.x, vec[
          i]->position.y);
    }
  }
  for (int i = 0; i < 4; i++) {
    inner_divide(root->children[i]);
  }
}

int main(int argc, char* argv[]) {
  /* Run the simulation */
  int time_limit = TIME_DEFAULT;
  int n_bodies = BODIES_DEFAULT;
  double cutoff_distance = CUTOFF_DISTANCE_DEFAULT;
  int n_workers = WORKERS_DEFAULT;

  /* Command line arguments */
  if (argc > 1) {
    n_bodies = atoi(argv[1]);
  }
  if (argc > 2) {
    time_limit = atoi(argv[2]);
  }
  if (argc > 3) {
    n_workers = atoi(argv[3]);
    if (n_workers > 64) {
      n_workers = 64;
    }
  }
  if (argc > 4) {
    cutoff_distance = atof(argv[4]);
  }

  /* The thread variables and properties for the workers */
  pthread_t worker_threads[n_workers];
  worker_info workers_data[n_workers];

  /* set global thread attributes */
  pthread_attr_t attr;
  pthread_attr_init(&attr);
  pthread_attr_setscope(&attr, PTHREAD_SCOPE_SYSTEM);


  /* Initialize the bodies at their first position */

  point origo;
  origo.x = 0;
  origo.y = 0;
  quads* root = init_child(0, origo, n_bodies);
#ifdef DEBUG_MODE
  FILE* output = fopen("output", "w");
#endif

  printf("[simulation] %d bodies over %d time steps with %d
      workers -- nlogn\n",
        n_bodies, time_limit, n_workers);
  struct timeval start = start_timer();
  for (int w = 0; w < n_workers; w++) {
    /* Iterate over all the workers and create their
        properties */
    workers_data[w].worker_id = w;
    workers_data[w].total_workers = n_workers;
    workers_data[w].count = n_bodies;
    workers_data[w].root = root;
#ifdef DEBUG_MODE
    workers_data[w].output = output;
#endif
    workers_data[w].time_limit = time_limit;
    workers_data[w].cutoff_distance = cutoff_distance;
  }
  for (int w = 0; w < n_workers; w++) {
    pthread_create(&worker_threads[w], &attr, Worker, (void*)
        &workers_data[w]);
  }
  for (int w = 0; w < n_workers; w++) {
    pthread_join(worker_threads[w], NULL);
  }
  stop_timer(start);
#ifdef DEBUG_MODE
  fclose(output);
#endif

}

void* Worker (void* in) {
  worker_info* data = (worker_info*) in;
  quads* root = data->root;

  for (int i = data->worker_id; i < data->count; i+=data->
      total_workers) {
    root->bodies[i] = malloc(sizeof(body));
    row_of_twenty(root->bodies[i], i);
    root->nw.x = min(root->nw.x, root->bodies[i]->position.x)
        ;
    root->nw.y = max(root->nw.y, root->bodies[i]->position.y)
        ;
    root->se.x = max(root->se.x, root->bodies[i]->position.x)
        ;
    root->se.y = min(root->se.y, root->bodies[i]->position.y)
        ;
  }
  barrier(data->total_workers);

  /* Do simulation */
  for (int64_t t = 0; t < data->time_limit; t++) {
    if (data->worker_id == 0)
      divide(data->count, root);
    barrier(data->total_workers);
    calculate_forces(data);
    barrier(data->total_workers);
    move_bodies(data);
    barrier(data->total_workers);
    if (data->worker_id == 0)
      clean_tree(root, 0);
    barrier(data->total_workers);
#ifdef DEBUG_MODE
    /* Avoid I/O unless debug-mode is activated */
    for (int64_t i = 0; i < data->count; i++) {
      fprintf(data->output, "%ld %ld %lf %lf\n", t, i, root->
          bodies[i]->position.x,
          root->bodies[i]->position.y);
    };
#endif
  }
  return NULL;
}
```

G Broken parallellization of 2.4

```
diff --git a/project/par_nlg.c b/project/par_nlg.c
index cfabd39..f5146a6 100644
--- a/project/par_nlg.c
+++ b/project/par_nlg.c
@@ -21,6 +21,7 @@ typedef struct quads {
   point se;
   /* Parallelism */
   bool done;
+  bool initd;
   pthread_mutex_t lock;
 } quads;

@@ -34,7 +35,7 @@ typedef struct worker_info {
   quads* root;
 } worker_info;

-void divide(int64_t, quads*);
+void divide(int64_t, quads*, worker_info*);
 void* Worker (void*);

 double point_distance(point a, point b) {
@@ -58,6 +59,8 @@ void clean_tree(quads* root, int64_t level) {
     for (int i = 0; i < 4; i++) {
       clean_tree(root->children[i], level+1);
     }
+    root->initd = false;
+    root->done = false;
   }
   else if (root->child_count > QUADS_MAX_ELEMENTS) {
     for (int i = 0; i < 4; i++) {
@@ -77,7 +80,6 @@ double distance_to_quad(point* origin, quads* target) {
   /* Calculates the distance to the closes point at the quad target from the
    * point origin */

-  // FIXME Every point should be within one of the quads
   double deltax = 0, deltay = 0;

   if (origin->x > target->se.x) {
@@ -158,6 +160,7 @@ void move_bodies(worker_info* data) {
 }

 void insert_body(quads* quad, body* o) {
+  pthread_mutex_lock(&quad->lock);
   quad->bodies[quad->child_count] = o;
   quad->child_count++;

@@ -165,12 +168,14 @@ void insert_body(quads* quad, body* o)
    {
   quad->nw.y = max(quad->nw.y, o->position.y);
   quad->se.x = max(quad->se.x, o->position.x);
   quad->se.y = min(quad->se.y, o->position.y);
+  pthread_mutex_unlock(&quad->lock);
 }

-void inner_divide(quads* quad) {
+void inner_divide(quads* quad, worker_info* data) {
   if (quad->child_count > QUADS_MAX_ELEMENTS) {
-    divide(quad->child_count, quad);
+    divide(quad->child_count, quad, data);

+    pthread_mutex_lock(&quad->lock);
     point mass_position_sum;
     mass_position_sum.x = 0;
     mass_position_sum.y = 0;
@@ -186,7 +191,9 @@ void inner_divide(quads* quad) {
       quad->center_of_mass.x = mass_position_sum.x / quad->
           sum_mass;
       quad->center_of_mass.y = mass_position_sum.y / quad->
          sum_mass;
     }
+    pthread_mutex_unlock(&quad->lock);
   } else {
+    pthread_mutex_lock(&quad->lock);
     point mass_position_sum;
     mass_position_sum.x = 0;
     mass_position_sum.y = 0;
@@ -203,11 +210,14 @@ void inner_divide(quads* quad) {
       quad->center_of_mass.x = mass_position_sum.x / quad->
          sum_mass;
       quad->center_of_mass.y = mass_position_sum.y / quad->
          sum_mass;
     }
+    pthread_mutex_unlock(&quad->lock);
   }
+  quad->done = true;
 }

 quads* init_child(int id, point middle, int64_t parent_count
    ) {
     quads* child = malloc(sizeof(quads));
+    child->initd = false;
     child->id = id;
     child->child_count = 0;
     child->sum_mass = 0;
@@ -236,34 +246,44 @@ point get_middle(int64_t count, body**
    vec) {
   return middle;
 }

-void divide(int64_t count, quads* root) {
+void divide(int64_t count, quads* root, worker_info* data) {
   body** vec = root->bodies;
-  point middle = get_middle(count, vec);
-  root->sum_mass = 0;
-  for (int i = 0; i < 4; i++) {
-    root->children[i] = init_child(i, middle, count);
-  }
-
-  for (int64_t i = 0; i < count; i++) {
-    root->sum_mass += vec[i]->mass;
-    if (vec[i]->position.y > middle.y) { // N
-      if (vec[i]->position.x > middle.x) { // NE
-        insert_body(root->children[0], vec[i]);
-      } else if (vec[i]->position.x <= middle.x) { // NW
-        insert_body(root->children[1], vec[i]);
+  point middle;
+  while (root->initd == false) {
+    if (pthread_mutex_trylock(&root->lock) == 0) {
+      if (root->initd == true)
+        continue;
+      middle = get_middle(count, vec);
+      for (int i = 0; i < 4; i++) {
+        root->children[i] = init_child(i, middle, count);
       }
-    } else if (vec[i]->position.y <= middle.y) { // S
-      if (vec[i]->position.x < middle.x) { // SW
-        insert_body(root->children[2], vec[i]);
-      } else if (vec[i]->position.x >= middle.x){ // SE
-        insert_body(root->children[3], vec[i]);
+      root->initd = true;
+
+      for (int64_t i = 0; i < count; i++) {
+        root->sum_mass += vec[i]->mass;
+        if (vec[i]->position.y > middle.y) { // N
+          if (vec[i]->position.x > middle.x) { // NE
+            insert_body(root->children[0], vec[i]);
+          } else if (vec[i]->position.x <= middle.x) { // NW
+            insert_body(root->children[1], vec[i]);
+          }
+        } else if (vec[i]->position.y <= middle.y) { // S
+          if (vec[i]->position.x < middle.x) { // SW
+            insert_body(root->children[2], vec[i]);
+           } else if (vec[i]->position.x >= middle.x){ // SE
+             insert_body(root->children[3], vec[i]);
+           }
+         } else {
+           printf("Error! x %lf y %lf\n", vec[i]->position.x, vec[i]->position.y);
+         }
        }
-     } else {
-       printf("Error! x %lf y %lf\n", vec[i]->position.x, vec[i]->position.y);
      }
    }
+   pthread_mutex_unlock(&root->lock);
    for (int i = 0; i < 4; i++) {
-     inner_divide(root->children[i]);
+     if (!root->children[i]->done) {
+       inner_divide(root->children[i], data);
+     }
    }
  }

@@ -355,8 +375,7 @@ void* Worker (void* in) {

    /* Do simulation */
    for (int64_t t = 0; t < data->time_limit; t++) {
-     if (data->worker_id == 0)
-       divide(data->count, root);
+     divide(data->count, root, data);
      barrier(data->total_workers);
      calculate_forces(data);
      barrier(data->total_workers);
```

H Listings for help script plot.py

```python
from matplotlib import pyplot as plt
from matplotlib import animation
import json

data = []
t = 0;
with open('output') as in_f:
  for line in in_f:
    line = line.strip()
    line = line.split(' ')
    line[0] = int(line[0])
    line[2] = float(line[2])
    line[3] = float(line[3])
    if line[0] > t:
      t = line[0]
    data.append(line)

figure = plt.figure()
plt.xkcd()
plt.title('N-body Simulation')
frames = []
for i in range(0, t, 5):
  frame = filter(lambda datum: datum[0] == i, data)
  xs = list(map(lambda f: f[2], frame))
  ys = list(map(lambda f: f[3], frame))
  frames.append((plt.scatter(xs, ys),))
  print("added frame %d" % i)

gif = animation.ArtistAnimation(figure, frames, interval=50,
    repeat_delay=3000, blit=True)

gif.save('nbody.mp4', fps=24, extra_args=['-vcodec', 'libx264
    '])
print("Printed animation to 'nbody.mp4'")

plt.show()
```

I Listings for help script performance.py

```
#!/usr/bin/env python3
import subprocess

processes = ['./par_nlg.out', './par_sq.out', './seq_sq.out',
    './seq_nlg.out']
sizes = ['120', '180', '240']
cores = ['1', '2', '3', '4']

def run_process(run):
  rv = subprocess.check_output(run).decode('utf-8')
  rv = rv.strip()
  rv = rv.split('\n')
  for i, line in enumerate(rv):
    rv[i] = line.split(' ')
  if 'workers' in rv[0]:
    cpu_count = int(rv[0][8])
  else:
    cpu_count = 1
  bodies = int(rv[0][1])
  time = float(rv[1][1])
  return (p, bodies, time, cpu_count)


for p in processes:
  runs = []
  for size in sizes:
    if 'par' in p:
      for c in cores:
        runs.append([p, size, '50000', c])
    else:
      runs.append([p, size, '50000'])
  if 'nlg' in p:
    for run in runs:
      run.append('0.7')


  for run in runs:
    results = []
    for i in range(5):
      results.append(run_process(run))
    for i in range(2):
      results.remove(min(results, key=lambda rvec: rvec[2]))
      results.remove(max(results, key=lambda rvec: rvec[2]))
    print(results[0])
```

J Listings for Makefile

```
CC=gcc
CFLAGS=-std=c99 -g -Wall -D_XOPEN_SOURCE=600 -O3 -lm

all: seq_sq.out seq_sq.debug.out par_sq.out par_sq.debug.out seq_nlg.out \
        seq_nlg.debug.out par_nlg.out par_nlg.debug.out

seq_sq.out: seq_sq.c gravn_common.o
        $(CC) $(CFLAGS) -o $@ $< gravn_common.o

seq_sq.debug.out: seq_sq.c gravn_common.o
        $(CC) $(CFLAGS) -DDEBUG_MODE -o $@ $< gravn_common.o

par_sq.out: par_sq.c gravn_common.o
        $(CC) $(CFLAGS) -lpthread -o $@ $< gravn_common.o

par_sq.debug.out: par_sq.c gravn_common.o
        $(CC) $(CFLAGS) -DDEBUG_MODE -lpthread -o $@ $< gravn_common.o

seq_nlg.out: seq_nlg.c gravn_common.o
        $(CC) $(CFLAGS) -o $@ $< gravn_common.o

seq_nlg.debug.out: seq_nlg.c gravn_common.o
        $(CC) $(CFLAGS) -DDEBUG_MODE -o $@ $< gravn_common.o

par_nlg.out: par_nlg.c gravn_common.o
        $(CC) $(CFLAGS) -lpthread -o $@ $< gravn_common.o

par_nlg.debug.out: par_nlg.c gravn_common.o
        $(CC) $(CFLAGS) -DDEBUG_MODE -lpthread -o $@ $< gravn_common.o

gravn_common.o: gravn_common.c
        $(CC) $(CFLAGS) -c -o $@ $<

clean:
        $(RM) ./*.out
        $(RM) ./*.o
```