UPPSALA
UNIVERSITET

DEPARTMENT OF INFORMATION TECHNOLOGY - Course 2020

HIGH PERFORMANCE PROGRAMMING - ASSIGNMENT 4: The Barnes-Hut Method

KIERAN BARBER

PANAGIOTIS PAPIAS

Teacher
Sverker Holmgren

Contents

1 The Problem **2**

2 The Solution **3**
- 2.1 Quadtree 3
- 2.2 Forces 6
- 2.3 Other Files 7

3 CPU Configuration **7**

4 Performance and Discussion **9**
- 4.1 Timings - Getting up to speed 9
- 4.2 Timings - Which section to focus on 9
- 4.3 Timings - Optimised 9
- 4.4 Unsuccessful Optimisations 11
- 4.5 Complexity 12

1 The Problem

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

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

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

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

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

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

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

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

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

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

2 The Solution

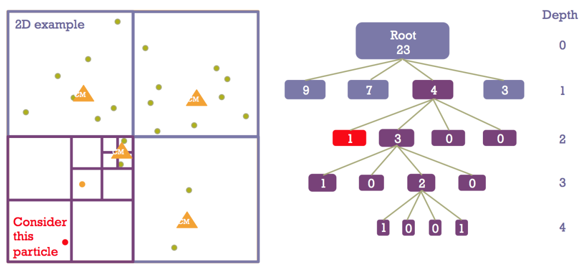

Figure 1: Diagram of Barnes Hut method

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

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

- `box_bounds.c box_boundsh`
- forces.c forces.h
- galsim.c
- particle.c particle.h
- quad.c quad.h
- vector.c vector.h
- Makefile
- timings.sh

2.1 Quadtree

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

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

The second case, is that we traverse the tree and we end up on a node that is taken, but with no children. In

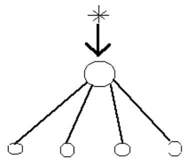

Figure 2: Diagram of Empty Tree

this scenario, we must take the particle to one of the children and then also take the particle that was in the node into one of the other children. If they end up going to the same place again, then we traverse the tree until they are split into separate nodes. We update the mass and centre of mass for each particle at the same time. The final case in our code is if we already have a particle in one of the children spots but not all of them.

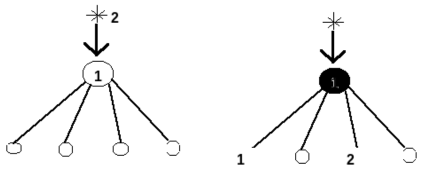

Figure 3: Diagram of Node with no children

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

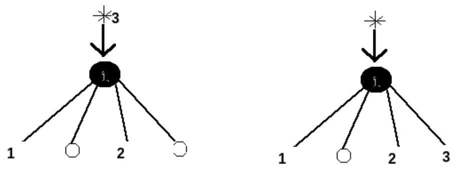

Figure 4: Diagram of Node with children

Quadtree code

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

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

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

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

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

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

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

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

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

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

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

2.2 Forces

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

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

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

Forces code

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

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

2.3 Other Files

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

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

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

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

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

3 CPU Configuration

```
Architecture:        x86_64
CPU op-mode(s):      32-bit, 64-bit
Byte Order:          Little Endian
CPU(s):              8
On-line CPU(s) list: 0-7
Thread(s) per core:  2
Core(s) per socket:  4
Socket(s):           1
```

```
NUMA node(s):        1
Vendor ID:           GenuineIntel
CPU family:          6
Model:               142
Model name:          Intel(R) Core(TM) i5-8250U CPU @ 1.60GHz
Stepping:            10
CPU MHz:             700.041
CPU max MHz:         3400,0000
CPU min MHz:         400,0000
BogoMIPS:            3600.00
Virtualization:      VT-x
L1d cache:           32K
L1i cache:           32K
L2 cache:            256K
L3 cache:            6144K
NUMA node0 CPU(s):   0-7
```

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

4 Performance and Discussion

4.1 Timings - Getting up to speed

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

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

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

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

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

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

4.2 Timings - Which section to focus on

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

4.3 Timings - Optimised

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

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

This is where our first optimisation is. In Version 2.2, we chose to move `vector_in_box` in to the forces file. This does indeed speed up the computation times for all apart from the -O0 flag.

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

Version 3 - `theta_bounds`

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

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

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

Version 4 - `theta_bounds`

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

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

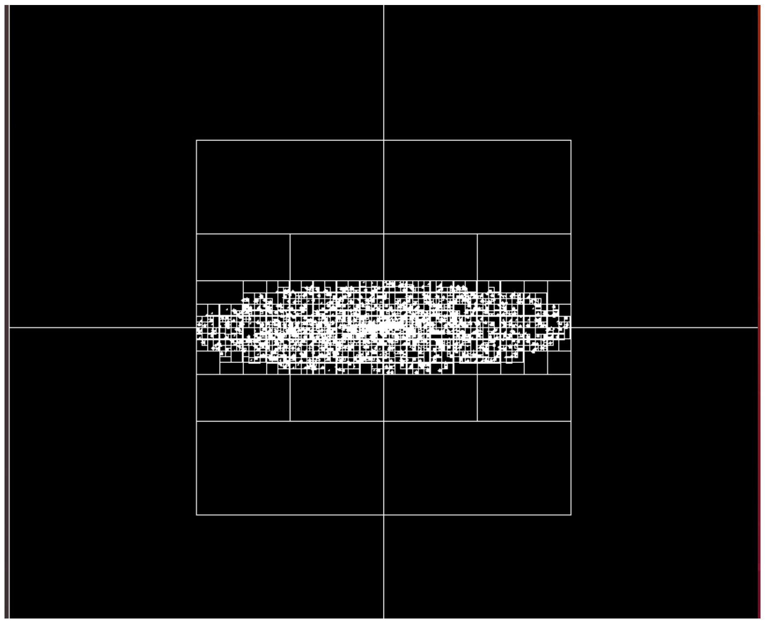

Figure 5: A simulation of a galaxy

4.4 Unsuccessful Optimisations

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

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

4.5 Complexity

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

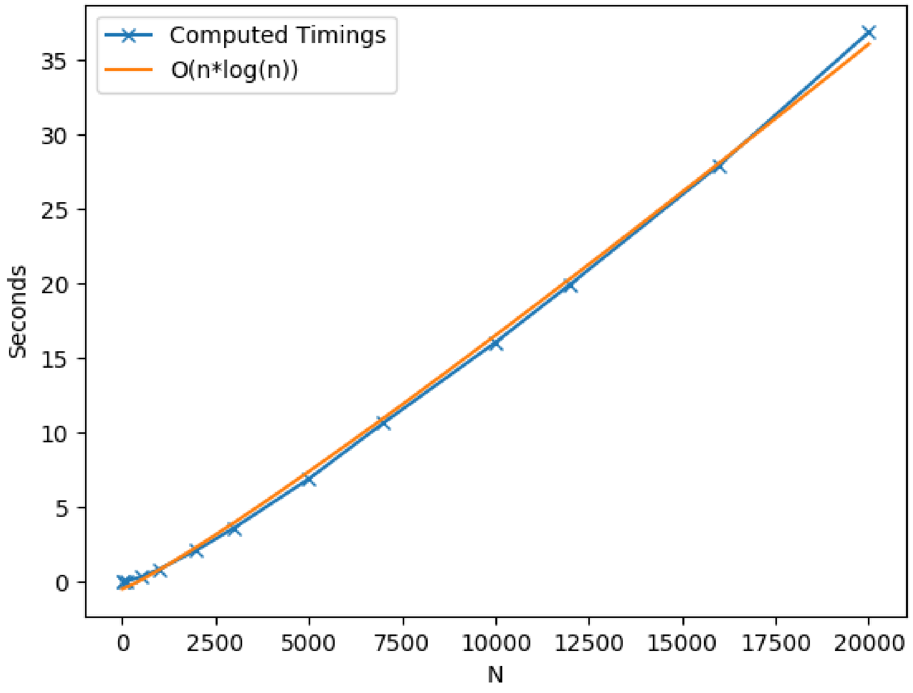

Figure 6: Time Complexity Plot