Assignment 4, Philipp Noel von Bachmann

March 4, 2022

1 Problem

In the N-body Problem, we want to predict the movement of n bodies that interact with each other over time. The interaction is given by the force they excert on each other, which in turn changes the acceleration of each body, thus the velocity and position. The problem is modeled by assuming pairwise interaction between the bodies and taking their sum as the total force excerted onto each body.

In this exercise, we use an algorithm known as the Barnes-Hut. This algorithm reduces the computational cost by approximating a group of starts that are far away as one single big star. Together with a representation known as a quad-tree, this enables us to reduce the computational complexity significantly.

2 Solution

2.1 Data structure

2.1.1 Body

Like in the previous assignment, we have the following 4 characteristics of each body:

- velocity
- position
- mass
- brightness

Mass and brightness are represented as double, since they are both continuous. As position and velocity are both vectors, they are represented as a struct of coordinates. Since we model the problem in 2D, each vector thus comprises of an x and y coordinate. The characteristics are collected in a struct called body. This format also allows for easy reading of the input gal file, since we can directly read each block into a body struct.

We also need to keep track of the accelerations of each body in an array of coordinates. We opt not to integrate this into the body struct to be able to retain easy writing and reading of .gal files.

2.1.2 Quadtree

A quadtree is a tree of nodes, where every node represents a subset of an area of interest. Each node can be either an internal or external node.

An external node is a node whose area contains only one body. If the area contains more than one body, the node is called an internal node. Instead of having an body that belongs to this node, the node has 4 child nodes which represent the 4 subareas. To construct a quadtree, we therefore split all nodes recursively until each leaf node is an external node. A node whose area contains no bodies is also considered a external node.

In general, we represent a node in a structure with the following attributes:

- lower: coordinate representing the lower edge of the area covered by the node
- upper: coordinate representing the upper edge of the area covered by the node
- children: A list containing a pointer to each of the children nodes if there are any, else a NULL pointer.
- body: A pointer to the body contained by this node if an external node, else a NULL pointer.
- total mass: A float representing the total mass of the bodies in the area of this node.
- center of mass: A coordinate, representing the center of all bodies contained by this node weighted by their mass.

2.2 Overview of Algorithm

In general, we have the following three steps of our main function:

- Load the data
- Perform the simulation
- Safe the output data

The main part is the simulation, which is done by recursively calling the step function which progresses the simulation by one timestep.

Algorithm 1 One environment step

```
procedure STEP
    for each body in all bodies do
        force = COMPUTE_FORCE(body, tree)
    for each body in all bodies do
        update the position of that body
    replace the old quadtree by a new empty one
    for each body in all bodies do
        Insert the body in the new quadtree
```

A crucial aspect is that we need to perform all force computations before we update the positions of our body, which will especially become important later for parallelization. In our initial implementation, we stick to rebuilding the quadtree after each step, instead of updating it. See also Section 3.4.2 for how this affects the runtime.

The force is computed by descending the quadtree:

Algorithm 2 Force computation

```
procedure COMPUTE_FORCE(body, current node)
    if current node is external then
        force = force between body and body of external node
    else
        for child in children of current node do
            if child is empty then
                continue
            if child is far enough away then
                force += force between body and a big body at center of child
            else need to descent into that child
                force += COMPUTE_FORCE(body, child)
    return force
```

Here, we safe the most amount of time compared to the naive algorithm from Assignment 3 in line 8, where we treat all bodies in the child as one big one and thus don't need to descent into that child.

3 Performance and Discussion for Serial code

3.1 Hardware information

- CPU: Apple M1 with 4 performance cores and 4 efficiency cores
- GPU: Apple M1 8 core.
- Compiler: Slang, Version 13.0.0
- Plugged into power adapter, this makes a big difference!

3.2 Performance Measurement script

The performance was measured with shell scripts to enable reproducability. The script mostly loops over the parameters of interest and computes the respective runtime. In the end we also check that the code is correct with the **compare_gals** file. Our time measurement was done within the c code with the help of the time **timeoftheday** function. We started the time measurement after the data loading and stopped the measurement before data writing.

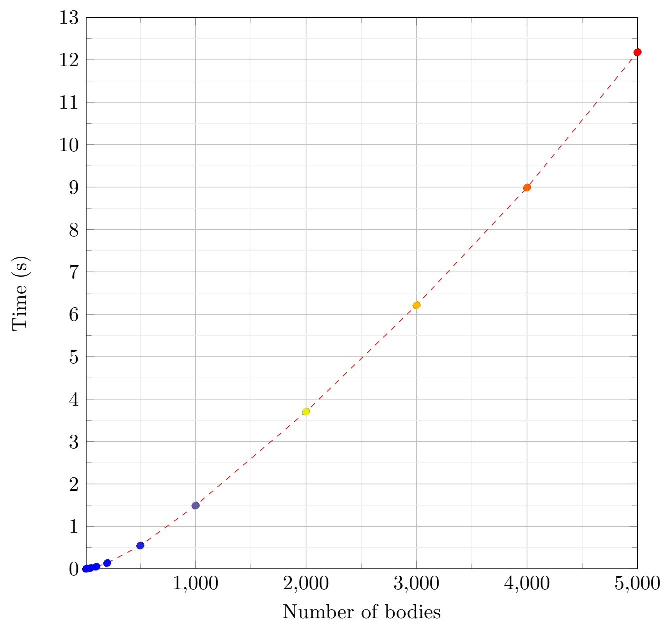
3.3 Calculation of $\theta_{\max}$

We calculate $\theta_{\max}$ as described in the Assignment, Section 5.2. This results in $\theta_{\max} = 0.25$, where an **pos_maxdiff** $= 0.000885759450$ was achieved.

3.4 Results

3.4.1 Baseline

The following graph shows the results of the algorithm with no optimization for a variable number of bodies, always using 100 steps:



Compared to the last assignment, these result are far quicker than the last assignment, for example for N=1000 we achieve 1.491261s compared to 6.502 in the last assignment. For large N, we see that the runtime becomes nearly linear.

3.4.2 Analysis of runtime

This section analyzes the runtime of the two main parts of algorithm 1:

- Tree: For the Barnes-Hut, we need to build and update the tree after each timestep.
- Body update: For each step, we need to update the forces and the body positions etc.

N bodies	10	100	1000
Tree time (s)	0.0003	0.003	0.042
Body update time (s)	0.001	0.045	1.449

We see that we spend most of the time on computing the update steps for the bodies and a negligible amount of time on the rebuilding of the tree. Therefore, we decide not to optimize the tree further by for example updating the tree instead of rebuilding, since the possible time to gain is very little. Instead we will focus on optimizing the force computation and position update of our body.

3.4.3 Inplace computation

In the baseline algorithm 2, we compute the force recursively if we need to descent into the children. However, this creates a lot of unneeded force objects, that we need to recursively pass around. Instead, we can just add the force to a global array of forces which gets initialized to 0 at the beginning of every step.

Algorithm 3 Force computation, assuming global array force_arr initialized to 0

```
procedure COMPUTE_FORCE_INPLACE(body, current node)
    if current node is external then
        force_arr[body] += force between body and body of external node
    else
        for child in children of current node do
            if child is far enough away then
                force_arr[body] += force between body and a big body at center of child
            else need to descent into that child
                COMPUTE_FORCE_INPLACE(body, child)
```

This leads to the following results:

N bodies	10	100	1000
time (s)	0.0003	0.004	1.117

We see that for $N = 1000$, this reduces the computation time from 1.491261s to 1.117s.

3.4.4 Early stopping

In a setup where we have a lot of distant bodies, the quadtree representation might be quite spare in the sense that we have a lof of emtpy nodes. Therefore, one idea is that at a minimum amount of nodes in that subtree, instead of descending that tree we just compute the naive update from Assignment 3 with all the bodies in that subtree. This would remove iterating over a lot of empty nodes and performing a lot of checks. Table 3.4.4 shows the results for 1000 bodies and 100 steps:

minimum N bodies	0 (baseline)	5	10
time (s)	1.119	1.370	1.704

We don't get the expected result, instead we see that our time increases. One reason might be that while there are a lot of empty nodes which take time iterating over and a lot of checks to determine if we need to descent into that child, these checks also gain time by not needing to descent into these childs. In total, this saves time. Therefore, we decide not to include this modification in the rest of the optimizations.

3.4.5 Memory usage

The quadtree for large galaxies will not fit into one cacheline, and we spend a lot of time loading new bodies in our memory. Therefore, the idea is to instead of updating each body in serial, load a subtree into the memory and then perform the force computation with that subtree for all bodies, and iterate over a number of subtrees.

Algorithm 4 One environment step with subtree update

```
1: procedure STEP
2:     subtree_list = list of nodes at a certain level of tree
3:     for subtree in subtree_list do
4:         for each body in all bodies do
5:             force += COMPUTE_FORCE(body, subtree)
6:     for each body in all bodies do
7:         update the position of that body
8:     replace the old quadtree by a new empty oen
9:     for each body in all bodies do
10:        Insert the body in the new quadtree
```

We investigate the runtime wrt the level we descent to build the subtree. Descending further down might lower the memory usage and therefore be more likely to fit in one cacheline on the one hand, but remove the chance to prune the tree earlier by approximating bodies as one big body. The results are shown for 1000 bodies and 100 steps.

N levels	0 (base)	1	2	3	4	5	10
time (s)	1.146	1.074	1.049	1.028	1.011	1.173	2.671

We see that in the best case, this speeds up computation by 0.13s for a level of 4. However as we further increase the level, we can't prune subtrees anymore and loose time again. In the end, we therefore fix the level at 4.

3.4.6 Optimization flags

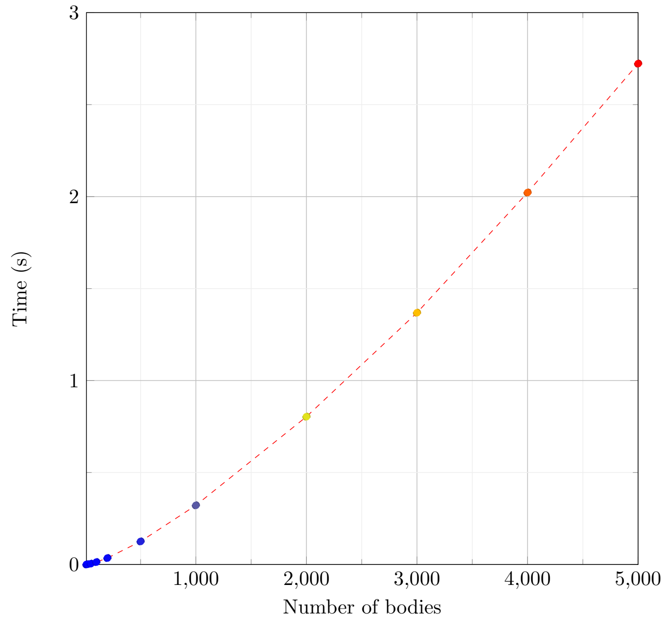
Finally, we use optimization flags to optimize the code further, as before results are shown for 1000 bodies and 100 steps.

	No flags	-O2	-O3
time (s)	1.022	0.337	0.325

We see that we achieve the best performance using -O3, therefore we use that option from now on.

3.5 Final code evaluation

The figure shows the results for the final code for 100 timesteps and a variable number of



bodies.

We can see that for a large number of bodies, the runtime becomes nearly linear. This is much better than $O(n^2)$ in Assignment 3.

4 Parallel implementation

4.1 Assignment 4

4.1.1 Code

If we look through the pseudo-code 1 we see that the force computation of each body is independent of all the other bodies. However we noted further, that we can only start updating the postion after we computed all the forces, otherwise we would get incorrect results. Therefore, we modify the main loop to:

Algorithm 5 One environment step with parallel updates

```
1: procedure STEP
2:     for thread in threads do
3:         Call thread with COMPUTE_FORCE_PARALLEL(start, end (for this thread))
4:     for thread in threads do
5:         JOIN(thread)
6:     for each body in all bodies do
7:         update the position of that body
8:     replace the old quadtree by a new empty oen
9:     for each body in all bodies do
10:        Insert the body in the new quadtree
```

and add the method:

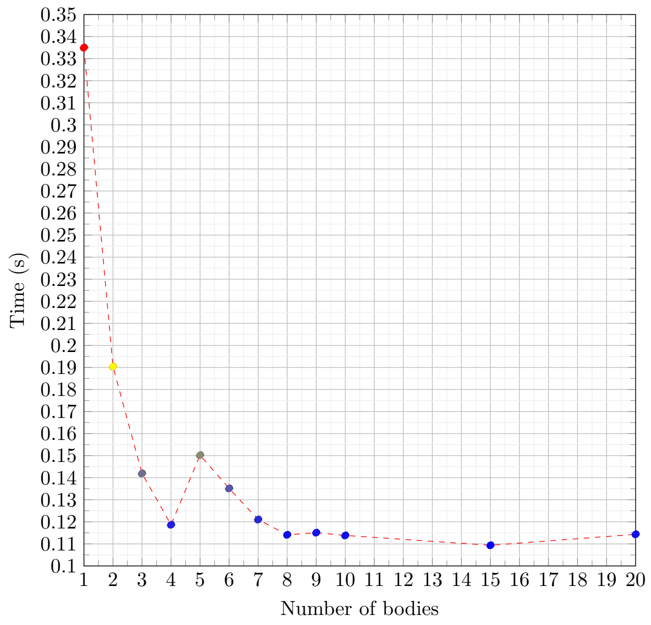
Algorithm 6 Force computation done by one thread

```
1: procedure COMPUTE_FORCE_PARALLEL(start, end)
2:     for each body in all bodies between start and end do
3:         force = COMPUTE_FORCE(body, tree)
```

Here, we divide the task between threads by giving each thread just a subset of bodies to update. As noted, we have to join the thread before we update our position. An alternative would be to use a barrier so that all threads wait until the others are finished computing the force, and then update the position in parallel. Note that for simplicity we only show the easy pseudo-code here, however all optimizations from the previous section like inplace computation and memory-optimization were also applied.

4.1.2 Results

The figure shows the speedup with the number of threads for 1000 bodies and 100 steps.



We can see that initally using more threads decrease our time nearly linear: For 1 thread we get a time of 0.33s and for 2 0.19s. However as we progress further, this increase flattens significantly until at around 8 threads, we come to a stop of increase, which makes sense considering the machine used has 8 cores. Interestingly, we don't get a linear decrease until 8 cores, as one would expect, rather we get an bump at 5 cores. One reason might be the architecture of the machine - The M1 Mac has 4 high performance and 4 efficiency cores, so the high performance cores probably have to wait for the efficiency cores to finish.

We also found that using a barrier or updating the positions in sequence after the join doesn't matter. This is probably because the operation first of all has linear time complexity and second is very cheap, since we mostly just need to write values to an array.

4.2 Assignment 3

4.2.1 Code

Recall the algorithm for assignment 3:

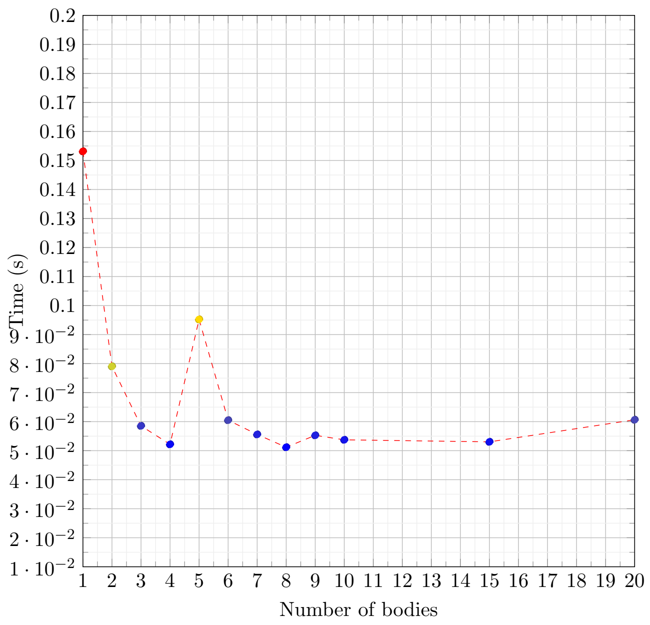
Algorithm 7 One environment step, Assignment 3

```
1: procedure STEP
2:     for each body in all bodies do
3:         for each second body in all bodies do
4:             force[body] += COMPUTE_FORCE(body, second body)
5:     for each body in all bodies do
6:         update the position of that body
```

Here, we see that we can parallelize the force computation step. However we need to wait with the update step, since otherwise force computations can be wrong. This is either achieved by calling join before the update and then updating in serial or implementing a barrier which lets all threads wait for the others to complete before the update, similar to Assignment 4.

4.2.2 Results

The figure shows the speedup with the number of threads for 1000 bodies and 100 steps.



We see the same pattern as in Assignment 4, when going over 4 cores our runtime rises again due to the same reasons. In general, we see that we surprisingly achieve a lower runtime however. This could be because we use mostly standard functions, thus making it easier for the compiler to do optimizations.

5 References

- For the timing, the **get_wall_seconds** from Lab06, exercise 7
- For data reading, the function **read_doubles_from_file** from **compare_gal_files**, but slightly modified
- For the barrier, the code from Lab09, exercise 4
- For the parallel assignment 3, the code from Assignment 3

- To get more familar with the Barnes-Hut, the explanation from `https://www.cs.princeton.edu/courses/archive/fall03/cs126/assignments/barnes-hut.html`, however no parts of the code.