Uppsala University

High Performance Programming

1TD062

Assignment 6

Authors:
Jiayi Yang, Dominik Lehnert

March 15, 2019

1 Introduction

The problem at hand deals with the calculation of mutual gravitational influences between a number of N objects in a two dimensional space. This so called N-Body problem is solved through the application of Newton's law of gravitation. The gravitational force of each of the N objects on all other N-1 objects needs to be calculated. This is done for a number of n steps and a given step size Δt.
This assignment is based on work done in previous assignments (up to assignment 5). Different to the last assignment is the usage of OpenMP instead of PThreads for the implementation of multi threading. Similar to the previous assignments is the application of the Barnes-Hut method instead of the O(N2) algorithm, to decrease the computational complexity.

2 Solution

The program to solve the N-body problem is very similar to the code used to solve the previous assignments. It is run given the following parameters:

./galsim N filename nsteps delta_t theta_max graphics threads
[.5cm] **N** - Number of stars/ particles to simulate
filename - of the to read the initial positions, masses and velocity's from
nsteps - for the number of timesteps
delta_t - for the timestep size Δt
theta_max - is the threshold value used for the Barnes-Hut method
graphics - as 1 or 0 for wether graphics should be displayed
nthreads - is the number of threads used by OpenMP

Firstly the initial positions, masses and velocity's are read from the input file and checked for validity using the following method:

```
int read_doubles_from_file(int n, double *p, const char
    *fileName){}
```

In a next step the quadtree is initialized. Iterating over all given particles each one gets inserted into the quadtree using the insert method:

```
void insert(treenode **node, double x, double y, double m)
```

This method determines whether an empty leaf needed to insert the particle is already available and splits the respective leaf if it is not yet the case, using:

```
void split(treenode **node)
```

The program then starts an iteration process over the number of the given **nsteps**. In each of the timesteps,this process of creating the quadtree is executed again in order to determine the new boxes in the quadtree after the positions of the particles have been adjusted after each timestep.
Inside the iteration process over the **nsteps**, the new position and velocity for each of the **N** particles has to be calculated. This is done by creating **n_threads** and calculating these values for **n_threads** different particles in parallel, joining the **n_threads** threads before starting the next **n_threads** threads.
In order to hand over all necessary values to the calculation() method when creating the pthreads, a **struct** is used. These information are unpacked in the calculation() method and stored into variables.
Following this the forces on the particle are calculated using the cal_force method:

```
void cal_force(double x, double y, double theta_max, treenode
   *node, double* sum_x, double* sum_y)
```

which iteratively calculates and updates the parameters **sum_x** and **sum_y**, storing the forces on the particle in x and y direction.
After returning into the cal_force method, these are then used to calculate the updated position and velocity of the particle. After successfully iterating through **nsteps**, the final positions and velocitys of the particles are written to an output file and all memory allocated to the quadtree gets freed:

```
void free_tree(treenode *node)
```

3 Optimization

3.1 Serial optimization

– Code compilation using the -03 flag
– Const keywords
– Loop unrolling

These measures for serial optimization had already been implemented in the previous assignments and proven their effectiveness and therefore been kept for this assignment.
The table below shows the performance results for the compilation with the different optimization flags, showing the best performance for using the -O3 flag.

Table 1: Program run time for different optimization flags

Flag	**Time [s]**
None	2.498
-O1	1.604
-O2	1.486
-O3	1.410
-Ofast	1.571

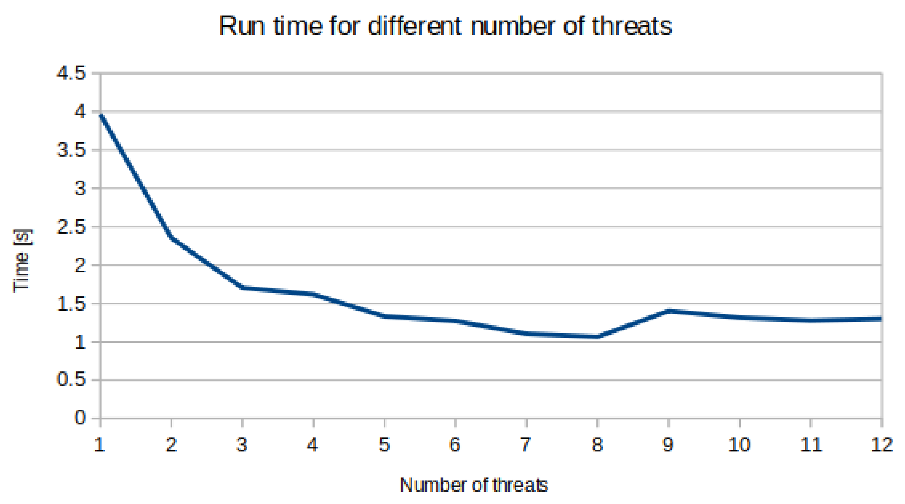
N=5000, n_steps = 100, theta_max = 0.25
AMD Ryzen 5 PRO 2500U w @ 2.0 GHz, Windows Subsystem for Linux, Ubuntu 18.04.1 LTS, gcc 7.3.0

3.2 Parallelization

To identify parallelization potential we conducted an analysis of our code from assignment 4 using gprof. This clearly showed that 97% of the program run time was spend in the **calc** method which calculated the forces on the particles and updated their position and velocity. To improve this we implemented OpenMP parallelization of the **for loop** iterating through all particles to calculate the resulting forces and positions for each of the **n_steps** timesteps:

```
#pragma omp parallel for
```

The following run times could be observed for N=5000, nsteps=100, delta_t=1e-5 and theta_max=0.25.



Running on an AMD Ryzen 5 PRO 2500U w @ 2.0 GHz, Windows Subsystem for Linux, Ubuntu 18.04.1 LTS compiling using gcc 7.3.0 with -O3 flag.

An improving run time can be observed with an increasing number of threads, up to the 8 threads which the maximum number of threads the processor can handle. From which point on the run time starts to decrease again.

To ensure the accuracy of our results and avoid different threads overwriting a shared variable we used ***private*** for **sum_x** and **sum_y**, thus creating private copies of these variables for each thread, which get joined back together.

```
#pragma omp parallel for private(sum_x, sum_y)
```

Furthermore used schedule to divide the associated loops into subsets.

```
#pragma omp parallel for private(sum_x, sum_y) schedule(dynamic)
```

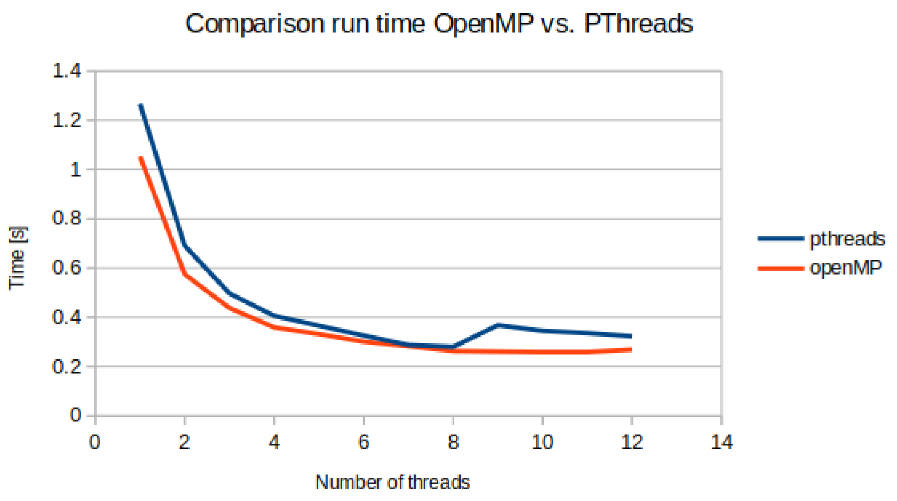
In order to determine the best schedule for our use case we tested the run time for the different schedules and found the dynamic schedule to show the best performance, as can be seen in the table below.

Table 2: Schedule comparison

Type	**Time [s]**
Static	3.073
Dynamic	2.629
Guided	2.841

N=10000, n_steps = 100, theta_max = 0.25, n_threads = 8
AMD Ryzen 5 PRO 2500U w @ 2.0 GHz, Windows Subsystem for Linux, Ubuntu 18.04.1 LTS, gcc 7.3.0 with -O3 flag

When comparing the speed ups achieved using OpenMP vs. PThreads the following graph the speed ups are pretty close to each other, with the OpenMP parallelization performing slightly better.



N=10000, n_steps = 100, theta_max = 0.25
AMD Ryzen 5 PRO 2500U w @ 2.0 GHz, Windows Subsystem for Linux, Ubuntu 18.04.1 LTS, gcc 7.3.0 with -O3 flag

Table 3: OpenMP vs. PThreads

# **Threads**	**OpenMp**	**PThreads**
	Time[s]	*Time[s]*
1	1.054	1.267
2	0.575	0.691
3	0.438	0.497
4	0.359	0.406
5	0.332	0.366
6	0.301	0.326
7	0.283	0.288
8	0.263	0.280
9	0.261	0.368
10	0.259	0.345
11	0.259	0.336
12	0.269	0.323

4 Division of labor

The work for the assignment was split up, wherein Jiayi focused on coding and debugging and Dominik on the report and measurements.