UNIVERSITY OF AMSTERDAM

PERFORMANCE ENGINEERING

Two-Dimensional N-Body Simulation

June 7, 2022

Students:
Dymitr Lubczyk
1356869

André Palheiros da Silva
14077744

Foivos Papapanagiotakis-Bousy
14078481

Tim van Kemenade
13864416

Group:
Group 19

Lecturer:
Dr. Ana L. Varbanescu

Course:
Performance Engineering

Course code:
5284PEEN6Y

1 Introduction

The N-Body problem dates back to the 17th century, when mathematicians and physicists such as Isaac Newton attempted to predict the behaviour of individual celestial objects under the influence of gravitation forces from each other. Nowadays, the N-Body problem can be computed and simulated with ease, changing the paradigm that now focuses on improving the performance of the simulation using techniques such as multi-threading.

1.1 The Barnes-Hut approximation

Simple N-Body simulation requires computing forces between all bodies, resulting in an asymptotic time complexity of $O(n^2)$. The Barnes-Hut approximation is an algorithm which uses the premise that distant body forces can be grouped, reducing the time complexity to $O(n \cdot logn)$. We intend to use this approximation to accelerate the simulation.

In a two-dimensional simulation, the algorithm [1] splits the two-dimensional plane into four equally sized cells. Every cell that contains more than one body continues to be split recursively in the same manner until every cell contains one or zero bodies. The results are stored in a quadtree. Below is an illustration of a plane and the quadtree it generates.

Once the quadtree is generated, the center of mass of each cell containing bodies is calculated. Then, for each body, the tree is traversed to calculate the forces acting upon them. A subtree is not further explored and the force is calculated with its root, if the root of the subtree is a leaf or the following condition is satisfied:

$$\frac{diag}{dist} < \theta$$

On this expression, *diag* is the diagonal of the area defined by the subtree, while *dist* is the distance between the center of the mass of the area and a body. The constant θ represents the precision of the algorithm, meaning that the higher the θ, the less precise the algorithm is.

The rationale behind the approximation criteria is the following: far away clusters of bodies can be treated as a single body represented by their center of mass. For this reason, the diagonal of the area of a cluster is used as a measure of bodies' proximity. The average distance of uniformly distributed points in a rectangle is equal to $\frac{\sqrt{2} \cdot diag}{3}$, which makes the diagonal a good unit of measurement.

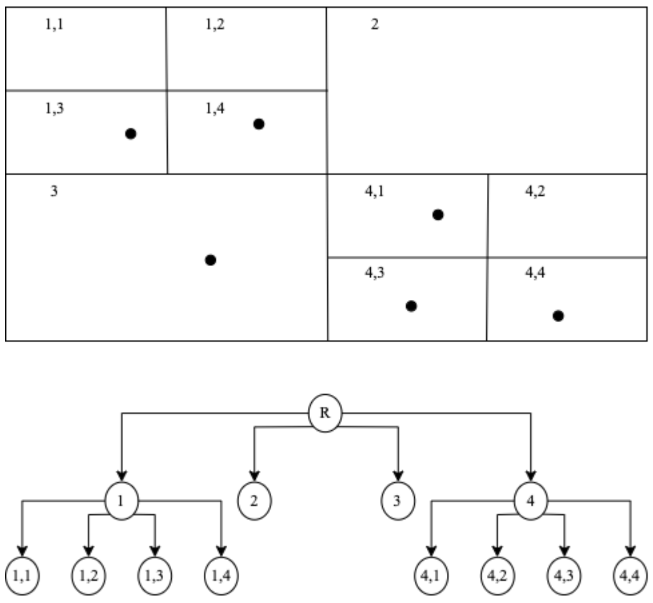


Figure 1: Example universe and corresponding quadtree

1.2 Application description

All applications built for this project will be a two-dimensional N-body simulation, which as input receives a data set containing several bodies. Each body takes as a parameter its mass, position (x, y) and velocity (v_x, v_y). The expected output of the applications will be the updated positions and velocities of the bodies. Additionally, as input, the simulations require a number representing the duration of one iteration, the number of total iterations and θ, which is only used in the Barnes-Hut algorithm. Lastly, all versions are implemented in C++17 and they only use the standard library. All scripts and prototypes along with usage instructions can be found here: `https://github.com/PhivPap/Performance-Engineering/tree/master/Project`

1.3 Experimental setup

1.3.1 Software

All *C++* versions of the n-body solutions are compiled with the *g++ (GCC)* compiler version 6.3.0 using the flags: *-O3 -std=c++17*. The tables below provide the values (or range of values) of all input configurations utilized in this project. It is noteworthy that all body properties are generated in a uniform manner using a generator script.

Universe configuration	
Bodies (32-bit unsigned int)	[1e2, 1e4]
Mass [kg] (64-bit float)	[1e10, 1e40]
Velocity-X [m] (64-bit float)	[−1e6, 1e6]
Velocity-Y [m] (64-bit float)	[−1e6, 1e6]
Position-X [m/s] (64-bit float)	[−5e16, 5e16]
Position-Y [m/s] (64-bit float)	[−5e16, 5e16]

Table 1: Universe configuration properties

Simulation configuration	
Iterations (32-bit unsigned int)	50
Iteration Duration [s] (64-bit float)	3600
Theta (Barnes-Hut only) (64-bit float)	0.5

Table 2: Simulation properties

Regarding the Barnes-Hut approximation, the size of the universe (which is used by the recursive subdivision) will be scaled according to the position of the furthest bodies, guaranteeing that no bodies will ever cross the boundaries of the universe dimension itself. Additionally, all simulations require a constant to establish the gravitational effects between the bodies. For this project, the Newtonian constant of gravitation will be used to compute the attractive forces between the bodies.

1.3.2 Approximation factor theta

To decide on the right value of θ we ran a few experiments to see the errors that we were getting, the error is defined as:

$$error = \frac{1}{n}\sum_{i=1}^{n}\frac{||ref_i - approx_i||_2}{||ref_i||_2}$$

Where ref_i is the position of the i-th body as computed by the reference implementation, while $approx_i$ is the position of the same body as calculated by the Barnes-Hut implementation. The variable n stands for the number of bodies in the simulation. The following is a table of errors for a 10000 body setup measured after 50 iterations:

Theta	**Error**
0.01	0%
0.05	0.06%
0.1	0.13%
0.5	1.05%
1	3.94%
1.5	11.74%

Table 3: Simulation error for various θ values

From now on, theta is set to 0.5 as it produces an error of 1.05%, which is deemed acceptable.

1.3.3 Hardware

All prototypes built for this project will be benchmarked on one node of the Distributed ASCI Supercomputer 5 (DAS-5) [2]. A node of the DAS-5 is a cache coherent Non Uniform Memory Access (ccNUMA) system, which consists of two Intel® Xeon® E5-2630 v3 processors. The specifications of this processor can be seen in Table 4 as produced by values gathered from the Agner Fog tables [3] and Intel's specification [4].

Intel® Xeon® E5-2630 v3 Processor	
Physical cores	8
Logical cores	16
Base frequency	2.40 GHz
Turbo frequency	3.20 GHz
Size L1 cache	32 KB
Size L2 cache	256 KB
Size L3 cache	20 MB
Latency L1 cache	4 cycles
Latency L2 cache	12 cycles
Latency L3 cache	34 cycles
Cache line size	64 Bytes

Table 4: DAS5 processor properties

The 64 GB (DDR3) main memory of the DAS-5 nodes was estimated to have a latency of approximately 100 cycles.

1.4 Performance requirements

Our goal in terms of performance is to speedup the naïve simulation algorithm using a Barnes-Hut approximation, which should not exceed an overall error of 1% (1.3.2). For 10,000 particles or more, we expect a speedup of at least 20 for 4 threads as compared to the naive O(n^2) implementation.

1.5 Approach

We start with a reference implementation that is simple and accurate, but suboptimal. Since our goal is to model and optimize a performant implementation of the Barnes-Hut approximation, we only use this version to verify the validity of our results and calculate the error produced by the approximation.

From that point on we will develop a version which uses the Barnes-Hut approximation method, which we intend to analyse, model and optimize. Once we implement improvements, we repeat the cycle until satisfied. At the end, we will reflect on the progress made and propose future work on the subject.

It is important to note that the aforementioned time complexity $n \cdot log(n)$ of the Barnes-Hut algorithm is an approximation, as the computational load not only depends on input size, but also on the input itself. Therefore, the execution time predictions will necessarily have a margin of error which will be explored in the *validation* subsection of each prototype.

From this point onwards, all estimations and measurements refer to an entire iteration of the algorithm unless stated otherwise. Input and output is not measured or modelled. Lastly, all multi-threaded models will allow for a thread count in the range [2, 8] as it is not the goal of this project to model hyperthreading or the communication between the two sockets of a DAS-5 node.

2 Prototypes

This section is dedicated to presenting all prototypes, detailing their implementation, models and results. In addition to the three Barnes-Hut prototypes found below, *Prototype 0* (a naïve O(n^2) N-Body solution) was first implemented to measure the errors and speedups of the approximations.

2.1 Prototype 1: Naïve Barnes-Hut

This prototype consists of an unoptimized implementation of the Barnes-Hut algorithm. The correctness of the program was verified by comparing its results to the ones produced by the naïve n-body solution. This prototype was modelled to determine which segments of the program should be optimized.

2.1.1 Implementation

In this section, the main kernel of the prototype is shown in pseudo-code along with some of its routines and subroutines. Routines *update_positions* and *get_area* are not detailed, as they only consist of a basic iteration over the list of bodies and represent less than 1% of execution time for a number of bodies higher than 5.

Algorithm 1 Barnes-Hut Algorithm

```
Require: bodies is an array of N particles, iter_duration is the simulated duration of one iteration and root is the root of the quad tree which necessarily hosts all the bodies in the simulated universe.
Ensure: The position and velocity attributes of all particles in bodies are updated to the correct values.
1: for $i \leftarrow 0$ to iterations do
2:     update_positions(bodies, N, iter_duration)
3:     root.area ← get_area(bodies, N)
4:     root.generate_quadtree()
5:     update_velocities(root, bodies, N, iter_duration)
6: end for
```

Algorithm 2 Routine: ”generate_quadtree”

```
Require: $quad.area$ and $quad.bodies$ are already set.
Ensure: The quad tree is generated such as no leaf of the tree has more than one body.
 1: if $quad.bodies.count <= 1$ then
 2:     return
 3: end if
 4: $center \leftarrow quad.area.center()$
 5: for $body$ in $quad.bodies$ do
 6:     if $body.pos_x < center_x$ & $body.pos_y < center_y$ then
 7:         $quad.top\_left\_quad.insert(body)$
 8:     else if $body.pos_x >= center_x$ & $body.pos_y < center_y$ then
 9:         $quad.top\_right\_quad.insert(body)$
10:     else if $body.pos_x < center_x$ & $body.pos_y >= center_y$ then
11:         $quad.bot\_left\_quad.insert(body)$
12:     else
13:         $quad.bot\_right\_quad.insert(body)$
14:     end if
15: end for
16: $quad.top\_left\_quad.generate\_quadtree()$
17: $quad.top\_right\_quad.generate\_quadtree()$
18: $quad.bot\_left\_quad.generate\_quadtree()$
19: $quad.bot\_right\_quad.generate\_quadtree()$
20: $quad.center\_of\_mass \leftarrow calc\_center\_of\_mass()$
```

Algorithm 3 Routine: ”update_velocities”

```
Require: $bodies$ is an array of $N$ particles, $iter\_duration$ is the simulated duration of one iteration and $root$ is the root of the quad tree which necessarily hosts all the bodies in the simulated universe.
Ensure: All particles in the array $bodies$ have the correct updated velocities.
 1: for $body$ in $quad.bodies$ do
 2:     $F_x, F_y \leftarrow compute\_body\_forces(quadtree, body)$
 3:     $body.vel_x \leftarrow body.vel_x + ((F_x/body.mass) \cdot iter\_duration)$
 4:     $body.vel_y \leftarrow body.vel_y + ((F_y/body.mass) \cdot iter\_duration)$
 5: end for
```

Algorithm 4 Subroutine: ”compute_body_forces”

```
Require: The quadtree is complete.
Ensure: Once the recursion concludes, $F_x$ and $F_y$ have the correct approximated values.
 1: if $quad.contained\_bodies = 0$ then
 2:     $F_x, F_y \leftarrow 0, 0$
 3: else if $quad.contained\_bodies = 1$ then
 4:     $F_x, F_y \leftarrow body\_to\_body\_attraction(body, quad.get\_single\_body())$
 5: end if
 6: if $quad.diag\_length/distance(quad, body) < \theta)$ then
 7:     $F_x, F_y \leftarrow body\_to\_quad\_attraction(body, quad)$
 8: else
 9:     $F_x, F_y \leftarrow$
10:         $compute\_body\_forces(quad.top\_left\_quad, body)$ +
11:         $compute\_body\_forces(quad.top\_right\_quad, body)$ +
12:         $compute\_body\_forces(quad.bot\_left\_quad, body)$ +
13:         $compute\_body\_forces(quad.quad.bot\_right, body)$
14: end if
```

2.1.2 Model

To model this prototype, we divide the kernel of the application into three main segments:

- Position update & Universe area computation (Algorithm 1, lines 2 & 3)
- Quadtree generation (Algorithm 1, line 4)
- Velocity update (Algorithm 1, line 5)

To put it in symbolic form, the following formula represents the execution time of the main kernel:

$$T(n) = T_{pu}(n) + T_{tg}(n) + T_{vu}(n)$$
$$T_{pu}(n) = n \cdot t_{pu}$$
$$T_{tg}(n) = n \cdot \Big(Dp(n) \cdot t_{tl} + \frac{4(t_q + t_{rc})}{3}\Big)$$
$$T_{vu}(n) = n \cdot \Big(Fc(n) \cdot t_{fc} + Nv(n) \cdot t_{rc} + (1 - Lvf(n)) \cdot Nv(n) \cdot t_{dc}\Big)$$

Starting from the top, there is the T_{pu} term, which defines the time to update body positions and to compute the area of the universe. The function responsible for that executes the same set of operations for every body, so t_{pu} is modelled to be constant.

Continuing, the term T_{tg} describes the tree generation process. It consists of the two components. The first component, $n \cdot Dp(n) \cdot t_{tl}$, describes the time spent on inserting bodies in the appropriate quads. The n factor is a result of the fact that at each level of the tree, there are approximately n bodies, the $Dp(n)$ factor approximates the tree depth for a given n and t_{tl} is the required time to insert one body in the right quad. The second component of T_{tg} comprises two following factors, $\frac{4n}{3}$ which is the number of nodes in a balanced quad tree and $t_{rc} + t_q$ which is the time required to perform a recursive function call, plus the time to create a quad object.

The last term, T_{vu}, stands for the velocities' calculation which includes the tree traversal and, in terms of computation, is the main part of the algorithm. $Fc(n)$ expresses the average number of times the algorithm performs a force calculation per body, t_{fc} is a constant time needed for one force calculation. $Nv(n)$ is the average number of visits in the quadtree per body, and t_{rc} is the (constant) cost of a recursive function call. Finally, $Lvf(n)$ is an approximation of a fraction of leaves visits in all the visits, so $1 - Lvf(n)$ is a fraction non-leaf visits. It is multiplied by average number of visits and a constant cost of distance calculation, which is performed only for non-leaf nodes.

Symbol	**Description**
t_{pu}	Time required to update the position of a single body
t_{tl}	Time required to insert a body in a quads
t_q	Time required to generate an empty quad
t_{rc}	Time required to call a function
t_{fc}	Time required to calculate a force acting on a body
t_{dc}	Time required to calculate the Euclidean distance between any two points
$Dp(n)$	Estimator of average tree depth
$Fc(n)$	Estimator of number of force calculations per body
$Nv(n)$	Estimator of number of quadtree node visits per body
$Lvf(n)$	Estimator of ratio: leaf_visits / total_visits

Table 5: Prototype 1: Constants and estimators - description

2.1.3 Calibration

The following section presents the values used to calibrate the model, as well as an explanation of how they were obtained.

Symbol	**Value**
t_{pu}	5ns
t_{tl}	29ns
t_q	60ns
t_{rc}	0.6ns
t_{fc}	55ns
t_{dc}	34ns
$Dp(n)$	$1.64 \cdot \ln(n) - 0.91$
$Fc(n)$	$45 \cdot \ln(n) - 160$
$Nv(n)$	$65 \cdot \ln(n) - 197$
$Lvf(n)$	$-0.0879 \cdot \ln(n) + 1.13$

Table 6: Prototype 1: Constants and estimators - values

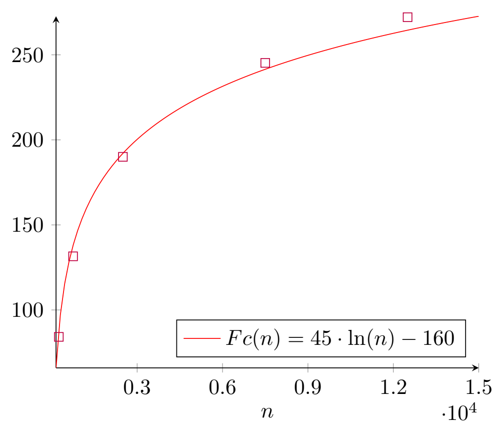
To obtain values above, the chrono high resolution clock (provided by the standard library of C++) was used. All the constant values apart from t_{rc} were measured as averages of execution times of highly isolated parts of code. Variances of those measurements are insignificant and can be ignored. To measure t_{rc}, a custom benchmark was set up. The benchmark measured balanced quadtree traversal up to a given depth limit. Arguments of the same datatype as in the Barnes-Hut implementation were passed to the function calls to closely imitate its behaviour. In the end, the total execution time was divided by the number of visited nodes, which is equal to the number of recursive function calls.

Finally, to approximate the estimator functions, training sets consisting of 5 records were created for each estimator. These estimator functions were produced using *Google Sheets* tools. Logarithmic formulas are used for estimator functions, as they capture the problem's nature in the most accurate way. The resulting formula that predicts execution time in nanoseconds given number of bodies n is:

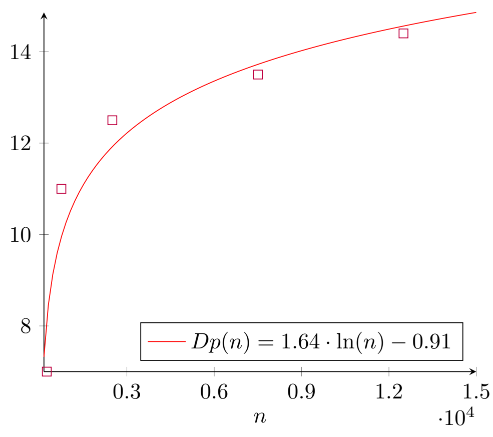
$$T_1(n) = n \cdot (194.25 \cdot \ln^2(n) + 1,685.50 \cdot \ln(n) - 7,988.05)$$

2.1.4 Validation

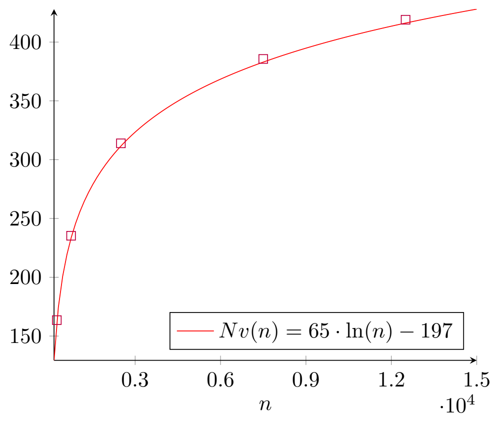
To verify the accuracy of the approximation functions, namely Fc, Dp, Nv and Lvf, a test set T consisting of 5 configurations was created. The functions are presented as graphs below with their samples from the test set and their Root-Mean-Square Error (RMSE).



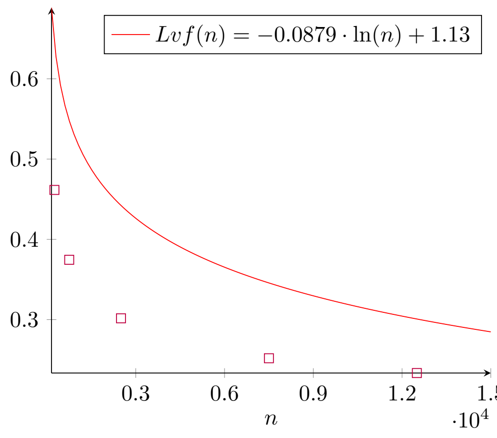
(a) RMSE(Fc, T) = 9.08



(b) RMSE(Dp, T) = 0.755



(a) RMSE(Nv, T) = 4.78



(b) RMSE(Nvf, T) = 0.127

Based on the RMSEs presented, it was deemed that these functions are acceptable. More importantly, the same approximations can be used in next prototypes, as they are not optimization dependent.

The tables below show predictions and measured execution times of each sub-kernel.

N	**Measured[s]**	**Predicted[s]**	**Error**
10000	0.0000252	0.00005	98.41%
5000	0.0000119	0.000025	110.08%
1000	0.00000313	0.000005	59.74%
500	0.00000172	0.0000025	45.35%
100	0.00000049	0.0000005	2.04%

Table 7: T_{pu} - Position update measurements

N	**Measured[s]**	**Predicted[s]**	**Error**
10000	0.00528988	0.0049007	7.36%
5000	0.0024571	0.0022865	6.94%
1000	0.000478777	0.00038123	20.37%
500	0.000240874	0.00017423	27.67%
100	0.0000505745	0.00002724	46.14%

Table 8: T_{tg} - Tree generation measurements

N	**Measured[s]**	**Predicted[s]**	**Error**
10000	0.294261	0.23543818	19.99%
5000	0.110804	0.10010936	9.65%
1000	0.0135631	0.01256517	7.36%
500	0.0052108	0.00483213	7.27%
100	0.0002839	0.00036496	28.55%

Table 9: T_{vu} - Velocity update measurements

N	Measured[s]	Predicted[s]	Error
10000	0.2995760	0.2403889	19.76%
5000	0.1132730	0.1024209	9.58%
1000	0.01404501	0.0129514	7.79%
500	0.00545305	0.0050089	8.15%
100	0.00033501	0.0003927	17.22%

Table 10: T_1 - Total execution time

N	T_{pu}	T_{tg}	T_{vu}
10000	0.01%	1.77%	98.23%
5000	0.01%	2.17%	97.82%
1000	0.02%	3.41%	96.57%
500	0.03%	4.42%	95.55%
100	0.15%	15.10%	84.76%

Table 11: Fraction of the execution time of each component

Based on the values in the tables, the following conclusions can be made; Firstly, the created model satisfies our expectations in terms of accuracy, so it can be used to find places of potential improvement. Secondly, we should start with an optimization of the velocity update component as it takes the biggest part of the execution time, and it is relatively easy to parallelize.

2.2 Prototype 2: Parallel velocity computation

This prototype consists of an improved implementation of the Barnes-Hut algorithm. Table 11 shows that the majority of execution time is spent on calculating the new velocities of the particles. Therefore, we chose to optimize that part of the program. The improvements of this decision are shown in Section 2.2.2.

2.2.1 Implementation

To optimize velocity computation, we came up with two major optimizations. First, we avoid calculating square roots (used by the Euclidean distance formula) by transforming the formula shown in Section 1.1 to:

$$diag^2 < \theta^2 \cdot dist^2$$

As θ is constant, θ^2 is also constant and does not introduce new computation to the kernel. More importantly however, the square root calculation required by the terms *diag* and *dist* can be avoided. This transformation will speedup the program as this calculation is performed approximately O($n \cdot \ln n$) times and the previous method was very expensive (see Algorithm 4, line 6). More precisely, it is expected to slightly speedup the tree generation, and significantly the force computation.

The second optimization, which offers even better results, is using an OpenMP *parallel-for* directive with dynamic scheduling on the *update_velocities* loop (see Algorithm 3). This is possible as there are no data dependencies between iterations. Static scheduling was expected to not provide optimal behaviour, as the amount of computation required to calculate velocities varies from body to body.

To experimentally verify this hypothesis, a *retired instructions* performance counter was used. Using the *likwid-perfctr* software, we measured the imbalance of retired instructions across cores (excluding core 0 as it also performs the sequential part of the program). For a configuration of 8 threads and 10000 particles, dynamic scheduling resulted in cores 1 to 7 having a load ratio in the range: [11.0%, 11.3%]. For the same configuration, static scheduling resulted in a load ratio in the range: [10.7%, 11.6%].

2.2.2 Model

The overall idea for model is the same as for Prototype1 2.1.2. There are three components and only one of them changed, namely $T_{vu}(n,p)$. Its new formula is presented below:

$$T_{vu}(n,p) = \frac{n}{p} \cdot \Big(Fc(n) \cdot Tfc(p) + Nv(n) \cdot t_{rc} + (1 - Lvf(n)) \cdot Nv(n) \cdot Tdc(p) \Big)$$

There are three changes to this component. First, as the work is divided between p threads n got replaced by $\frac{n}{p}$. Then t_{fc} is no longer a constant, and it depends on the number of threads (overhead related to the memory access), so it is replaced with $Tfc(p)$. The same applies to the t_{dc}, $Tdc(p)$ transition. This change in performance of the force and distance calculations is a direct cause of false sharing. False sharing is a by-product of caching and it occurs when a thread edits data which is physically close to the data stored in the L1 or L2 cache of another core. Here, false sharing occurs because all bodies are stored in one array. When the velocity of one body is updated, every other thread which has that body in its cache memory will have it invalidated, which has some overhead. Some solutions which mitigate the false sharing, such as the cost-zones approach, are explored in section 5.2.

The table below describes new symbols that are introduced. The remaining symbols have the same meaning as in Table 5.

Symbol	**Description**
$Tfc(p)$	Estimator of time required to calculate a force acting on a body
$Tdc(p)$	Estimator of time required to calculate the Euclidean distance between any two points

Table 12: Prototype 2: New estimators - description

2.2.3 Calibration

The following section presents the updated values used to calibrate the new model.

Symbol	**Value**
t_{pu}	8ns
t_{tl}	68ns
t_q	105ns
t_{rc}	0.6ns
$Tfc(p)$	$2.23 \cdot p + 52.1$
$Tdc(p)$	$2.27 \cdot p + 28.5$
$Dp(n)$	$1.64 \cdot \ln(n) - 0.91$
$Fc(n)$	$45 \cdot \ln(n) - 160$
$Nv(n)$	$65 \cdot \ln(n) - 197$
$Lvf(n)$	$-0.0879 \cdot \ln(n) + 1.13$

Table 13: Prototype 2: Constants estimators - new values

The values in the table above were obtained as explained in Section 2.1.3. However, in contrast to *Prototype 1*, the caching behaviour of the program is no longer consistent in its parallel sections. As such, instead of using a constant time for the duration of force (t_{fc}) and distance (t_{dc}) computations, the model now uses the approximation functions $Tfc(p)$ and $Tdc(p)$ respectively. The resulting formula that predicts execution time in nanoseconds based on the number of bodies n and number of threads p is:

$$T_2(n,p) = n \cdot (12.96 \cdot \ln^2(n) + 153.38 \cdot \ln(n) - 211.74 + \frac{162.83 \cdot \ln^2(n) + 1649.16 \cdot \ln(n) - 7724.31}{p})$$

2.2.4 Validation

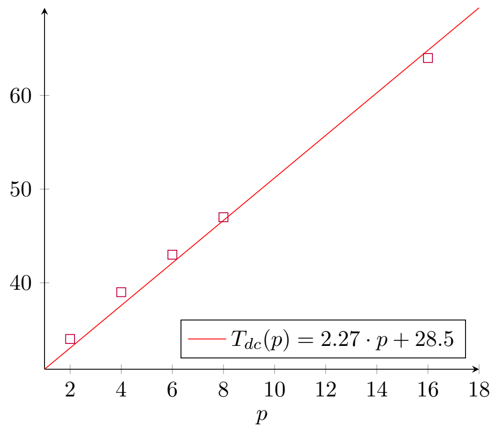
The following table demonstrates the performance gain of avoiding square root calculations. Comparisons are limited to a single thread, as optimizations are made on the operations level.

N	$t_{fc}[ns]$	$t_{fc}^{new}[ns]$	$t_{dc}[ns]$	$t_{dc}^{new}[ns]$
10000	56	50	41	28
5000	56	50	41	28
1000	55	50	40	28
500	55	50	40	28
100	55	50	40	28

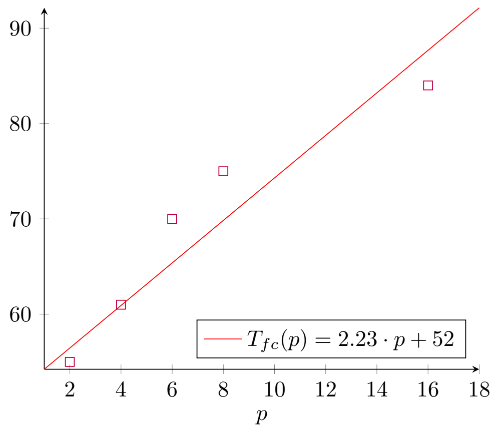
Table 14: Square root removal performance improvement

The improvement on t_{fc} is equal to roughly 10%, while on t_{dc} it is equal to 30%. These two sections are crucial for the algorithm performance, which can be seen at Section 3.

In order to validate the T_{dc} and T_{fc} estimators, the same approach as in Section 2.1.4 is taken. RMSE values are measured on a test set T and compared with the average value of that set.



(a) RMSE(T_{dc}, T) = 0.58



(b) RMSE(T_{fc}, T) = 1.5

As RMSE values are relatively small compared to values in the test sets, the estimators are examined and correct.

Values presented in the tables below are used to validate the accuracy of the model and will be used to determine the direction of further optimizations of the program in the following prototypes.

N	P	Measured[s]	Predicted[s]	Error
10000	2	0.00007632	0.00008	4.82%
10000	4	0.00007856	0.00008	1.83%
10000	8	0.00008155	0.00008	1.90%
1000	2	0.00001052	0.000008	23.93%
1000	4	0.00001063	0.000008	24.71%
1000	8	0.00001164	0.000008	31.28%
100	2	0.00000141	0.0000008	43.19%
100	4	0.00000134	0.0000008	40.50%
100	8	0.00000106	0.0000008	24.72%

Table 15: T_{pu} - Position update measurements

N	P	Measured[s]	Predicted[s]	Error
10000	2	0.01100650	0.010994	0.11%
10000	4	0.01125110	0.010994	2.28%
10000	8	0.01153170	0.010994	4.66%
1000	2	0.00111320	0.000844	24.16%
1000	4	0.00110579	0.000844	23.65%
1000	8	0.00117920	0.000844	28.41%
100	2	0.00010557	0.000059	44.21%
100	4	0.00010725	0.000059	45.08%
100	8	0.00011252	0.000059	47.65%

Table 16: T_{tg} - Tree generate measurements

N	P	Measured[s]	Predicted[s]	Error
10000	2	0.09389030	0.1186517	26.37%
10000	4	0.05471470	0.0652796	19.31%
10000	8	0.02919910	0.0385936	32.17%
1000	2	0.00632598	0.0063621	0.57%
1000	4	0.00345042	0.0034872	1.07%
1000	8	0.00190396	0.0020498	7.66%
100	2	0.00038823	0.0001862	52.05%
100	4	0.00025512	0.0001017	60.15%
100	8	0.00022964	0.0000594	74.13%

Table 17: T_{vu} - Velocity update measurements

N	P	Measured[s]	Predicted[s]	Error
10000	2	0.104973	0.129726	23.58%
10000	4	0.066044	0.076354	15.61%
10000	8	0.040812	0.049668	21.70%
1000	2	0.007450	0.007214	3.16%
1000	4	0.004567	0.004339	4.98%
1000	8	0.003095	0.002902	6.23%
100	2	0.000495	0.000246	50.35%
100	4	0.000364	0.000161	55.64%
100	8	0.000343	0.000119	65.29%

Table 18: T_2 - Total execution time

N	P	T_{pu}	T_{tg}	T_{vu}
10000	2	0.07%	10.49%	89.44%
10000	4	0.12%	17.04%	82.85%
10000	8	0.20%	28.26%	71.54%
1000	2	0.14%	14.94%	84.92%
1000	4	0.23%	24.21%	75.55%
1000	8	0.38%	38.10%	61.52%
100	2	0.28%	21.32%	78.40%
100	4	0.37%	29.49%	70.14%
100	8	0.31%	32.78%	66.91%

Table 19: Fraction of the execution time of each component

There are two main conclusions to be drawn based on data presented in the tables above. First, even though the model does not provide perfectly accurate results as it up to 65% off for small inputs, it gives good results for bigger input sizes. Taking that into consideration, we argue that the model serves its purpose and can be used in next prototypes. Additionally, the velocity update part T_{vu} is reduced from around 95% to 65% for 8 thread executions, as a near linear speedup was achieved. It was possible to conclude that the tree generation part, which now takes up to 30% is worth optimizing.

2.3 Prototype 3: Faster tree generation

This prototype builds on the previous optimizations to produce an even faster implementation. With the optimizations that were implemented for prototype 2, the fraction of the execution time of the tree generation (T_{tg}) has now grown to be at least 30% for a large number of threads (see Table 19). Consequently, we choose to pursue optimizations on the part of the program.

2.3.1 Implementation

As the tree generation is a recursive procedure, the first idea for speeding it up was to implement a task-based parallelization of the recursion. The idea was to divide the workload of the tree generation at the first or second level of the tree over multiple threads using the OpenMP task directive. This, however, proved to be slower than the sequential version of the recursion. Investigating the reasons for the slowdown, it was determined that the reason for this was that in every recursive call, four heap allocations were performed. The underlying problem with heap allocation is that it introduces unwanted synchronization, as the operating system needs to perform all memory allocations sequentially.

To avoid the high cost of multiple synchronization points, the quadtree was changed to an array implementation where all quads are allocated at once as an array with one system call. Now, every recursive call in the tree generation instead of allocating memory, requests some slots from the pre-allocated array using an atomic counter to avoid data races. While this proved to be faster than the previous sequential implementation, the speedup was not due to parallelism, as running this version sequentially was quicker. The issue with parallelism turned out to be a load balancing problem, as the quadtree was significantly imbalanced (depth is usually 3 to 4 times more than a perfectly balanced tree). As a consequence, parallelism as an optimization for this part was discarded, with the bulk quad initialization being kept as it provides significant speedup by reducing the total time taken by heap allocations. Additionally, the bulk quad initialization provided a decent speedup to the "velocities update" part of the program, as the tree traversal became faster when the tree is stored as an array (due to lower cache miss ratio). Lastly, it should be noted that it was estimated that a quadtree pre-allocated on an array of size $4 \cdot n$, (where n is the number of particles in the simulation), was sufficiently large to host the trees generated by the datasets used.

Another optimization implemented for this prototype is replacing *std::list* with *std::vector* (both implementations provided by the standard library of C++) which is used by each quad to determine in which of the four subtrees its bodies should be placed in (see Algorithm 2, lines 5-15). This decision was based on the following two observations; List insertion is expensive as it always requires heap allocation, while list traversal is inefficient as it is not "cache friendly". Changing from *std::list* to *std::vector* sped up the insertion time by a factor of approximately 3. Pre-allocating the vector with some initial size decreases the insertion time, but increases the allocation time of the quads (as these vectors are fields of the quad class). Benchmarking for 10,000 bodies, it was estimated that a good initial size is 40. With this pre-allocation, the average time to insert a body to the vector is more than 6 times faster than the previous list insertion.

The last minor optimization added was the merge of the tree generation recursion with the recursive center of mass calculation, which was previously a complete separate procedure. This reduces the tree traversals of the tree generation to only a single one.

2.3.2 Model

The idea behind the model remains the same apart from the tree generation part. In the tree generation formula, factor $\frac{4}{3}$ has been replaced with 4. The reasoning behind this change originates from the fact that previously it was assumed that the quadtree is balanced. However, with this new approach where we pre-allocate an array of quads, it was experimentally determined that $4 \cdot n$ is a safe upper bound for the numbers of quads in the tree.

$$T_{tg}(n) = n \cdot \Big(Dp(n) \cdot t_{tl} + 4(t_q + t_{rc})\Big)$$

The descriptions of the symbols remain the same, but their new calibration is explored in the next section.

2.3.3 Calibration

The following section present values used to calibrate *Prototype 3*.

Symbol	**Value**
t_{pu}	8ns
t_{tl}	11ns
t_q	80ns
t_{rc}	0.6ns
$Tfc(p)$	$2.16 \cdot p + 36.5$
$Tdc(p)$	$2.27 \cdot p + 28.5$
$Dp(n)$	$1.64 \cdot \ln(n) - 0.91$
$Fc(n)$	$45 \cdot \ln(n) - 160$
$Nv(n)$	$65 \cdot \ln(n) - 197$
$Lvf(n)$	$-0.0879 \cdot \ln(n) + 1.13$

Table 20: Prototype 3: Constants and estimators - new values

Comparing to the last calibration in Section 2.2.3, three values changed. Firstly, t_{tl} decreased from $68ns$ to $11ns$. The basis for that improvement is the change from *std::list* to *std::vector*, since there is no need to create a new list node every time a body is inserted into a quad. This dramatically reduces the number of system calls and consequently improves performance of an average tree insertion operation. In addition, the average time spent on the quad object generation decreased from $105ns$ to $80ns$, as multiple small system calls are combined in a large one, which decreases the communication overhead between the program and the operating system. Finally, $Tfc(p)$ decreased from $2.23 \cdot p + 52.1$ to $2.16 \cdot p + 36.5$. This is likely caused by the fact that the tree is now stored as an array, which improves the caching behaviour of quads. The new calibrated model is presented below:

$$T_3(n,p) = n \cdot (12.96 \cdot \ln^2(n) + 56.75 \cdot \ln(n) + 32.92 + \frac{162.83 \cdot \ln^2(n) + 947.16 \cdot \ln(n) - 5228.31}{p})$$

2.3.4 Validation

To validate the assumption that $4 \cdot n$ pre-allocated quads are sufficient for the tree generation part, the program was executed for a hundred iterations and the biggest tree size was collected for each n. It was then observed that the tree size is not increasing over time, but also that with the increasing problem size, a lesser fraction of quads is used. This gives us a confidence to validate the upper-bound for the problem size ranges mentioned in the Table 1.

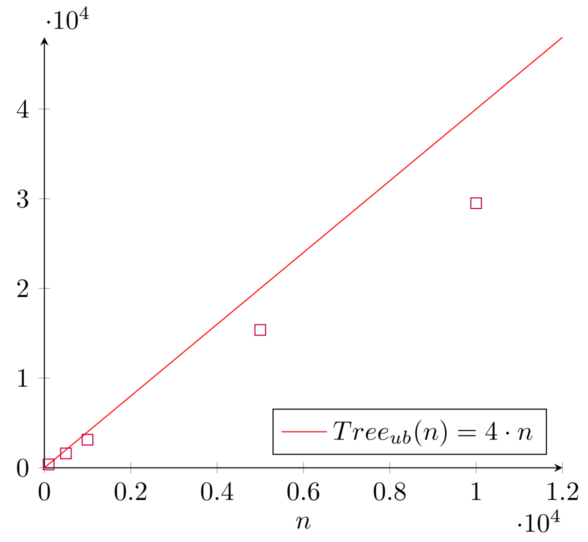


Figure 5: Tree size upper-bound compared with actual maximal size

From the tables presented below, the following conclusions can be drawn. First, the optimizations introduced by *Prototype 3* drastically decreased the time spent on the tree generation portion by a factor of approximately 2.5. Additionally, the velocity computation part (T_{vu}) was sped up from 10% to 30%.

N	P	Measured[s]	Predicted[s]	Error
10000	2	0.00008358	0.00008	4.29%
10000	4	0.00008602	0.00008	7.00%
10000	8	0.00008532	0.00008	6.24%
1000	2	0.00000966	0.000008	17.19%
1000	4	0.00001091	0.000008	26.65%
1000	8	0.00001164	0.000008	31.28%
100	2	0.00000142	0.0000008	43.68%
100	4	0.00000139	0.0000008	42.62%
100	8	0.00000148	0.0000008	46.06%

Table 21: T_{pu} - Position update measurements

N	P	Measured[s]	Predicted[s]	Error
10000	2	0.00469381	0.004775	1.74%
10000	4	0.00440949	0.004775	8.30%
10000	8	0.00441796	0.004775	8.09%
1000	2	0.00057162	0.000436	23.68%
1000	4	0.00056162	0.000436	22.32%
1000	8	0.00056459	0.000436	22.73%
100	2	0.00007122	0.000039	44.54%
100	4	0.00007530	0.000039	47.54%
100	8	0.00007163	0.000039	44.86%

Table 22: T_{tg} - Tree generation measurements

N	P	Measured[s]	Predicted[s]	Error
10000	2	0.08569450	0.0980357	14.40%
10000	4	0.04518360	0.0548822	21.46%
10000	8	0.02639690	0.0333055	26.17%
1000	2	0.00563194	0.0051367	8.79%
1000	4	0.00333478	0.0028692	13.96%
1000	8	0.00190396	0.0017355	8.85%
100	2	0.00025241	0.0001472	41.67%
100	4	0.00020719	0.0000820	60.41%
100	8	0.00017696	0.0000494	72.07%

Table 23: T_{vu} - Velocity update measurements

N	P	Tpu	Ttg	Tvu
10000	2	0.09%	5.19%	94.72%
10000	4	0.17%	8.88%	90.95%
10000	8	0.28%	14.30%	85.43%
1000	2	0.16%	9.20%	90.64%
1000	4	0.28%	14.37%	85.35%
1000	8	0.47%	22.76%	76.77%
100	2	0.44%	21.91%	77.65%
100	4	0.49%	26.52%	72.99%
100	8	0.59%	28.64%	70.76%

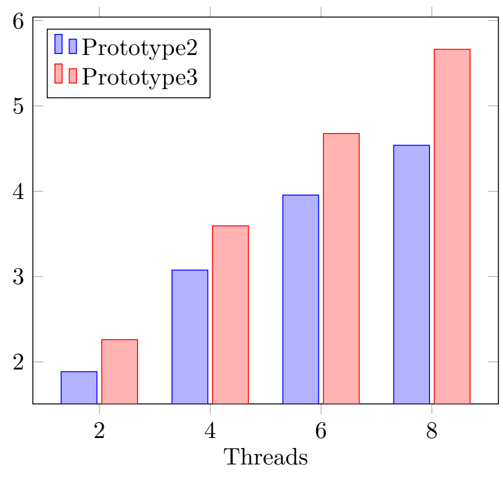
Table 24: Fraction of the execution time of each component

N	P	Measured[s]	Predicted[s]	Error
10000	2	0.090472	0.102891	13.73%
10000	4	0.049679	0.059738	20.25%
10000	8	0.030900	0.038161	23.50%
1000	2	0.006213	0.005581	10.18%
1000	4	0.003907	0.003313	15.20%
1000	8	0.002480	0.002180	12.11%
100	2	0.000325	0.000188	42.31%
100	4	0.000284	0.000122	56.91%
100	8	0.000250	0.000090	64.12%

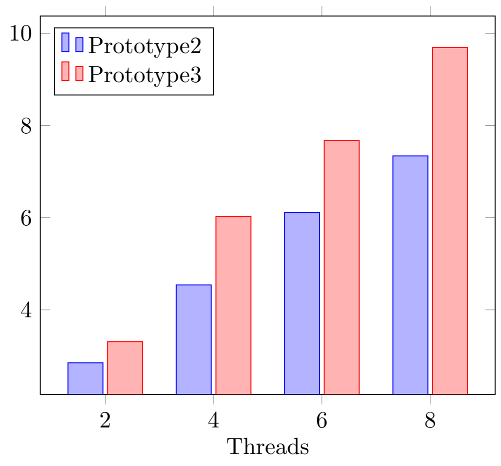
Table 25: T_3 - Total execution time

3 Speedups

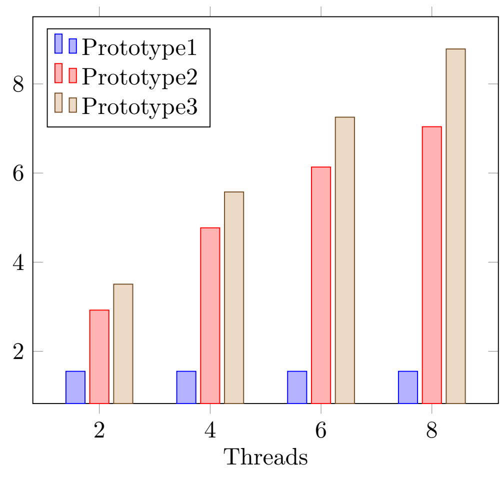
The following section presents the speedups achieved based on execution time measurements of all prototypes. The first two speedup graphs are plotted in reference to *Prototype 1* and the next two in reference to the results obtained by *Prototype 0* (the naive non Barnes-Hut version).



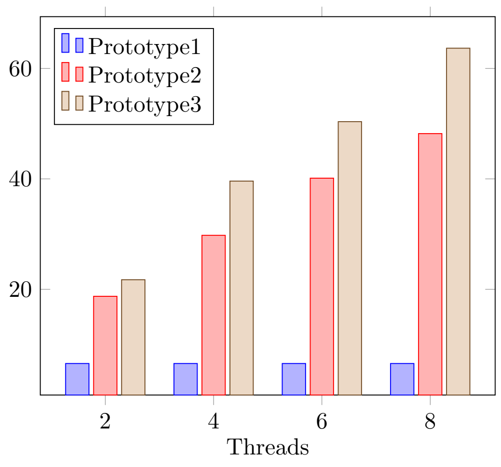
(a) 1K particles



(b) 10K particles



(a) 1K particles

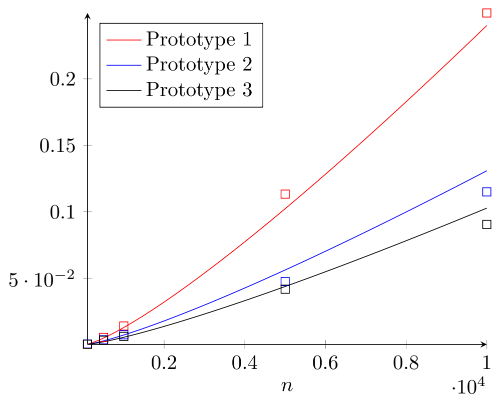


(b) 10K particles

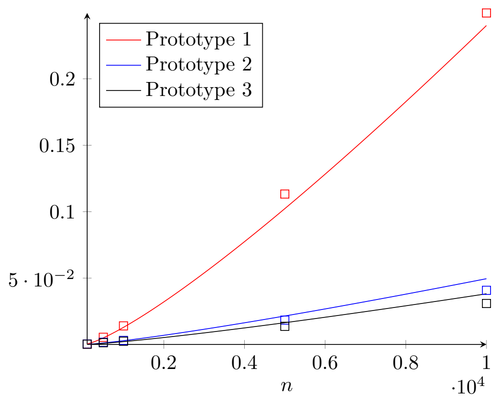
The significant difference in speedups observed on the bottom two graphs is a consequence of the higher asymptotic computational complexity of the non Barnes-Hut version (O(n^2) compared to O($n \cdot logn$)).

4 Performance Prediction

In this section, the accuracy of the models and the performance limitations of the prototypes are discussed. For the performance upper bound calculation, the calibrated formulas are used, as they are deemed fairly accurate. On the graphs below, it is possible to consult the accuracy of the models compared to actual measurements. More detailed information about the accuracy of the models over multiple configurations can be gathered from the tables found in the *validation* section of each prototype.



(a) Execution time prediction for 2 threads



(b) Execution time prediction for 8 threads

To predict maximal achievable speedup, the following function is defined:

$$f(n) = \lim_{p \to \infty} \frac{T_1(n)}{T_3(n,p)}$$

Assuming infinite number of threads, the parallelizable part of $T_3(n,p)$ tends to 0 and as a result:

$$f(n) = \frac{194.25 \cdot ln(n)^2 + 1685.50 \cdot ln(n) - 7988.05}{12.96 \cdot ln(x)^2 + 153.38 \cdot ln(x) - 211.74}$$

Note that $f(n)$ is strictly increasing for $n > 100$, so the maximal speedup S_{max} can be expressed as:

$$S_{max} = \lim_{n \to \infty} f(n) = 14.98$$

The following is a graph of $f(n)$, which is the speedup upper-bound of our best implementation as compared to *Prototype 1*.

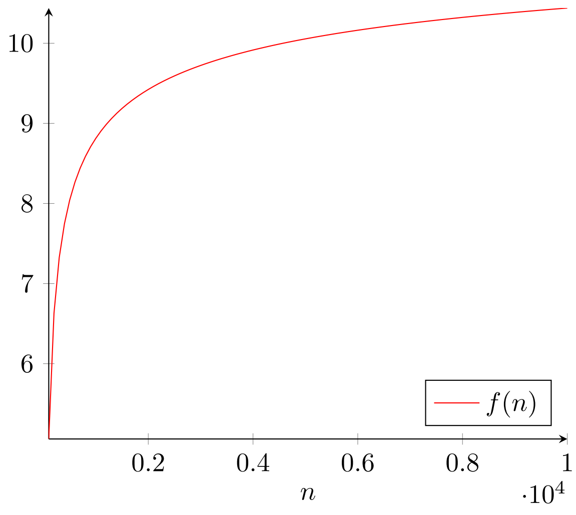


Figure 9: Theoretical speedup upper-bound

5 Conclusion & Future Work

Overall, this project shows the complexity of accurately predicting execution time of a Barnes-Hut approximation implementation. It requires extensive statistical analysis of both the algorithm and the behaviour of the program. While we are satisfied with the results reported previously, we have also received constructive criticism, which is addressed in the three sections below.

5.1 Accuracy of the predictions

In some of the models presented in this paper, it is possible to notice patterns in the measured error. For instance, T_3 (Table25) suffers from higher errors on the boundaries of its domain or T_{tg} (Table22) has an error, which is decreasing with increasing problem size. Those examples might indicate that the models are missing something, as the errors are not random. For example, an explanation for this phenomena could be different caching behaviour depending on problem size, which could not be captured efficiently by the statistical models.

5.2 False sharing

As stated in Section2.2.2, while parallelizing the velocity update loop, an important slowdown of some sections of the program was noted. Upon further investigation, it was determined that this was a result of false sharing of bodies. As one thread updates the velocity of one body, all instances of that body in cache memories of other cores will cause a cache line invalidation. This leads to a linear deterioration (based on thread count p) of the execution time of the functions where false sharing occurs.

One naïve attempt to correct this issue would be to allocate bodies with independent heap allocations instead of a single array allocation. While this will reduce false sharing, it will not provide any speedup as it reduces cache hit ratios by the same amount.

Another solution to this would be to allocate the bodies in separate arrays and have each thread operate on their own set of bodies. Despite this solution having a chance of achieving better results, it is still suboptimal for the following two reasons: Firstly, the load balance changes over the iterations of the algorithm, meaning it is not possible to correctly determine a good body distribution ahead of time. Secondly, many cache invalidations will still occur if the bodies are not distributed according to their position in the universe.

A good solution to this problem is the cost-zone method which dynamically distributes the bodies over different threads according to their position in the universe. This leads to better data locality, as the force calculation always requires nearby bodies while distant bodies can be grouped and approximated. This optimization offers much value in performance as not only it reduces false sharing, but it will also independently increase cache hit ratios.

5.3 Input topology

All models built for this project predict execution time on DAS-5 based on the number of particles in the simulation n and the number of threads p (if the application is multithreaded). However, the models can only be so accurate, as they do not take into consideration the topology of the bodies in the universe (the asymptotic computational complexity of the algorithm is not exact, it depends on both approximation factor θ and body distribution). To improve future model accuracy, it would be beneficial to investigate the correlation between performance and topology. Topology could be used as an input to the new model, using mathematical properties to define it.

Bibliography

[1] J. Heer, *The barnes-hut approximation - efficient computation of n-body forces.* [Online]. Available: https://jheer.github.io/barnes-hut/.

[2] NWO/NCF, *Das-5 overview.* [Online]. Available: https://www.cs.vu.nl/das5/.

[3] A. Fog, *Instruction tables.* [Online]. Available: https://www.agner.org/optimize/instruction_tables.pdf.

[4] Intel, *Intel® xeon® processor e5-2630 v3.* [Online]. Available: https://ark.intel.com/content/www/us/en/ark/products/83356/intel-xeon-processor-e52630-v3-20m-cache-2-40-ghz.html.