1 Introduction

1.1 Question 1.1 a)

for a number of bodies = N the cost per step is $O(N^2)$ for a standard N-body solver with Euler integration this is because of the direct pairwise nature of the force calculations which involves calculating the force each body exerts on the other for each time step. This uses two nested for loops of length N this results in $N(N-1)$ force calculations. The computational complexity remains $O(N^2)$ due to this. For my implementation this is the case as in the force calculation the largest loop is a nested for loop with N*N complexity and the collision check uses again a double nested for loop for calculation this means the predominant calculation remains $O(N^2)$. this is supported by Figure 2

1.2 Question 1.1 b)

Several optimizations can significantly enhance the efficiency and accuracy of N-body simulations. Implementing the Barnes-Hutt algorithm, which groups distant bodies as single point masses, reduces the algorithm's complexity to $O(Nlog(N))$offering substantial time savings. Additionally, the Fast Multipole Method (FMM) can further streamline computations, lowering complexity to $O(N)$ by spatially dividing the simulation area and summarizing the effects of distant bodies. To improve accuracy, particularly in scenarios where bodies are in close proximity, dynamic time stepping can be employed. This technique adjusts the time step based on the minimum distance between bodies (minD), allowing for finer precision during close encounters and larger steps when bodies are farther apart.

1.3 Question 1.1 c)

Commonly Euler method schemes have first order convergence denoted

$$O(\Delta t) = C\Delta t,$$

where C is a constant. From Figure 1 we can see that the minimum stable timestep is 1e-6

1.4 Qestions 1.1 d)

my estimated convergence it $O(\Delta t)$ and looking at Figure 4 we can see this is the case until the timestep becomes stable.

1.5 Question 2

the chosen optimization strategy is to use Barnes-Hutt Oct trees to change the time complexity to $O(NlogN)$ The key idea behind this algorithm is to group distant particles into larger clusters and approximate their cumulative gravitational forces on a given particle as if they originated from a single, more massive body located at the cluster's center of mass. This is achieved by recursively dividing the simulation into octrees each node of the tree represents a region of space and contains information about the total mass and centre of mass of all particles in the region. Unfortunaly this algorithm did not work due to a stack overflow error when trying to acces the distance array in force calculations.

1.6 Question 3

for the vectorised code I changed the originial data structure implementation of array of structures (AOS) to separate arrays as this allows for better cache utilization and allows for SIMD as data elements of the same type are stored contiguously allowing operations on them to be loaded into SIMD registers. I also used mm alloc which is a function that aligns allocations to the start of the cache line. this reduces the overhead of dealing with miss-aligned data and allows for more efficient SIMD implementation. I used "pragma omp simdto help the compiler identify places to implement SIMD instructions which greatly increased the speed of the vectorised function compared to the original. the vectorised version for N = 3375, time = 10 and timestep = 0.01 has a time of 66.7 seconds and the non vectorised has time of 156.7 seconds. showing a 230% improvement. [1]

1.7 Question 4

to enhance the code from 3 I implemented a data style called chunking. This divides data into smaller blocks or chunks before processing which improves data locality allowing the data to be found in quicker registers [2]. It also can help load balance between cores in my multiprocessing directives. I used "pragma omp parallel forto parallelise the chunking of my data. This meant that each chunk would be created on it's own thread and calculated on this thread. I used target as a way to enhance speed by sending the processes to the GPU. I used these on most for loops however I could not use them on the collision for loops as this caused cache conflicts and would alter my data creating an incorrect simulation. The chunking also allows me to not use critical regions or reductions in my force calculations as each update is happening on a separate chunk of the array I do not need to worry about write conflicts. Figure 3 shows how the number of threads available changes the time execution time. This initially increases as the overhead of distribution outweighs the performance gain of multithreading. All experiment were performed on N=3356 final time=10 and dt=0.001 on my own computer.

Referenser

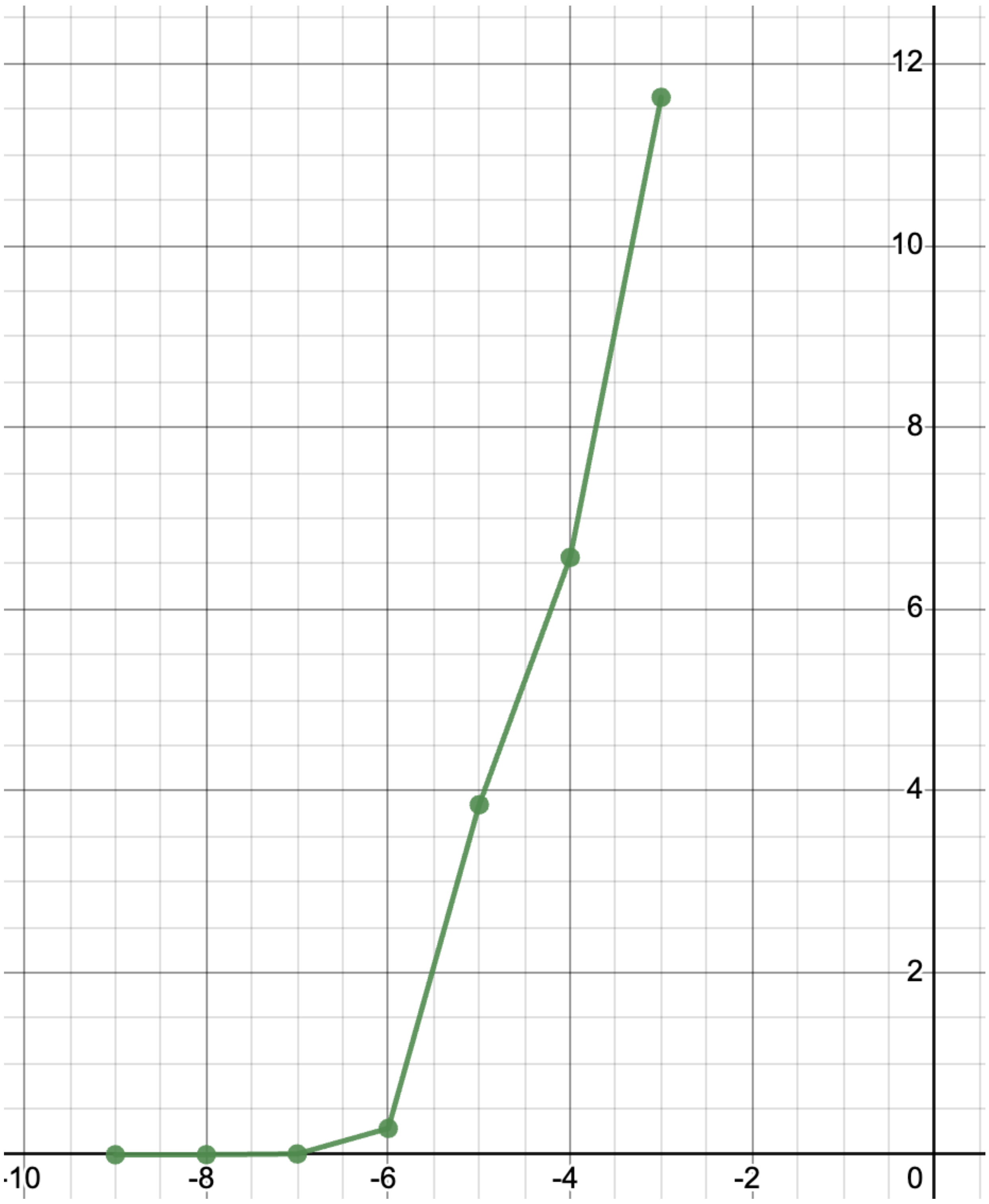
[1] F. Baru-Dev, "nbody-demo: A demonstration of an n-body simulation," https://github.com/fbaru-dev/nbody-demo/tree/master, 2023, accessed: 2024-02-12.

[2] C. Bastoul and P. Feautrier, "Improving data locality by chunking," in *Compiler Construction: 12th International Conference, CC 2003 Held as Part of the Joint European*

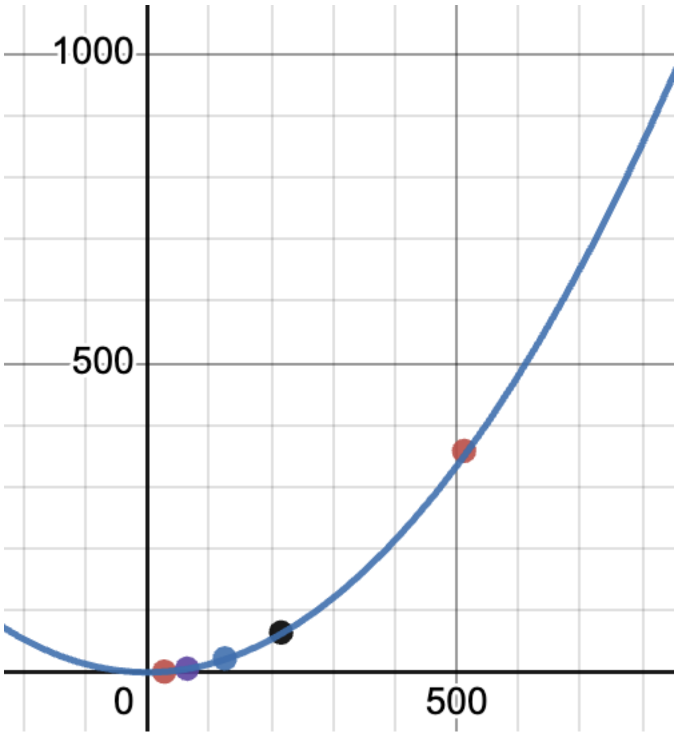
Conferences on Theory and Practice of Software, ETAPS 2003 Warsaw, Poland, April 7–11, 2003 Proceedings 12. Springer, 2003, pp. 320–334.

A appendix

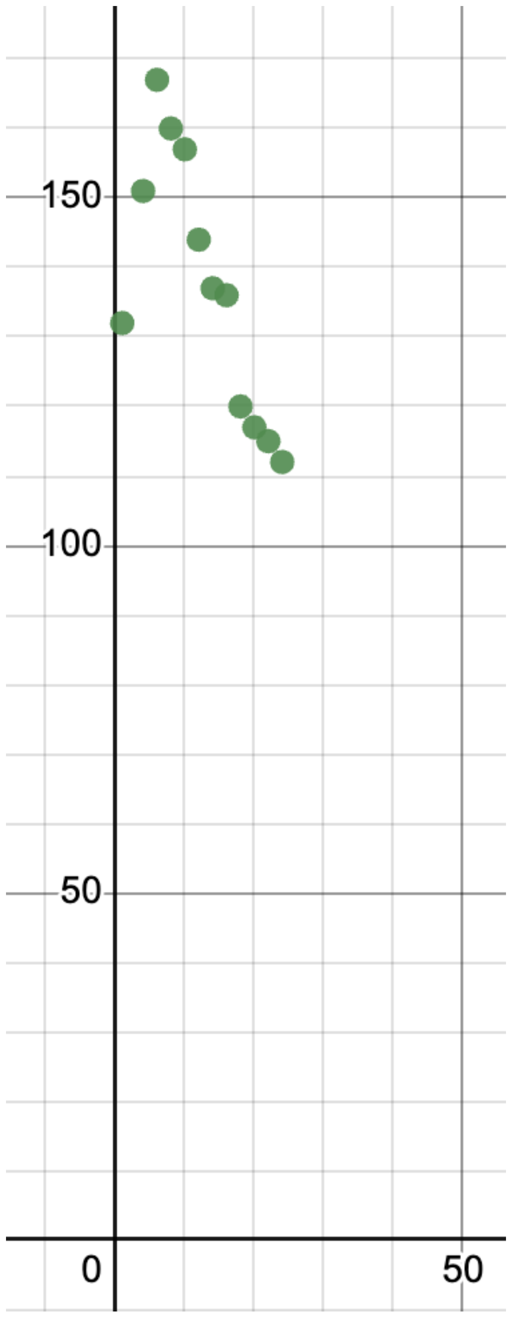
A.1



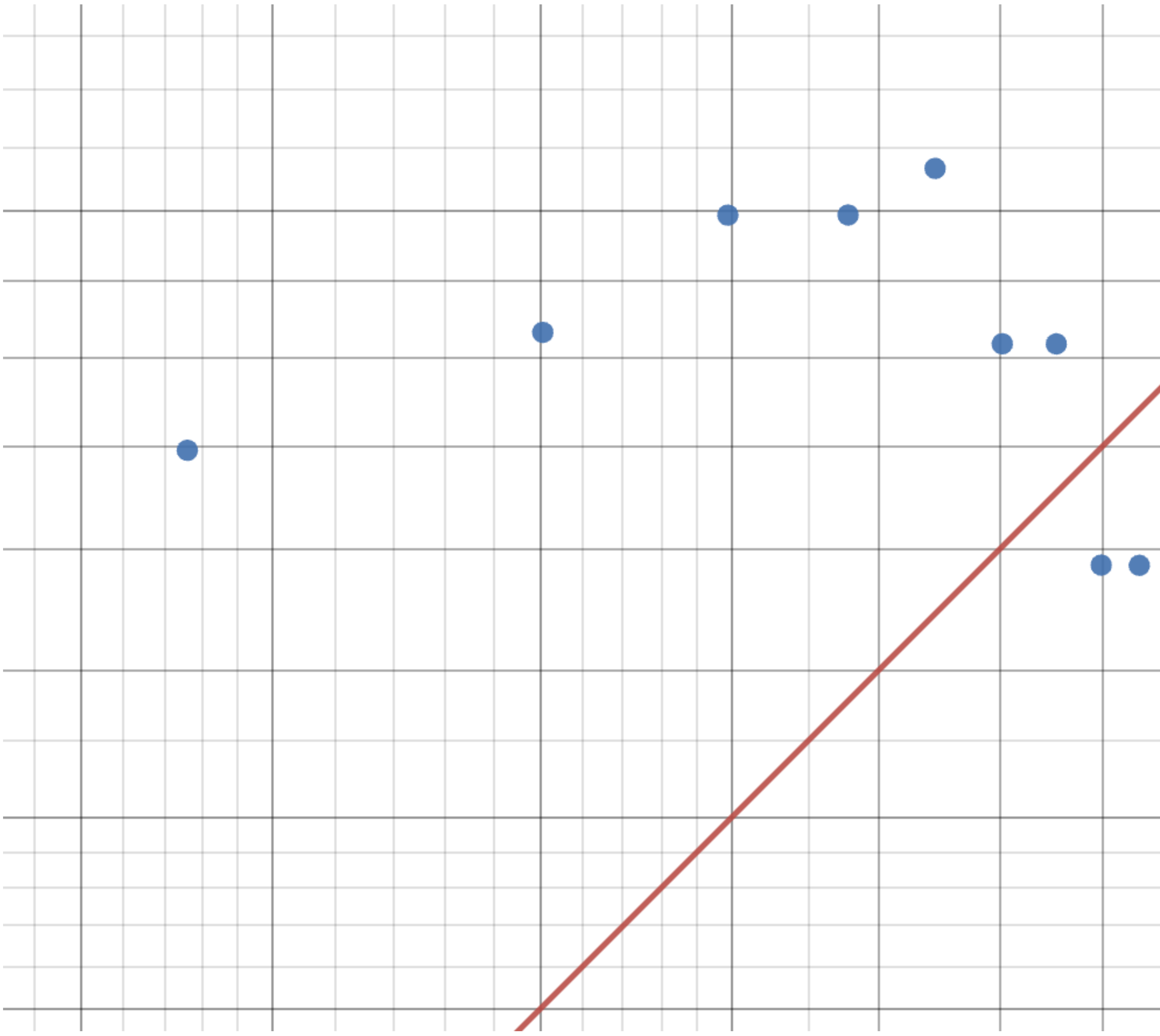
Figur 1: plot of error as time step increases



Figur 2: plot of time against Number of bodies



Figur 3: A figure that shows the time taken to execute as the number of threads increases



Figur 4: Figure showing the log(log(h)) to log(log(error))