GEORG-AUGUST-UNIVERSITÄT GÖTTINGEN IN PUBLICA COMMODA SEIT 1737

HPS

Seminar Report

# Dwarf Galaxy Simulation of N-Body Problem

Amelie Thran, Niels Jautelat

MatrNr: 18640850, 21966034

Supervisor: Patrick Höhn

Georg-August-Universität Göttingen
Institute of Computer Science

September 30, 2025

# Abstract

The movement of stars within a galaxy is a complex system in which everything interacts with everything else. This cannot be described using analytical equations; instead, it must be simulated numerically

This numerical solution consists of letting the stars move for a set period of time, before calculating the forces each star has acted upon all others, leading to poor scaling of the problem for higher number of stars.

To speed up this calculation we try a number of solutions to parallelize this problem and to use our computing resources more effectively, including summarizing multiple distant stars into sectors, or changing the size of the simulation step, depending on the distance to the center of the galaxy.

We applied these principles to spherical dwarf galaxies which were also physically accurately generated for this project, including the only recently discovered category of ultra-faint dwarf galaxies. We discovered that the distance to the center correlates incredibly well to the distance of the nearest other star, leading to virtually identical results when scaling the step size based on those metrics with considerably less compute overhead when looking at the distance to the galactic center.

Further we discovered that this is actually the most efficient way to simulate the movement of the stars up to 100,000 stars. At this point it becomes more efficient to incur the performance losses of summarizing distant stars in an area of the simulation space as a sectors, instead of looking at them individually.

We also noticed that these dwarf galaxies indeed need large amounts of dark matter to stay stable.

## Declaration on the use of ChatGPT and comparable tools in the context of examinations

In this work I have used ChatGPT or another AI as follows:

- ☐ Not at all
- ☐ During brainstorming
- ☐ When creating the outline
- ☐ To write individual passages, altogether to the extent of 0% of the entire text
- ☑ For the development of software source texts
- ☑ For optimizing or restructuring software source texts
- ☐ For proofreading or optimizing
- ☐ Further, namely: -

I hereby declare that I have stated all uses completely.
Missing or incorrect information will be considered as an attempt to cheat.

# Contents

**List of Tables** **v**

**List of Figures** **v**

**List of Listings** **v**

**List of Abbreviations** **vi**
Abbreviations and Names of our Solutions . . . . . vi

**1 Introduction** **1**

**2 Methodology** **1**
2.1 Problem statement . . . . . 1
2.2 Physical equations for generating a spherical galaxy . . . . . 1
2.2.1 Distribution of stars . . . . . 1
2.2.2 Distribution of initial masses . . . . . 2
2.2.3 Distribution of initial velocities . . . . . 3
2.3 Physical equations for simulating the movement of the stars . . . . . 3
2.3.1 Modeling two-body systems . . . . . 4
2.3.2 Modeling n-body systems . . . . . 4
2.4 Solution approaches . . . . . 4
2.4.1 Sequential solutions . . . . . 4
2.4.2 Parallel Solutions . . . . . 4
2.4.3 Validation of improvement . . . . . 6

**3 Implementation** **6**
3.1 Galaxy generation . . . . . 6
3.1.1 Distribution of stars . . . . . 6
3.1.2 Distribution of initial masses . . . . . 7
3.1.3 Distribution of initial velocities . . . . . 7
3.2 Movement simulations . . . . . 7
3.2.1 Sequential Solutions . . . . . 7
3.2.2 Parallel Solutions . . . . . 8

**4 Performance analysis / evaluation** **9**
4.1 Performance . . . . . 9
4.1.1 Performance of the sequential solution . . . . . 9
4.1.2 Performance gain using simple parallelization . . . . . 10
4.1.3 Speed of simulations . . . . . 12
4.2 Scaling behavior . . . . . 15
4.2.1 Scaling with tasks and nodes . . . . . 15
4.2.2 Scaling with stars . . . . . 17
4.3 Accuracy of simulations . . . . . 17

**5 Challenges / Discussion** **19**
5.1 Challenges . . . 19
5.1.1 Dark Matter . . . 19
5.1.2 Amount of data . . . 19
5.1.3 Complexity of development for HPC . . . 19
5.2 Future Outlook . . . 20
5.2.1 What we would have liked to do . . . 20
5.2.2 Other approaches . . . 20

**6 Conclusion** **20**

**References** **22**

**A Work sharing** **A1**
A.1 Niels . . . A1
A.2 Amelie . . . A1

**B Code samples** **A1**
B.1 Galaxy generation . . . A1
B.2 Simulation . . . A2
B.2.1 Sequential . . . A2
B.2.2 Parallel . . . A4

# List of Tables

1 Comparison of steps per second for simple and optimized sequential solutions. 9
2 Accuracy of the solutions, shown as percentage of stars within a certain distance of reference . . . 18

# List of Figures

1 Steps per second with a simulation of 10000 stars, with the simple sequential solution, depending on number of tasks and nodes . . . 10
2 Steps per second with a simulation of 10000 stars, with the simple parallel solution, depending on number of tasks and nodes . . . 10
3 Steps per second with a simulation of 10000 stars, with the simple MPI solution, depending on number of tasks and nodes . . . 11
4 Steps per second with a simulation of 10000 stars, with the optimized sequential solution, depending on number of tasks and nodes . . . 11
5 Steps per second with a simulation of 10000 stars, with the optimized parallel solution, depending on number of tasks and nodes . . . 12
6 Heatmaps of the steps per second of different star amounts with the distance dependent sector simulation, depending on number of tasks and nodes . . . 13
7 Maximum Steps per second achieved, depending on the amount of stars and the method used for the simulation . . . 14
8 Percentage increase of steps per second depending on the number of tasks and nodes, for different solutions . . . 16
9 Steps per second scaling based on the method . . . 17

# List of Listings

1 Generating plummer sphere point cloud . . . A1
2 Generating inital masses based on the Salpeter IMF . . . A1
3 Generating inital velocities . . . A2
4 Struct for data handling . . . A2
5 Movement simulation of the SimpleSequential Solution . . . A3
6 Force calculation of the OptimizedSequential Solution . . . A3
7 Changes to parallize with OpenMP . . . A4
8 Changes to parallize with MPI . . . A4
9 Calculations of distance to center . . . A5
10 Logic of Sector based calculations . . . A5

# List of Abbreviations

**HPC** High-Performance Computing

**IMF** Initial Mass Function

**pc** Parsec, a standard measurment of distance in astrophysics, equivalent to roughly 3.26 ly, $2 \times 10^6$ AU or $3.09 \times 10^{16}$ m

## Abbreviations and Names of our Solutions

We tested and compared multiple ways to speed up the simulation of the movement of the stars. All those solutions, we tested have a two letter acronym that are listed here.

**SS** Simple Sequential, the naive approach of for loops inside of each other. Described in the methodology section as the "Simple Sequential Solution"

**OS** Optimized Sequential, a slightly optimized version of the Simple Sequential Solution, reducing memory access. Described in the methodology section as the "Optimized Sequential Solution"

**SP** Simple Parallel, the Simple Sequential Solution, parallelized with OpenMP

**OP** Optimized Parallel, the Optimized Sequential Solution, parallelized with OpenMP

**SM** Simple MPI, the Simple Sequential Solution, parallelized with MPI

**DD** Distance Dependent, a solution that varies the step size based on the distance to the galactic center. Described in the methodology section as the "Radial distance dependent granularity"

**NN** Nearest Neighbor, a solution that varies the step size based on the distance to the nearest other star. Described in the methodology section as the "Neighbor distance dependent granularity"

**MS** MPI Sectors, a solution that summarizes distant stars into sectors to save on compute. Described in the methodology as "Sector based calculations".

**DS** Distance Sector, a solution that combines the Distance Dependent Solution with the Sector solution. Described in the methodology as "Sector based calculations with Radial distance dependent granularity"

**NS** Nearest Sector, a solution that combines the Nearest Neighbor Solution with the Sector solution. Described in the methodology as "Sector based calculations with Neighbor distance dependent granularity"

# 1 Introduction

The simulation of galaxies is a highly complex n-body problem. Each star in a galaxy gravitationally pulls on every other star, leading to complex interactions that can't be solved analytically and that have to be simulated. The main objective of the project is to independently develop increasingly more efficient ways to simulate this problem, analyze their scaling behavior and therefore get an understanding of these different optimization strategies. Using these simulations we can also test the dark matter hypothesis. To tackle this problem, we first started with a naive singe threaded approach of calculating the forces between every star, applying the force and advancing the simulation a small step forward. During the development efforts of the project, we segmented the simulation volume into smaller chunks to summarize

# 2 Methodology

## 2.1 Problem statement

In this project we develop a simulation of stellar movement within a galaxy and both develop and test further optimizations for parallelization.
In order to stay within our constraints of limited computing and project time, we targeted to simulate ultra-faint dwarf galaxies. A class of galaxies that formed only a few million years after the big bang, and only discovered in the last 20 years. They are the most dark matter dominated systems that are currently known to exist in the universe and therefore are stable with only a few hundreds to thousands of stars.
Specifically we simulated 1000 stars with a light-to-matter ratio of 3,400, meaning we have 3,400 times more dark matter than regular matter, over 10,000 years, with simulation steps of one year. The parameters are similar to the parameters of the galaxy 'Segue 1' [Obs11] which served as a reference for our simulation.
To evaluate the scaling behavior of different optimizations, we further simulated galaxies with 5,000, 10,000, 50,000 and 100,000 stars.

## 2.2 Physical equations for generating a spherical galaxy

To simulate the movement of stars in spherical ultra-faint dwarf galaxies, we first have a dataset including the position, masses and velocities of the stars in the galaxy. For that we use the common assumption of a Plummer sphere with the mass distributed according to the Salpeter IMF which are further explained below.

### 2.2.1 Distribution of stars

The distribution of the stars in the galaxy is given by the equation for the Plummer sphere [AHW74]. The density of a Plummer sphere in 3 dimensions is known to be

$$\rho_P(r) = \frac{3M_0}{4\pi a^3}\left(1+\frac{r^2}{a^2}\right)^{-5/2}.$$

Where $M_0$ is the mass of the galaxy, $r$ is the radius and $a$ is the half-mass radius, a physical property of a spherical galaxy describing the distance at which half of the mass

is enclosed within that radius. Therefore, the enclosed mass for any given radius of the Plummer sphere is given by

$$M(r) = M\frac{r^3}{(r^2+a^2)^{3/2}}.$$

We can assume that the masses of stars are distributed randomly as all the stars of such galaxy would have been created at the same time from the same collapsing gas cloud. Therefore, no new star formation could produce local difference, and we can normalize the enclosed mass to get a fraction $X \in [0,1]$ of how much mass is enclosed

$$X(r) = \frac{r^3}{(r^2+a^2)^{3/2}}.$$

This can also be interpreted, according to the previous argument of the masses being distributed randomly, as the fraction of stars enclosed.
To calculate realistic star positions from this equation we then solve for $r$

$$r(X) = \frac{a}{\sqrt{X^{-2/3}-1}} \tag{1}$$

With X being a uniform random number from 0 to 1.
The radii we get from that equation then get distributed uniformly over the sphere giving a set of stellar positions comparable to real observations

### 2.2.2 Distribution of initial masses

The mass distribution of stellar masses follows a trend, described by the Salpeter IMF [Sal55]

$$\xi(m)\Delta m = \xi_0 \left(\frac{m}{M_\odot}\right)^{-2.35} \left(\frac{\Delta m}{M_\odot}\right).$$

This says that the probability $P(m)$ of a star with mass of $m$ and a tolerance of $dm$ is

$$P(m)dm \propto m^{-\alpha}dm$$

With $alpha = 2.35$. We define a minimum stellar mass as $m_{min} = 0.1M_\odot$ and a maximum stellar mass of $m_{max} = 50M_\odot$. Now we can normalize $P(m)$

$$\int_{m_{min}}^{m_{max}} P(m)dm = 1.$$

Therefore,

$$p(m) = \frac{(1-\alpha)m^{-\alpha}}{m_{max}^{1-\alpha} - m_{min}^{1-\alpha}}.$$

With this we can calculate the cumulative distribution function

$$F(m) = \int_{m_{min}}^{m} P(m')dm' = \frac{m^{1-\alpha} - m_{min}^{1-\alpha}}{m_{max}^{1-\alpha} - m_{min}^{1-\alpha}}.$$

Now by rearranging the equation and inserting a uniform random number $X$ from 0 to 1, we can calculate a random mass in a similar way to before, that is distributed according to the Salpeter IMF.

$$m = \left(X\left(m_{max}^{1-\alpha} - m_{min}^{1-\alpha}\right) + m_{max}^{1-\alpha}\right)^{1/(1-\alpha)} \tag{2}$$

**Correction for dark matter** Ultra-faint dwarf galaxies are the most dark matter dominated systems in the universe and a simulation that doesn't account for the mass of the dark matter can't produce simulations of stable systems. Since the positions and masses were placed evenly and because dark matter also abides by the same laws of gravity and is distributed using the same Plummer sphere density function, we can account for the missing dark matter by multiplying the stellar masses with the light-mass ratio of the galaxy. In the case of our reference galaxy Segue 1, it has a light-mass ratio of 3400, meaning that much more dark matter is present in the galaxy compared to matter in stars. Therefore, we also use that multiplier for our stellar masses

### 2.2.3 Distribution of initial velocities

Now that we have our positions and masses we need the initial velocities of the stars, before we can start using this data.
The gravitational potential of a Plummer sphere [AHW74] is known as

$$\Phi(r) = -\frac{GM_{tot}}{\sqrt{r^2 + a^2}}$$

The escape velocity in a gravitational potential is described by

$$v_{esc}\,(r) = \sqrt{2|\Phi(r)|} = \sqrt{\frac{2GM_{tot}}{\sqrt{r^2 + a^2}}}.$$

By introducing the dimensionless variable $q$

$$q = \frac{v}{v_{esc}(r)} \in [0, 1]$$

we can get the probability distribution of

$$P(q) \propto q^2 \left(1 - q^2\right)^{7/2}.$$

Therefore we get the velocity magnitude distribution at the radius of r

$$P\,(v|r)\,dv \propto \left(\frac{v}{v_{esc}(r)}\right)^2 \left(1 - \left(\frac{v}{v_{esc}(r)}\right)^2\right)^{7/2}$$

or explicitly

$$P\,(v|r) = C \cdot \frac{v^2}{v_{esc}^3\,(r)} \left(1 - \frac{v^2}{v_{esc}^3\,(r)}\right)^{7/2}, 0 \leq v \leq v_{esc}(r)$$

with $C = \frac{512}{7\pi}$.
Instead of calculating a direct solution we propose a solution to a rejection sampler and get our velocity through that. That velocity is then given a random direction on the sphere in accordance with the Virial theorem.

## 2.3 Physical equations for simulating the movement of the stars

Now that we have our galaxies modeled, we can look at the relatively more simple part if simulating the movement

### 2.3.1 Modeling two-body systems

Between every two objects there exists a gravitational force, described with

$$F = G \cdot \frac{m_1 m_2}{r^2}.$$

The general equation for force is

$$F = m \cdot a.$$

Therefore, the acceleration $a_1$ of object $O_1$ with mass $m_1$ due to the gravitational force being created by object $O_2$ with mass $m_2$ is

$$a_1 = G \cdot \frac{m_2}{r^2}.$$

Using this equation, we can calculate the future position of both objects for any future date, assuming only these two objects influence each other and nothing else.

### 2.3.2 Modeling n-body systems

While one or two bodies in a gravitational system lead to easy equations, the introduction of a third or more bodies leads to a chaotic system that can't be described analytically. Therefore, to solve these systems we simulate them numerically, by calculating the forces between each objects, calculating the change in velocity of each object and then letting each object move for a small amount of time.

## 2.4 Solution approaches

### 2.4.1 Sequential solutions

**Simple Sequential Solution.** To simulate the movements we now can use our set of data along with a predetermined number of steps, and time span each step should cover. For each time step we iterate through every star, accumulate the acceleration they experience by again iterating through every star and calculating the force the star experiences. Then we can move the star according to the new velocity vector for one time step, before doing it again.

**Optimized Sequential Solution.** Iterating twice through all stars is inefficient since we look at each pair of stars twice. By calculating not just the force of one star, but both stars every time we can cut the number of iterations and the memory reads in half. This also gives us the ability to share some of the calculation, further cutting the time the simulation needs.

### 2.4.2 Parallel Solutions

The easiest way to parallelize this serial calculation is to distribute the iterating through the stars onto more nodes using OpenMP and MPI. While this is something we have explored and tested, in the following we would like to discuss logical changes to the calculation process we explored. As shown above, stars in a longer distance have a smaller influence on the allocation than stars nearby. According to this effect the calculation can be optimized.

**Radial distance dependent granularity.** According to the density profile of the Plummer sphere more stars are in the center of the galaxy, meaning they are closer together. And we know that the force a star experiences scales with $1/r^2$. Therefore, we can assume that stars more in the center of the galaxy experience more force from the other stars, resulting in more change in their trajectory in any given time and therefore their velocity vector needs to be updated more frequently to have the same accuracy.
For our example of a galaxy with 1000 stars, has a half-mass radius of 30 pc. For our simulation we use a step size ten times bigger then our sequential solution, but at a distance of 15 pc from the center we decrease that step size again, by a factor of 8 and within 7.5 pc of the center, we double it again to 16.
Given that many stars now will only be updated a tenth of the times and some more frequently, we hope to achieve an improvement of 2-5x in terms of speed of calculation while keeping the accuracy high.

**Neighbor distance dependent granularity.** Instead of assuming that the stars near the galactic core are more likely to have close neighbors, we can also check the distance to the nearest neighbor for every star and do our computation based on that. We expect to need some additional computational power to calculate the nearest neighbors of all stars, but this way we can use more precision for the stars that need it the most and avoid giving additional computational resources to stars that don't need it.
For our implementation that means a step size ten times bigger as our sequential solution, just like before, 8 times more precision, when a star is within 8 pc and 16 times more precision when a star is within 4 pc.
We aim for a similar improvement as to the radial distance solution. However, less sparse galaxies would mean more and more of the stars would have close neighbors, leading to the optimization potentially performing worse than the original unoptimized version.

**Sector based calculations.** As long as only the update rate is modified still the influence of every star is calculated. It can be made even more efficient by using sectors. For that entire simulation area range is divided into cubes. A star experiences the forces and accelerations by the other stars within every cube (or sector) as normal while the stars in the remaining sectors are grouped together as a mass at the center of their sector and interact with the star only by the chunk approximation.
For our simulations we divide the simulation area into 1000 sectors with lengths of 100 pc, given a simulation area of $1000\,\text{pc} \cdot 1000\,\text{pc} \cdot 1000\,\text{pc}$.
For a simulation with only 1000 stars we do not expect any improvements, but with increasing number of stars, this optimization should be a significant improvement as it has a significantly better scaling behavior.

**Sector based calculations with Radial distance dependent granularity.** By combining the optimization strategies of increasing the step size farther away from the center and decreasing it more in the center and grouping distant stars into sectors, we hope to achieve a combination of the effect, of more precision where it is needed, while also decreasing the amount of calculations and the better scaling of sectors with higher number of stars, compared to calculating the forces between every single star.

**Sector based calculations with Neighbor distance dependent granularity.** The last method we explored combines the neighbor distance dependent granularity with the

sector based calculations.

### 2.4.3 Validation of improvement

For validating our results we used two different approaches. On the one hand we analyzed the runtime of the different programs and on the other hand we looked at the simulation itself to validate the accuracy of the location of the stars.

**Steps per second** The performance analysis calculates the steps per second by multiplying the percentage of the total progress of the simulation by the target number of steps. This is divided by the time that has elapsed up to the percentage achieved. In the end we get a performance value for each recorded run of the simulation. Since the recording is done every percent of the total steps to be calculated. For comparison between different runs we used the maximum steps per second of each run.

**Accuracy of location** However, it is not only important how much time a simulation takes, but also how good the result is. According to the physics every simulation with the same starting position of the stars should return the same result, regardless the method used, if the steps are small enough. We compare the stars position of two log-files after the simulation by calculating the Euclidean distances of the same star in both files. The percentage of stars within a previous chosen limit compared to the results of the Simple Sequential solution is the accuracy value.

# 3 Implementation

The programs to generate the galaxies and to calculated their movements where written in C and ran on the servers, while the analysis was done off-site with Python.
For our implementations our coordinates are defined as parsecs from the galactic core, in the three cardinal directions.

## 3.1 Galaxy generation

The generation of the galaxy was done once per size of galaxy on the server. Since it doesn't have to be repeated for every run and scales linearly with the number of stars instead of in quadratically, we elected not to spend time parallelizing it.

### 3.1.1 Distribution of stars

The way to generate a distribution of stars, as a Plummer sphere is given by inserting random uniform numbers from 0 to 1 into the equation 1, which we implemented in the code block 1.
In the code snippet, '$N$' is the number of stars to be generated, '$a$' stands for the half-mass radius, and the arrays of '$x$', '$y$', '$z$', '$r$' describe the positions of the stars.

### 3.1.2 Distribution of initial masses

The way to generate the initial masses according to the Salpeter IMF is to inserting random uniform numbers from 0 to 1 into the equation 2, which we implemented in the code block 2.
In the code snippet, '$N$' is the number of stars to be generated, '$alpha$' is the Salpeter slope defined as 2.35 as described before,'$m_{min}$' and '$m_{max}$' are the minimum or maximum masses of our stars, here defined as 0.1 or 50 times the mass of the sun respectively, and '$m$' is the array describing the masses of the stars. Since the positions and masses are randomly distributed we do not need to take the position of the star into consideration.
The global variable '$DarkMatterCorrection$' applies a correction factor as discussed in the paragraph about dark matter at 2.2.2.

### 3.1.3 Distribution of initial velocities

The way to calculate the initial velocities is given in section 2.2.3.
For our implementation, we first calculate the escape velocity at the galactic center. Every star then gets their local escape velocity calculated.
To get the randomly distributed values, we take two random uniform values between 0 and 1. The first we scale based on the probability density function. This gives us our 'x-value', the other one we scale based on the maximum of the probability density function at around 0.1. With that we transformed our problem into a linear one and to produce values that are distributed as needed, we just need to check that the first scaled value is under the line/smaller than the second scaled value.
Now that we have a properly distributed range, we can multiply the value with the local escape velocity and apply that into a random direction.

## 3.2 Movement simulations

All implementations of the simulation have a minimum distance of $\epsilon$ defined. In the code implemented as the variable '*eps*'. The likelihood of any two stars colliding or just coming together close enough to significantly effect their orbits is incredibly small and would only occur once in every thousand runs, if we estimate it with a generous margin. With later simulation solutions, the ramifications of a star shooting out of the galaxy, could lead to loss of data on that star or significantly longer calculations as the area of our simulation gets bigger. For that reason, whenever we calculate the distance, we use a four dimensional Pythagoras with all three space dimensions and a fixed $\epsilon$-value of 0.1 pc.
After every percent of progress in the simulation we save a snapshot with all position, movement and mass data, as well as the parameters we input at the start of galaxy formation and simulation. This helps automate the analysis of data as well as giving us the option of continuing from a snapshot, should we require that.
The data for every star is stored in a struct as seen in the code block 4

### 3.2.1 Sequential Solutions

**Simple Sequential Solution.** All our solutions were developed from a sequential solution, described in the section 2.4.1 as the simple sequential solution. The main logic of the code can be found in the code block 5.
To do our simulation, we have 3 nested for-loops. The first one goes through the time

steps we set and the second and third go through every combination of stars to calculate the forces the second acts upon the first. After that we go through each star again and calculate their new velocities and positions after experiencing the forces and movements for a set period of time.

**Optimized Sequential Solution.** In the code block 6 you can see the changes that are needed to accommodate both calculations at once. Since we do not just go through all stars one after the other, we already have to have all the memory of the last step wiped, before even computing the acceleration of the next star.

### 3.2.2 Parallel Solutions

There are two separate tools to parallelize the code. OpenMP and MPI. We started with OpenMP to do a simple parallelization of our code, before using MPI for more complex strategies

**Parallelizing with OpenMP** In the code block 7 we have displayed the changes needed to turn the previously mentioned sequential solutions into parallel ones. It is a simple solution that works, but that does not offer the same flexibility as MPI, which we will be using from now on.

**Parallelizing with MPI** Our simplest MPI implementation works by taking the simple sequential approach, splitting the evenly distributing the stars between all tasks and then only calculating the forces for the stars assigned to that task. Between every time step, the new positions get distributed to all other tasks, to keep the simulation data current and accurate.
The main logic of that can be found in code block 8.

**Radial distance dependent granularity.** The main logic can be found in the code block 9.
To achieve more granularity at the center, we first go through every star and calculate its distance to the center. If it is within the '$r\_ref$' distance, defined as 15 pc, it is within the outer threshold, if it is within half of that it is in the inner threshold. When we do a timestep, we further do 16 cycles. If the star is in the inner threshold it gets its acceleration computed every cycle, if it is in the outer threshold, it only gets updated every second cycle and stars outside of the thresholds only get updated on the last of the 16 cycles.

**Neighbor distance dependent granularity.** The neighbor distance calculation is the exact same idea, but instead of calculating the distance to the center, we go through each star and calculate the distance, to get the distance to the nearest neighbor.
The outer threshold is here defined as 8 pc with the inner threshold being half that.

**Sector based calculations.** The logic of our sector based calculations is in code block 10.
Before our logic we created 1000 sectors (10 by 10 by 10, with a side length of 100 pc each) that comfortably fit even the most extreme outliers of our simulation. At the beginning we assign each star into its sector with the function $sector\_index$. Then in the simulation,

we calculate the mass and weighted average position of the stars in each sector. We compute the forces on each star from each other star in the sector as before, but for stars outside the sector, we use the total mass at the weighted average position for each sector, instead of the stars in them.

**Sector based calculations with Radial distance dependent granularity.** For this approach we took the code for the sector based calculations and changed the calculation of the forces within the sector to the one of the Radial distance solution.

**Sector based calculations with Neighbor distance dependent granularity.** For this approach we took the code for the sector based calculations and changed the calculation of the forces within the sector to the one of the Neighbor distance solution.

# 4 Performance analysis / evaluation

We tested our code for both speed and accuracy.
The speed tests were done on the SCC-CPU cluster. For them, we simulated 10,000 years with steps of 1 year for 1000, 5000, 10000, 50000 and 100000 stars. We did that with all our solutions running as 1, 2, 4, 8, 16 or 32 tasks on 1, 2, 4, 8, 16 or 32 nodes. All runs were capped at 10min of runtime and while that meant many runs were cut short prematurely, we could almost always get useful data from it, since every percentage of the simulation completed, we save a snapshot that includes the completed amount of steps and the time it took so far, giving us insight into the speed of the simulation.
The accuracy tests comprised of simulations of 1000 stars for 10,000 years with steps of 0.1 years and were done both on the cluster, as well as external machines, after we verified that the data is indeed the same.

## 4.1 Performance

### 4.1.1 Performance of the sequential solution

To measure the performance of our optimizations to the sequential solution, we ran both the optimized and the unoptimized solutions 21 times with 5000 stars and 21 times with 10000 stars.

| | Simple Sequential | Optimized Sequential |
|---|---|---|
| 5000 stars | $408 \pm 5$ | $823 \pm 5$ |
| 10000 stars | $203 \pm 2$ | $386.8 \pm 0.5$ |

Table 1: Comparison of steps per second for simple and optimized sequential solutions.

This leads to a speed up of $102\,\% \pm 3\,\%$ for the simulation of 5000 stars and a speed up of $91\,\% \pm 2\,\%$ for simulations with 10000 stars.
On average our optimizations speed up the simulation by $97\,\% \pm 3\,\%$.

### 4.1.2 Performance gain using simple parallelization

**Simple Solution** The possible performance gain for simple parallelization is here illustrated with runs with 10000 stars. We compare the sequential solutions to parallel solutions that have the core logic unchanged, as described in section 3.2.2 *Parallelizing with OpenMP* and *Parallelizing with MPI*.

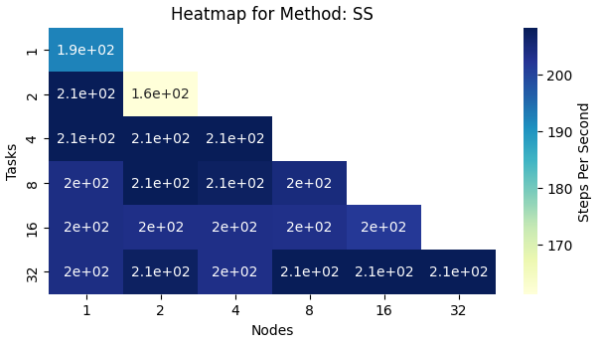



Figure 1: Steps per second with a simulation of 10000 stars, with the simple sequential solution, depending on number of tasks and nodes

In figure 1, you can see our sequential solution. As expected, there is a bit of variance, but the number of tasks and nodes does not have any effect on the speed of the simulation, which reached up to 208 steps per second.

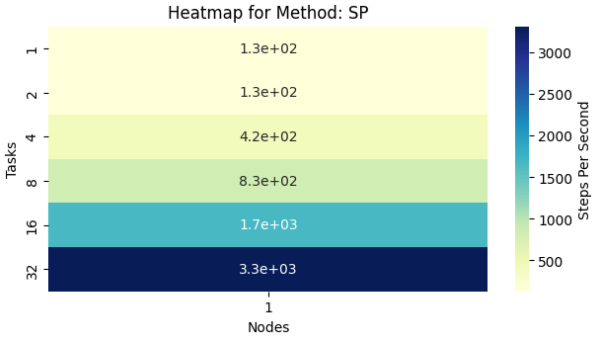



Figure 2: Steps per second with a simulation of 10000 stars, with the simple parallel solution, depending on number of tasks and nodes

In figure 2 we see the steps per second for the same code, parallelized with OpenMP. The y-axis in this case stands for the number of threads that OpenMP has access to. While this parallelization suffers a penalty, from the extra overhead, when using just one or two threads, it is faster by over an order of magnitude, compared to the sequential solution with 3315 steps per second, when using 32 threads.
The mayor drawback of OpenMP is however, that it does not work across multiple nodes or tasks, making it impossible to scale beyond a certain point.

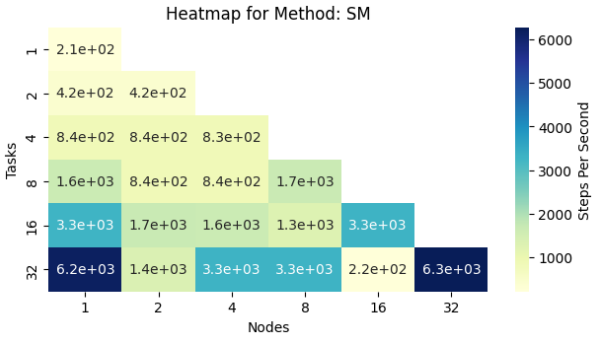



Figure 3: Steps per second with a simulation of 10000 stars, with the simple MPI solution, depending on number of tasks and nodes

Looking at figure 3, we can see the speeds of the simulation with different combinations of tasks and nodes for the simple parallelization using MPI. The speed of it peaks at 32 tasks on either 1 or 32 nodes with up to 6282 steps per second. Almost a doubling of the result of OpenMP.

**Optimized solution** In the following figure 4 you can see the speed of our optimized sequential solution, based on number of tasks and nodes. Again, you can see that there is some variance, but as expected for a sequential, there is no correlation between more tasks or nodes and more performance. We can however observe the performance uplift, discussed in 4.1.1. This leads to a peak speed of 389 steps per second, an increase of 87 %.

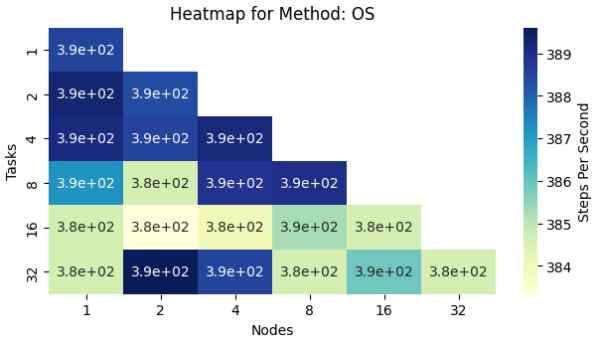



Figure 4: Steps per second with a simulation of 10000 stars, with the optimized sequential solution, depending on number of tasks and nodes

It is important to note however, that this does not scale with more cores, nearly as well as the simple solution. As you can see in the figure 5, we get barely any improvement, with 435 steps per second being the fastest run, while the performance even regressed at 32 threads.
This can be explained by the fact that OpenMP divides the stars into equal chunks, our optimization now gives the first stars a bit more forces to compute to save on additional loading and compute time for the later stars. This triangular loop means that one thread will have orders of magnitude more work than another, negating a lot of the speed up.
Another even more significant contributor to this meager performance is the penalty we incur because of cache contention. In our sequential solution, the outer loop decides what star we are currently updating and the inner loop decides from what other star, the forces we are currently calculating are coming from. That means, when dividing the workload

onto more threads, every thread will have their own set of stars that are only updated by that thread. With the optimized approach, we also update the star from the inner loop. Now we can have the problem, that multiple threads try to update the same star, leading to a cache contention and the CPU pausing the threads until all updates to the stars acceleration have been processed one after the other in a queue.

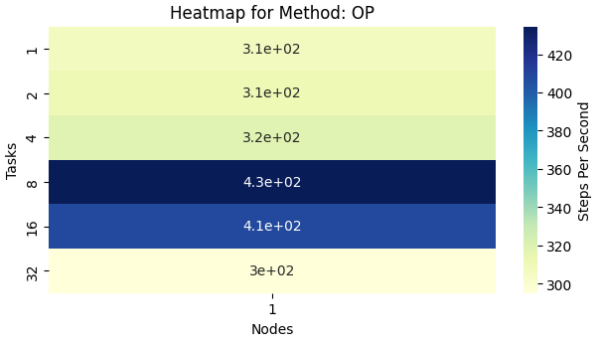
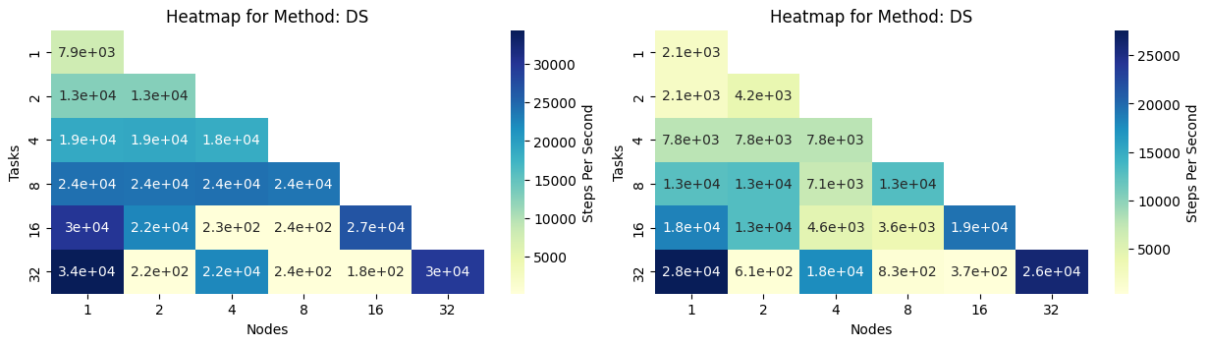



Figure 5: Steps per second with a simulation of 10000 stars, with the optimized parallel solution, depending on number of tasks and nodes

### 4.1.3 Speed of simulations

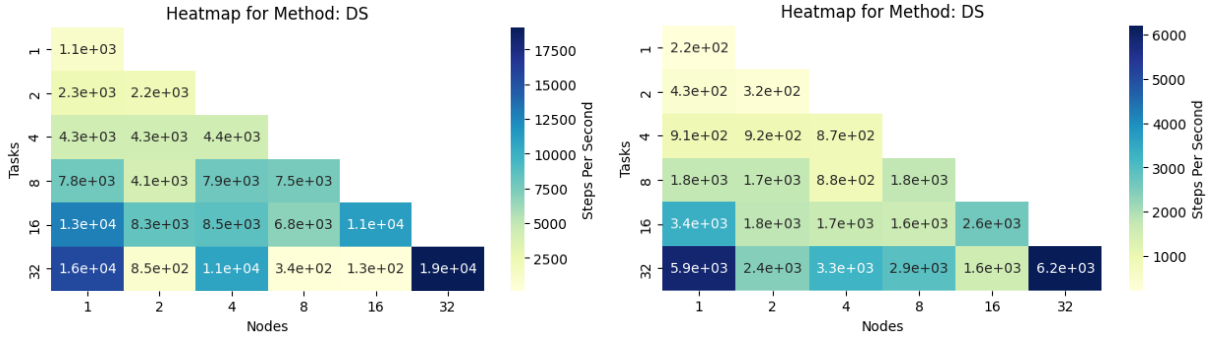
Every combination of solution, star amount, nodes and tasks has run at least once on the cluster. As an example, we have shown the results of the distance dependent sector solution in figure 6.




(a) 1000 stars



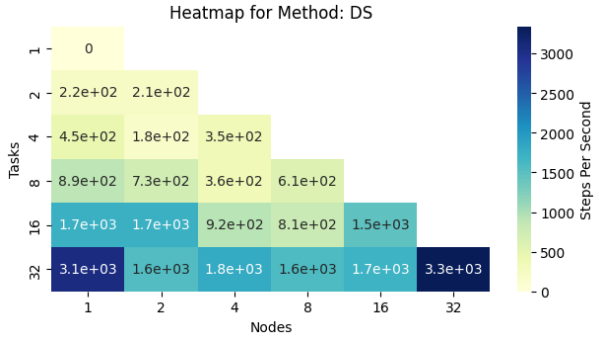

(b) 5000 stars




(c) 10000 stars




(d) 50000 stars




(e) 100000 stars

Figure 6: Heatmaps of the steps per second of different star amounts with the distance dependent sector simulation, depending on number of tasks and nodes

To get these results, we take the snapshot that got the furthest into the simulation and divide the amounts of steps it got to by the timestamp of the snapshot. This gives us the speed of that run.
The value 0 means that the simulation took at least one step, but it didn't complete at least one percent of the simulation before the allotted time of 10min ran out.

We create similar heatmaps for every combination of solution and star amount and take the highest result of each as the speed, that solution can reach. To now compare the speed between our solutions, we plot the steps per seconds achieved per method against the amount of stars and get a ranking of which is the fastest or the slowest and how does that compare across the different amounts of stars. This can be seen in the figure 7.

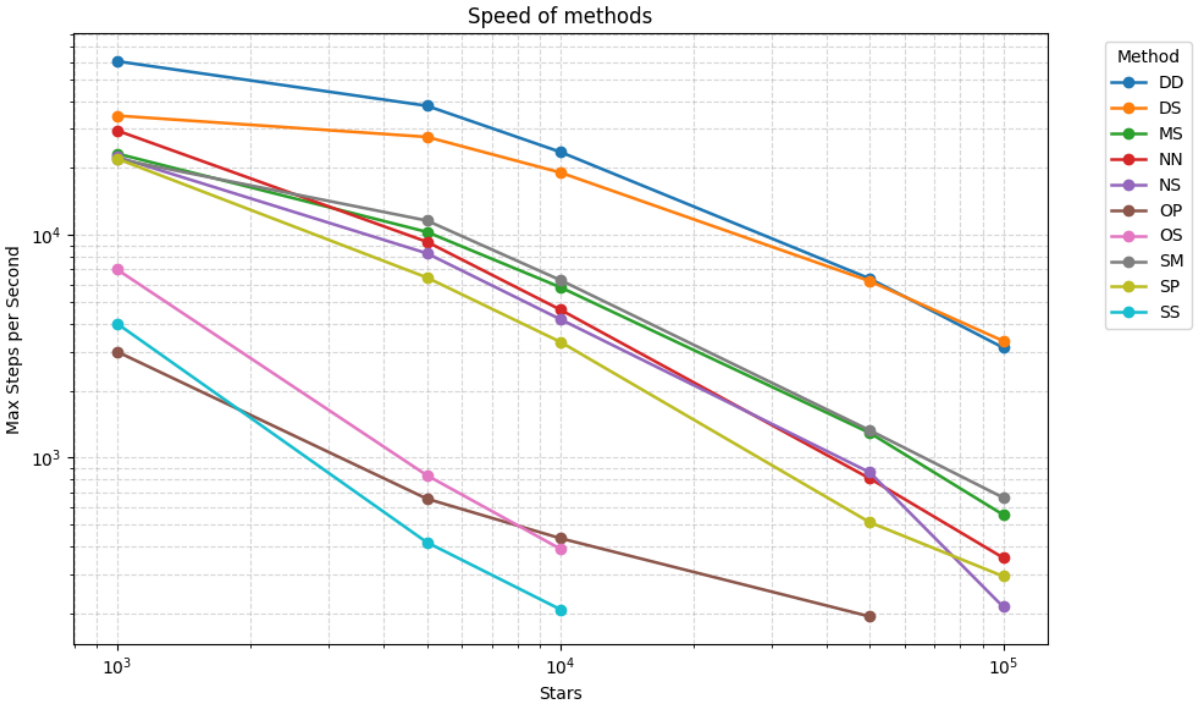



Figure 7: Maximum Steps per second achieved, depending on the amount of stars and the method used for the simulation

We can see three distinct groupings. A group of slow solutions, a middle pack and two fast solutions.

The slow solutions consist of our simple and optimized sequential solution as well as the optimized solution, parallelized with OpenMP. As we expect, the sequential solutions are the slowest. The optimized solution, parallelized with OpenMP, starts in the same grouping, initially even performing worse than the sequential solutions, because of the cache contention, previously discussed in 4.1.2. With more stars and the same number of threads, the likelihood of cache contention decreases, increasing it's speed relative to other solutions dramatically at higher star counts. In our testing however, it is never actually close to being as fast as the simple sequential solution with the same parallelization.

The middle pack shows us that a lot of our solutions are just about trading blows with a simple solution optimized with MPI. The solutions that calculate the distance to the nearest neighbor, before deciding, how granular the star need to be updated save a lot of time in their computations, but the calculations to get the distance to other stars is compute intensive enough to offset the gains at higher number of stars. Out of the middle pack the only other stand out solution is the sector solution, without any other extra calculations. It is very close to the simple MPI solution and with it theoretically scaling better with more stars, it is promising.

The overall stand outs are the distance based solutions. They save so much compute power by being less granular further for the center, that they are clearly the fastest solutions. We can clearly see the penalty that the solution incurs, when combining it with the sector based approach at lower counts of stars, but what we also clearly see is that the gap is closing with more stars and finally the solution that combines sectors with

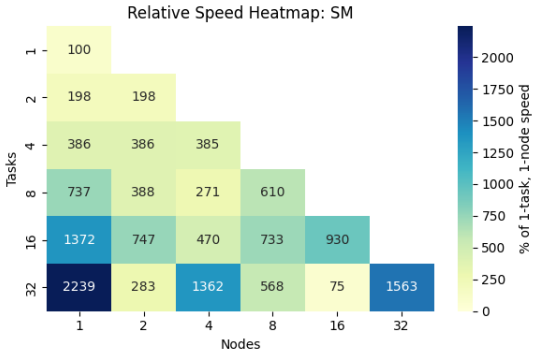
the distance based granularity is the fastest solution for 100,000 stars.

## 4.2 Scaling behavior

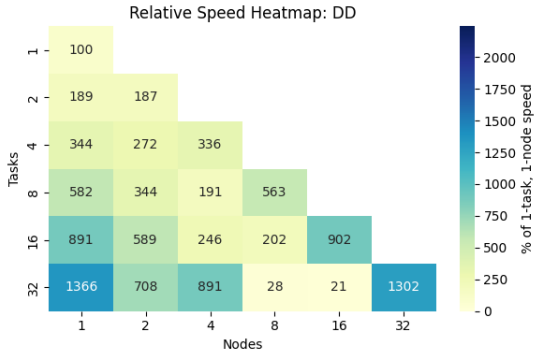
Now that we looked at the speed of the solutions, lets look at the scaling of them.

### 4.2.1 Scaling with tasks and nodes

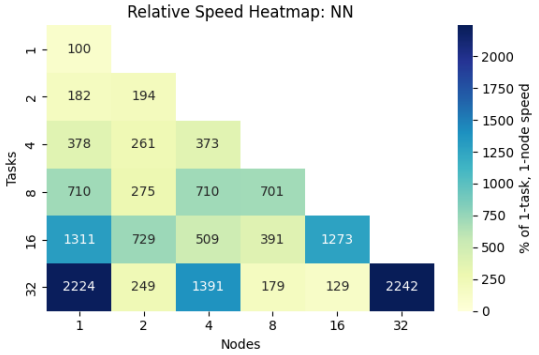
When running a simulation on the SCC-Cluster, we can choose the amount of tasks and nodes the work should be distributed on. To show how that effects the speed, we took our 6 solutions that rely on MPI, scaled the speed to the percentage of speed it had with only one task on one node and then averaged those percentages from runs with 1000, 5000 and 10000 stars. You can see the resulting speed up or slow down, depending on the amount of nodes and tasks in figure 8. Runs with higher number of stars weren't used, as they had configurations, where the first percentage wasn't completed.



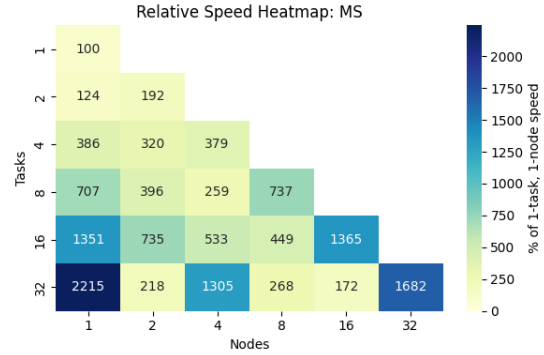

(a) Scaling of performance of the Simple MPI solution.



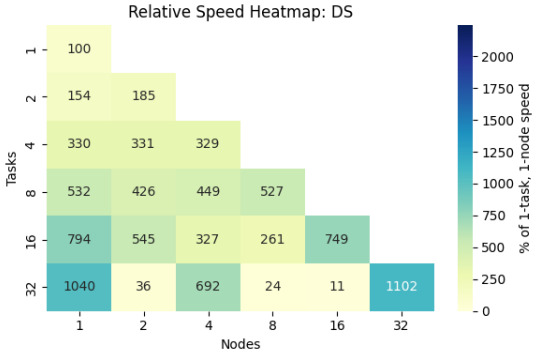

(b) Scaling of performance of the Distance Dependent solution.



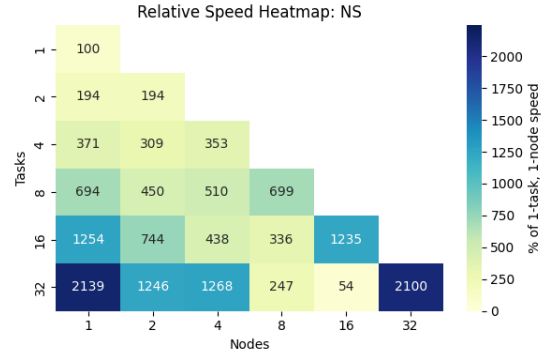

(c) Scaling of performance of the Nearest Neighbor solution.




(d) Scaling of performance of the MPI Sector solution.




(e) Scaling of performance of the Distance Sector solution.




(f) Scaling of performance of the Nearest Sector solution.

Figure 8: Percentage increase of steps per second depending on the number of tasks and nodes, for different solutions

You can clearly see three trends. Firstly the code scales best with all tasks on one node. Secondly, distributing all tasks on their own node, results in pretty similar, if not identical scaling. And lastly, while mixing the number of nodes and tasks leads to bad scaling, sometimes even a performance regression, having 32 tasks divided onto 4 nodes seems to be a sweet spot that scales far better than distributing the task on more or less nodes (except having them all on one node, or all on separate nodes).
It is important to note that even the best scaling combinations, do not scale linearly with the amount of tasks, as there is a considerable overhead in collecting and distributing data between the ranks. This also explains the different scaling behaviors. When all ranks are

on the same node, the shared memory enables fast exchange of data, resulting in high performance gains. A similar thing is happening, when every rank is on a separate node. It does not have to share the memory bandwidth with any other rank and therefore can also exchange data faster.
While we do not know the underlying network architecture well enough to pin point an exact reason, it is also our observation, that using 8 ranks per node allows the memory bandwidth to be used more efficiently, leading to more through put at that sweet spot.

### 4.2.2 Scaling with stars

To measure the scaling of our solutions in respect to the amount of stars, we take figure 7, use the speed of each method at 1000 stars as a reference and plot the relative speed of the method with more stars, instead of the absolute speed. This leads to the figure 9, which shows how each solution scales with more stars.

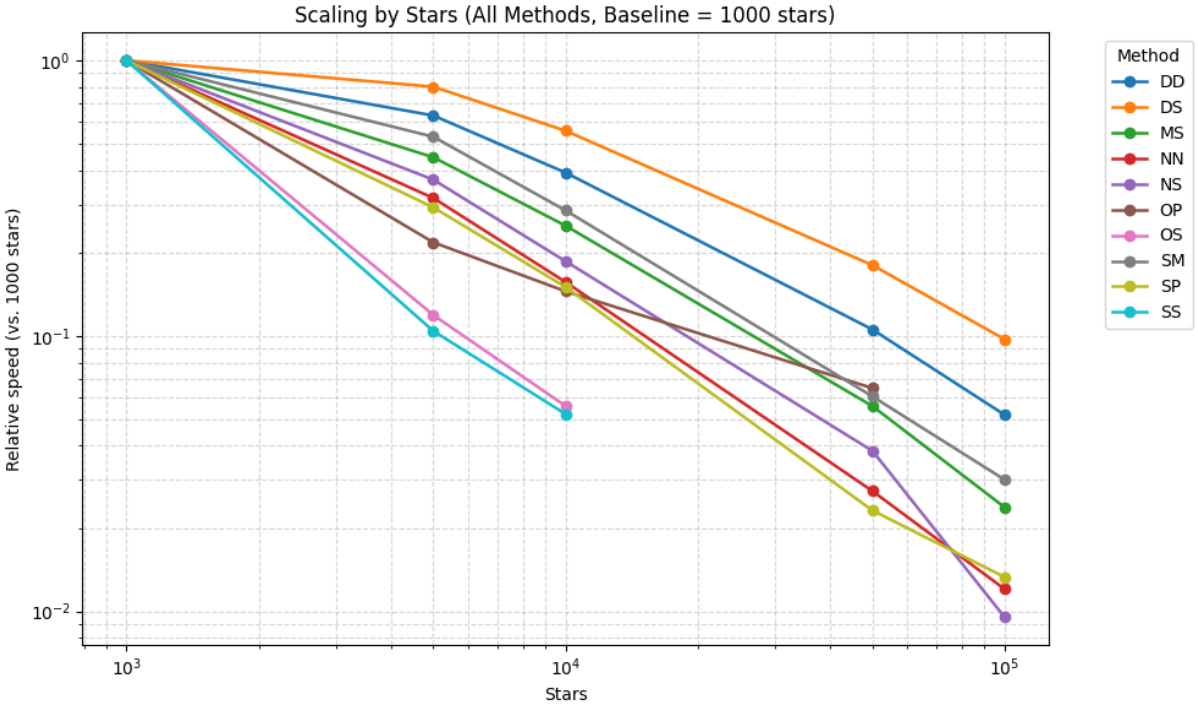



Figure 9: Steps per second scaling based on the method

What we can see in the graph is that for our runs the distance dependent sector solution scales the best while our sequential solutions have the worst scaling.
The calculations to find the nearest star have a huge penalty for scaling, while simply assuming that farther out stars are farther away from the nearest star tops the charts. Combining an approach with sectors also generally seems to improve the scaling behavior.

## 4.3 Accuracy of simulations

While we discussed the speed and scaling behaviors of our solutions a last question lingers. Are they accurate enough?
As previously stated, the simulation for the accuracy measurements were done with 1000 stars and for 10,000 years, with a step size of 0.1 years. Since we do not care about the

speed of the operation and we could verify that the same code gives the same result, regardless of if it is running on the cluster or a different machine, these simulations ran on a personal computer with an AMD Ryzen 8840HS. Where possible the code ran on all 8 cores/16 threads.
There is no real world data we can draw from to compare, therefore, we can only compare our results to a reference. In this case the sequential solution with out any optimization. To compare the simulations, we compare the distances of the stars in our simulation to the reference and calculate how many of the stars are within a certain distance threshold compared to the reference. This can be seen in table 2.

| Solution | $1 \times 10^{-4}$ pc | $1 \times 10^{-5}$ pc |
|---|---|---|
| Simple Sequential (Reference) | 100 % | 100 % |
| Simple Parallel | 100 % | 100 % |
| Simple MPI | 100 % | 100 % |
| Optimized Sequential | 100 % | 100 % |
| Optimized Parallel | 99.6 % | 65.2 % |
| Nearest Neighbor | 100 % | 99.9 % |
| Distance Dependent | 100 % | 99.9 % |
| MPI Sector | 100 % | 81.3 % |
| Nearest Sector | 100 % | 81.4 % |
| Distance Sector | 100 % | 81.4 % |

Table 2: Accuracy of the solutions, shown as percentage of stars within a certain distance of reference

What we can clearly see is that the solutions that do not change the underlying logic all give the exact same results as our reference, with only one exception. Parallelizing the Optimized Solution not only leads to performance degradation, but also to inaccuracies in the simulation. While the other solutions before keep the order in which the acceleration is added up, including the same code, when just using a single thread, using more threads will rearrange that order, resulting in floating point errors that compound over time. That is because we do not simply go through each star and calculate the forces for that star, but also for a different star that otherwise comes later. This isn't a problem with only one thread, as the order is still respected. We go through every star that has an index smaller and calculate the force that acts from that star, before directly calculating the forces from stars with a higher indices. With more threads, a later star will sometimes already have the forces from stars with higher indices calculated, before stars with a lower index in a different thread get to that star.
We should note that we are talking about over half of all stars not being within a distance of roughly the mars to the sun, in a galaxy roughly 6 million times bigger when using the half-mass radius or 100 million times, if we look at the farthest stars, still part of the galaxy.
We can safely say that any simulation that is more accurate than that is accurate enough.

Our Nearest Neighbor and Distance Dependent solutions are incredibly accurate with 99.9 % of all stars being within $1 \times 10^{-5}$ pc from the reference, despite using step sizes, ten times bigger compared to the reference, for most stars.

Implementing sectors into the solutions has a noticeable impact on accuracy, however with 81.3 % or 81.4 % when varying the step size based on distance to neighbors or the center the solution is still very accurate.

# 5 Challenges / Discussion

## 5.1 Challenges

### 5.1.1 Dark Matter

The first problem we encountered was that the very first galaxies we simulated weren't stable and would diverge by a lot in a short amount of time. With the idea of using sectors already in our minds, this could lead to loss of data, when a star ventures outside of the sectors. After triple and quadruple checking our equations to generate and simulate the galaxies, we came to the same conclusion as the field of astronomy before us. We needed a correction factor in our mass. By multiplying the masses of the stars with a factor of 3400 the galaxy simulations became stable.
To do several long term simulations with the needed precision needed to find the perfect light-mass ratio for this galaxy would have meant significant time and compute resources invested in a small side question and was unfortunately not possible.
We can however confirm the dark matter hypothesis anecdotally. Without a factor similar to the one of Segue 1 in the real world, the simulation is not stable.

### 5.1.2 Amount of data

All in all, we gathered data from 2176 runs that were completed on the SCC-CPU cluster and a few hundred more on both an 8-core laptop and an 12-core desktop computer.
This resulted in almost 100 GB of data, that was generated and saved. Managing this was a challenge.
(We would like to note that we very regularly copied and removed the data from the cluster and that we weren't a significant contributor to the storage problems on there.)
30 GB of that were used for the analysis. We could have decreased the size of that to about 500 MB by removing almost all snapshots except the very latest and by removing the star locations and velocities from the data which we use for the performance tests, we could further go down to a few Megabytes. But doing that would remove the ability to continue from a snapshot and reducing the amount of snap shots would hinder the diagnostic of our simulations.
Therefore, this remains a challenge that might be unsolvable without accepting throwing away data.

### 5.1.3 Complexity of development for HPC

Comparing the slides of the presentation and the report, you will see some differences in our numbers and conclusions.
Our understanding of the data and of developing for HPC systems has continuously improved over the span of the project. While this is a very good thing, it also has lead to the realization, that we were previously interpreting some data wrong, or that there was a previously undiscovered bug in our code.

While the underlying logic of our solutions have not been touched since our presentation of our results, even though we would like to bring some improvements, some bug fixes have lead to changes in the data and therefore our conclusions.

## 5.2 Future Outlook

### 5.2.1 What we would have liked to do

As described previously in the section above, we learned a lot and would like to do significant changes to speed up the code. For example, the code that calculates how far away the nearest star is could have been made redundant if we were saving the smallest distance while calculating forces in the previous step. Things like that could improve our code and with more time produce even faster and accurate results.
The one thing we really wanted to with our data is to create physically accurate visualization. We know the mass of the stars and using the approximation that the stars are all from the main sequence, we can use that to calculate the dominant wave length and brightness of the star, creating a physically accurate and beautiful visualization. Unfortunately, that took too much of our time.

### 5.2.2 Other approaches

When googling or asking ChatGPT for solutions to this problem, you will stumble across the Barnes-Hut simulation. A method of recursively dividing space up into sectors for simulations similar to the ones we did.
The reason that we didn't use that one is that we wouldn't learn anything. We could copy the code for that algorithm from GitHub or ChatGPT and get great results. And if we were only after results that would do. But in a course designed to learn how do use HPC systems and how to develop for it, we decided it would be a better use of our time to see for ourselves, what works or doesn't work.
Similarly writing the code for GPUs would have been incredible, unfortunately the time investment needed to write for a completely different architecture would be enormous and that code wouldn't even run on the cluster that we were using.

# 6 Conclusion

We set out to write a program that would generate realistic spherical dwarf galaxies and simulate the movement of stars within them. Our goal was to develop solutions that could speed up the simulation by utilizing the resources available on an HPC cluster while developing our understanding of working with and developing for HPC clusters.
Ultimately, we were successful in this endeavor and on the way improving the speed of the simulation by a factor of over 100, when comparing the 200 steps per second our simple sequential solution could achieve with 10000 stars, compared to the 23 000 steps per second of the distance dependent solution running with 32 ranks at the same time.
With this, we learned that in a spherical galaxy, the changing the step size based on how far the nearest other star is, gives the same results accuracy wise, as changing the step size based on the distance to the center, but with much less compute overhead. We further learned that in order to keep a galaxy stable, especially one as sparse as the ones

we simulated, we need to account for a significant amount of dark matter. Furthermore we should be cautious with using OpenMP without making sure that multiple threads don't need to change same variable, as this can result in significant cache contention that tanks performance.

# References

[AHW74] S. J. Aarseth, M. Hénon, and R. Wielen. "A Comparison of Numerical Methods for the Study of Star Cluster Dynamics". In: *Astronomy and Astrophysics* (1974).

[Obs11] Keck Observatory. *Found: Heart of Darkness*. `https://keckobservatory.org/found_heart_of_darkness/`. Accessed: 2025-09-27. 2011.

[Sal55] Edwin E. Salpeter. "The Luminosity Function and Stellar Evolution". In: *Astrophysical Journal* (1955).

# A Work sharing

## A.1 Niels

Niels was primarily responsible for the introductory theoretical section. He conducted the necessary literature research to ensure that our assumptions for the subsequent model were physically correct. Furthermore, he developed the code for the sector based solutions of our work. He submitted the programs to the clusters for testing and evaluation. The presentations were prepared and delivered trough collaborative work. Niels wrote the methodology section and his sections on implementation.

## A.2 Amelie

Amelie programmed a sequential solution to the problem. She then transformed the code into parallel solutions using OpenMP and MPI. To further improve performance, she implemented programs for radial and neighbor distance dependent granularity. The next step was to evaluate and compare the results of the different approaches, which she also did. Using these results she returned to the code and modified it. As described above, the presentations were created through collaborative work. For the report, Amelie wrote most of the implementation, performance analysis and discussion sections.

# B Code samples

## B.1 Galaxy generation

```
void make_point_cloud(int N, double a, double *x, double *y, double *z,
    double *r)
{
     for (int i = 0; i < N; i++)
     {
          //Code to get a random distance from the center, based on the
    plummer density
          double X = rand_uniform();
          r[i] = a / sqrt( pow(X, -2.0/3.0) - 1.0 );
          //Code to get a random point on the shpere with a known radius
          double u = rand_uniform();
          double cosTh = 1.0 - 2.0*u;
          double sinTh = sqrt(1.0 - cosTh*cosTh);
          double phi = 2.0 * PI * rand_uniform();
          //Conversion to cartesian coordinates
          x[i] = r[i] * sinTh * cos(phi);
          y[i] = r[i] * sinTh * sin(phi);
          z[i] = r[i] * cosTh;
     }
}
```

Listing 1: Generating plummer sphere point cloud

```
void distribute_stellar_mass(int N, double m_min, double m_max, double
    alpha, double *m)
{
```

```
    double exp = 1.0 - alpha;
    double C = pow(m_max, exp) - pow(m_min, exp);
    for (int i = 0; i < N; i++)
    {
        double u = rand_uniform();
        m[i] = DarkMatterCorrection * pow( u*C + pow(m_min, exp), 1.0/exp );
    }
}
```

Listing 2: Generating inital masses based on the Salpeter IMF

```
void distribute_stellar_velocity(int N, double a, double Mtot, double *r,
                                 double *vx, double *vy, double *vz) {
    // velocity scale in pc/Myr
    double v0 = sqrt(G * Mtot / a);

    for (int i = 0; i < N; i++) {
        // local escape speed: v_esc = v0 * sqrt(2) * (1 + (r/a)^2)^(-1/4)
        double psi = v0 * sqrt(2.0) * pow(1.0 + (r[i]/a)*(r[i]/a), -0.25);
        double v;
        // rejection sampling on f(q) ∝ q^2 (1 - q^2)^(7/2)
        do {
            double q  = rand_uniform();
            double g1 = q*q * pow(1.0 - q*q, 3.5);
            double g2 = rand_uniform() * 0.1;   // approximate maximum
            if (g2 < g1) {
                v = q * psi;
                break;
            }
        } while (1);
        // random direction
        double u = rand_uniform();
        double cosTh = 1.0 - 2.0*u;
        double sinTh = sqrt(1.0 - cosTh*cosTh);
        double phi = 2.0 * PI * rand_uniform();
        vx[i] = v * sinTh * cos(phi);
        vy[i] = v * sinTh * sin(phi);
        vz[i] = v * cosTh;
    }
}
```

Listing 3: Generating inital velocities

## B.2 Simulation

### B.2.1 Sequential

```
typedef struct {
    int id;
    double x,y,z;
    double vx,vy,vz;
    double m;
```

```
} Star;
```

Listing 4: Struct for data handling

```
for (int step = 0; step <= nsteps; step++) {
    // compute accelerations by gravitational force
    for (int i = 0; i < N; i++) {
        ax[i] = ay[i] = az[i] = 0.0;
        for (int j = 0; j < N; j++) if (i != j) {
            double dx = S[j].x - S[i].x,
                   dy = S[j].y - S[i].y,
                   dz = S[j].z - S[i].z;
            double r2     = dx*dx + dy*dy + dz*dz + eps*eps;
            double inv_r3 = 1.0 / (r2 * sqrt(r2));
            double f      = G * S[j].m * inv_r3;
            ax[i] += f * dx;
            ay[i] += f * dy;
            az[i] += f * dz;
        }
    }
    // add acceleration to velocity and calculate movement
    for (int i = 0; i < N; i++) {
        S[i].vx += ax[i] * dt;
        S[i].vy += ay[i] * dt;
        S[i].vz += az[i] * dt;
        S[i].x  += S[i].vx * dt;
        S[i].y  += S[i].vy * dt;
        S[i].z  += S[i].vz * dt;
    }
    [...]
}
```

Listing 5: Movement simulation of the SimpleSequential Solution

```
for (int step = 0; step <= nsteps; step++) {
    for(int i = 0; i< N; i++)
    {
        ax[i]=ay[i]=az[i]=0.0;
    }
    // compute accelerations
    for (int i = 0; i < N; i++) {
        for (int j = i+1; j < N; j++){
            double dx=S[j].x-S[i].x,
                   dy=S[j].y-S[i].y,
                   dz=S[j].z-S[i].z;
            double r2 = dx*dx+dy*dy+dz*dz+eps*eps;
            double inv_r3 = 1.0/(r2*sqrt(r2));
            double f = G * S[j].m * inv_r3;
            ax[i] = f*dx;
            ay[i] = f*dy;
            az[i] = f*dz;

            f = - G * S[i].m * inv_r3;
            ax[j] = f*dx;
            ay[j] = f*dy;
            az[j] = f*dz;
        }
    }
    [...]
```

```
}
```

Listing 6: Force calculation of the OptimizedSequential Solution

### B.2.2 Parallel

```
for (int step = 0; step <= nsteps; step++){
    // compute accelerations
    #pragma omp parallel for schedule(static)
    for (int i = 0; i < N; i++) {
        [...]
    }
    // update velocities and positions
        #pragma omp parallel for schedule(static)
        for (int i = 0; i < N; i++) {
            [...]
        }
    }
    [...]
}
```

Listing 7: Changes to parallize with OpenMP

```
for (int step = 0; step <= nsteps; step++) {
    // compute accelerations for our chunk
    for (int i = local_start; i < local_start + local_N; i++) {
        ax[i] = ay[i] = az[i] = 0.0;
        for (int j = 0; j < N; j++) {
            if (i == j) continue;
            double dx = S[j].x - S[i].x;
            double dy = S[j].y - S[i].y;
            double dz = S[j].z - S[i].z;
            double r2 = dx*dx + dy*dy + dz*dz + eps*eps;
            double inv_r3 = 1.0 / (r2 * sqrt(r2));
            double f = G * S[j].m * inv_r3;
            ax[i] += f * dx;
            ay[i] += f * dy;
            az[i] += f * dz;
        }
    }

    // update velocities & positions for our chunk
    for (int i = local_start; i < local_start + local_N; i++) {
        S[i].vx += ax[i] * dt;
        S[i].vy += ay[i] * dt;
        S[i].vz += az[i] * dt;
        S[i].x  += S[i].vx * dt;
        S[i].y  += S[i].vy * dt;
        S[i].z  += S[i].vz * dt;
    }

    // share updated stars so every rank has the full array
    MPI_Allgatherv(
         MPI_IN_PLACE,
        0, MPI_DATATYPE_NULL,
        S, counts, displs, MPI_STAR,
        MPI_COMM_WORLD
    }
```

```
    [...]
}
```

Listing 8: Changes to parallize with MPI

```
const int    max_level = 16;          // maximum sub-steps per global dt
const double r_ref     = 15.0;        // [pc] outer scale radius
const double r_inner   = r_ref * 0.5; // inner scale radius

for (int step = 0; step <= nsteps; step++) {
    // --- assign time-step levels based on radius ---
    for (int i = local_start; i < local_start + local_N; i++) {
        double dx = S[i].x, dy = S[i].y, dz = S[i].z;
        double r = sqrt(dx*dx + dy*dy + dz*dz);
        if (r < r_inner)        level[i] = max_level;   // smallest dt = dt/4
        else if (r < r_ref)     level[i] = max_level/2; // dt/2
        else                    level[i] = 1;           // dt
    }

    // --- sub-cycling ---
    for (int sub = 1; sub <= max_level; sub++) {
        // compute & apply updates for stars whose level divides this sub-step
        for (int i = local_start; i < local_start + local_N; i++) {
            int li = level[i];
            // update when sub % (max_level/li) == 0
            if (sub % (max_level / li) != 0) continue;

            // compute acceleration for star i
            ax[i] = ay[i] = az[i] = 0.0;
            for (int j = 0; j < N; j++) {
                [...]
            }
            [...]
        }
        [...]
    }
    [...]
}
```

Listing 9: Calculations of distance to center

```
for (int step = 0; step <= nsteps; step++) {
    // Gather all stars onto each rank
    MPI_Allgatherv(S_local, local_N, MPI_Star,
                   S_all, counts, displs, MPI_Star, MPI_COMM_WORLD);

    // Build sector grid
    int    grid_size = GRID_SIZE;
    int    n_sectors = N_SECTORS;
    Sector *sectors = calloc(n_sectors, sizeof(Sector));
    for (int j = 0; j < N; j++) {
        int idx = sector_index(S_all[j].x, S_all[j].y, S_all[j].z,
                               grid_size, SECTOR_SIZE);
        if (idx < 0) continue;
        sectors[idx].mx   += S_all[j].x * S_all[j].m;
        sectors[idx].my   += S_all[j].y * S_all[j].m;
        sectors[idx].mz   += S_all[j].z * S_all[j].m;
```

```
        sectors[idx].mass += S_all[j].m;
        sectors[idx].count++;
    }

    // Compute accelerations
    for (int i = 0; i < local_N; i++) {
        ax[i] = ay[i] = az[i] = 0.0;

        // Direct summation
        for (int j = 0; j < N; j++) {
            if (S_local[i].id == S_all[j].id) continue;
            double dx   = S_all[j].x - S_local[i].x;
            double dy   = S_all[j].y - S_local[i].y;
            double dz   = S_all[j].z - S_local[i].z;
            double r2   = dx*dx + dy*dy + dz*dz + eps*eps;
            double invr3 = 1.0 / (r2 * sqrt(r2));
            double f    = G * S_all[j].m * invr3;
            ax[i] += f * dx;
            ay[i] += f * dy;
            az[i] += f * dz;
        }

        // Sector far-field
        int my_sec = sector_index(S_local[i].x, S_local[i].y,
                                  S_local[i].z,
                                  grid_size, SECTOR_SIZE);
        for (int s = 0; s < n_sectors; s++) {
           if (s == my_sec || sectors[s].mass == 0.0) continue;
           double sx = sectors[s].mx / sectors[s].mass;
           double sy = sectors[s].my / sectors[s].mass;
           double sz = sectors[s].mz / sectors[s].mass;
           double dx = sx - S_local[i].x;
           double dy = sy - S_local[i].y;
           double dz = sz - S_local[i].z;
           double r2   = dx*dx + dy*dy + dz*dz + eps*eps;
           double invr3 = 1.0 / (r2 * sqrt(r2));
           double f    = G * sectors[s].mass * invr3;
           ax[i] += f * dx;
           ay[i] += f * dy;
           az[i] += f * dz;
        }
    }
    [...]
}
```

Listing 10: Logic of Sector based calculations