Seminar:
Practical Course on High-Performance Computing

Report on
Simulating Multi-Star Systems Using Python (and maybe C++)

Ughur Mammadzada
ughur.mammadzada@stud.uni-goettingen.de
Universität Göttingen

Supervisor: Martin Paleico

30 September 2024

Contents

1 Introduction **2**

2 Methodology **2**
2.1 Physics . . . 2
2.2 Space Chunking . . . 3
2.3 Barnes-Hut / OctTree . . . 4
2.4 Recordings . . . 5

3 Implementation **6**
3.1 Space Chunking . . . 6
3.2 OctTree . . . 7

4 Evaluation **8**
4.1 Space Chunking . . . 8
4.2 OctTree . . . 9

5 Discussion **10**

6 Conclusion **10**

A Appendix **11**
A.1 Calculation of execution time on Xeon E5-2650 v4 . . . 11
A.2 OctTree_v2 Sync Stage . . . 11
A.3 OctTree_v2 Graphs . . . 13
A.4 Links . . . 16

1 Introduction

This report covers the work conducted during the practical seminar **"Practical Course on High-Performance Computing"** at the University of Goettingen during Summer Term 2024.

The practical project aims to simulate an N-body (Multi-Star) system using Python and parallelize it using OpenMPI.

Simulating N-body systems is computationally expensive, especially in gravitational simulations, where each body must interact with all other bodies. Given the number of bodies n, the complexity of the calculations is $O(n^2)$.

While the parallelization of basic body-body interactions is not difficult and partially reduces the wall-clock time, it does not solve the complexity of the calculations.

To solve the complexity of the calculation such simulations often use approximations, such as Barnes-Hut algorithm with OctTrees [1] or Space Chunking, which have complexities of $O(n \log n)$ and $O(n\sqrt{n})$ (optimal case) respectively.

To put into perspective, if we are doing our computations on a Xeon E5-2650 v4, and don't have any overhead and memory latencies, to simulate 1M bodies in ideal case we would need (calculations in A.1):

- $O(n^2)$: $\sim$ 1 hour 15 minutes
- $O(n\sqrt{n})$: $\sim$ 5 seconds
- $O(n \log n)$ $\sim$ 0.09 seconds

And for visual comparison Fig. 1 shows how number of operations increase with number of bodies.

In this project 2 different simulation approaches were executed. Space Chunking and an OctTree. As expected OctTree was outperforming Space Chunking method, however due to more complex sturcture the MPI communications were harder to implement. Furthermore, initial attempt to make OctTree simulation parallel was failed as the sequential code was faster due to communication overhead. This partially was resolved by altering the data type and making the communication packages lighter.

This report will first discuss these two approaches in Ch. 2 and how they will be parallelized. It will also explain the decisions made and the reasoning behind them. Then in Ch. 3 the report will explain the simulation code and the MPI communications.

The simulations were run on the Scientific Computing Cluster (SCC) at GWDG[2]. The results of the simulations and the evaluation of execution will be presented in Ch. 4. This will be followed by Discussion and Conclusion.

2 Methodology

2.1 Physics

To compute the force between two bodies with masses m_1 and m_2, and positions (x_1, y_1, z_1) and (x_2, y_2, z_2), the following equation from classical physics is used:

$$\vec{F} = G\frac{m_1 m_2}{r^2}\hat{r} \tag{1}$$

where:

- G is the gravitational constant.
- r is the distance between the two bodies, calculated as $r = |\vec{r_2} - \vec{r_1}| + \epsilon$.
- $\vec{r_i} = \overrightarrow{(x_i, y_i, z_i)}$
- $\hat{r}$ is the unit vector in the direction of the force, calculated as $\hat{r} = \frac{\vec{r_2} - \vec{r_1}}{|\vec{r_2} - \vec{r_1}|}$.

After we have the force $\vec{F}$ to calculate how the a body moves from the position (x_t, y_t, z_t) at time t to $(x_{t+dt}, y_{t+dt}, z_{t+dt})$ during the time dt we have to calculate the velocity $\vec{v}_t$ it has using the force as acceleration:

$$\vec{v}_{i,t+dt} = \vec{v}_{i,t} + \frac{\vec{F}_i dt}{m_i} \tag{2}$$

here $\vec{F}_i$ is the force applied to the body i.

And with the velocity we can calculate:

$$\vec{r}_{i,t+dt} = \vec{r}_{i,t} + \vec{v}_{i,t+dt}dt \tag{3}$$

To make the calculations smoother, Leapfrog Integration[3] can be used:

$$\vec{v}_{i,t+\frac{dt}{2}} = \vec{v}_{i,t} + \frac{\overrightarrow{F_i(\vec{r}_{i,t})}\frac{dt}{2}}{m_i} \tag{4}$$

$$\vec{r}_{i,t+dt} = \vec{r}_{i,t} + \vec{v}_{i,t+\frac{dt}{2}}dt \tag{5}$$

$$\vec{v}_{i,t+dt} = \vec{v}_{i,t} + \frac{\overrightarrow{F_i(\vec{r}_{i,t+dt})}dt}{m_i} \tag{6}$$

Here the force with the current position of the object is calculated, however the velocity update is made by half step. The half step velocity $\vec{v}_{i,t+\frac{dt}{1}}$ is then used to calculate the position at the time dt, and that position is then again used to calculate the force for the complete velocity update. The complete velocity is not used in this iteration, and in the next iteration a half step is added to it, so in practice this velocity value is never used, however the movements are more smooth.

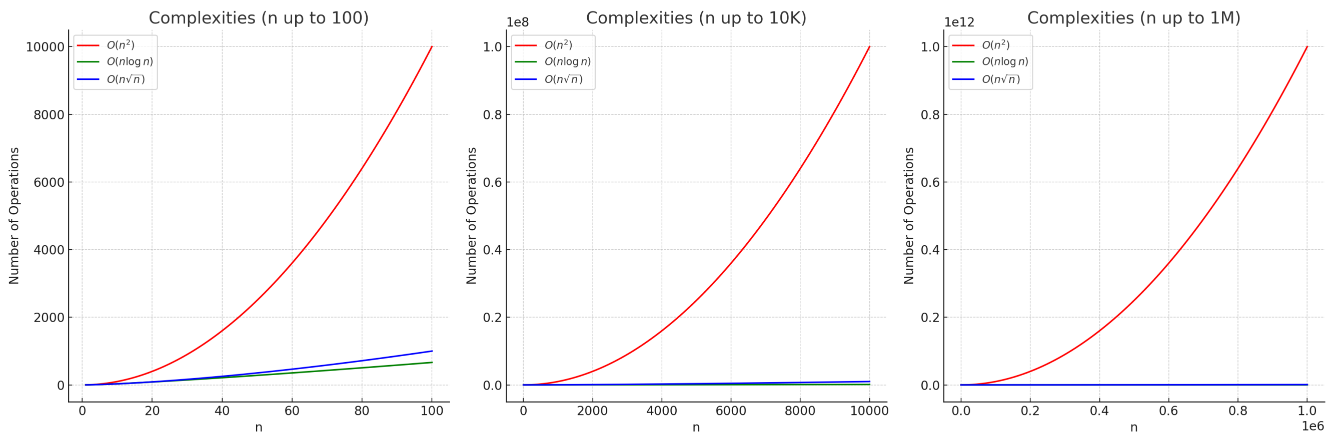

Figure 1: Graphs of different complexities.

2.2 Space Chunking

If the simulation consisted of only body-body interactions it would be too time consuming, as discussed. We could make it parallel by broadcasting the bodies to workers in equal amounts. Each worker still would had to have the copy of the entire array or list of objects, since any object interacts with any other objects. But with 4 workers and 100 objects instead of calculating how each body interacts with other 99 we would calculate how 25 objects interact with 99 other on each worker, making the calculations 4 time faster. With little bit more effort we could divide the other part of the calculation as well, so that one worker calculates 25 objects versus 45 other, and the other worker calculates 25 same objects versus 44 other remaining objects.

In space chunking method however that is not the case. In this kind of simulation main objective is to approximate distant interactions by replacing bunch of far away objects with a "phantom" object that has big mass as the sum of all masses of objects in that direction and their centre of mass. To make such simulation more accurate, instead of directions equal chunks of space are used.

This way if before we had to calculate $100 \times (100 - 1) = 9.900$ interactions ($n^2 = 100^2 = 10.000$), now we have to ideally calculate only $4(\times 25 \times (25 - 1) + 3) = 2404$ interactions ($n\sqrt{n} = 100 \times 10 = 1000$). The reason why there is 2.5 times mismatch with the complexity is because the actual complexity is $O(2n\sqrt{n})$.

If we consider that we have n objects, for the optimal chunking we need to have $\sqrt{n}$ chunks. The reason is in ideal setting n objects are equally distributed between m chunks.

If $m = n$ then instead of calculating n to n object object calculations the calculation will essentially become n chunks to n chunks with the same complexity $O(n^2) = O(m^2)$. If $m = 1$ then all object are in the same chunk

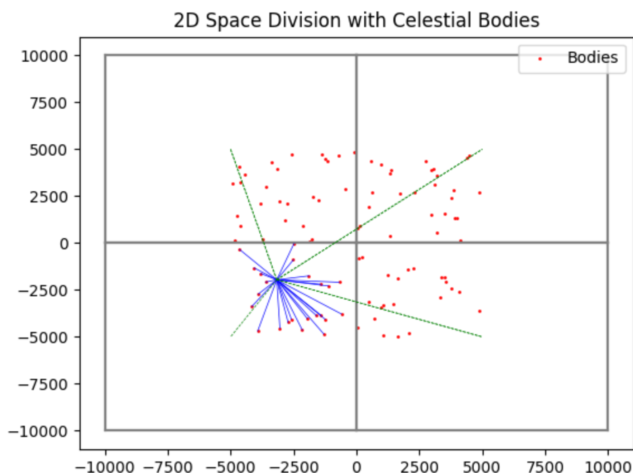

Figure 2: 100 bodies in a chunked space. Each body interacts with all bodies in own chunk and with "huge far-away phantom" objects represented as centres of mass of bodies in other chunks and their total masses.

and we again have the same $n \times n$ calculations. So ideally we need something in between. And the middle is:

$$m = \sqrt{n} \tag{7}$$

$$O(\frac{n^2}{m} + nm) = O(n\sqrt{n} + n\sqrt{n}) = O(2n\sqrt{n}) \tag{8}$$

And that is why number of calculations is twice as much as expected.

When it comes to parallelization, with this approach we can assign each chunk to a worker. The worker will perform body-body interactions between objects inside the chunk and use total masses to approximate other chunks. In such set up each worker will only need to have list of object that are in the assigned chunk, and gather "phantom" objects from other workers. However this

might introduce communication overhead as to proceed the worker will be waiting for 2 number from all other workers. Furthermore moving objects between chunks would introduce another communication challenge.

To make the communication more simple another approach can be used. In this project's Space Chunking the synchronization steps are:

1. All workers have all chunks and objects in their memory
2. After each iteration all workers get updated object from all other workers
3. Based on updated positions each worker recreates the chunked space

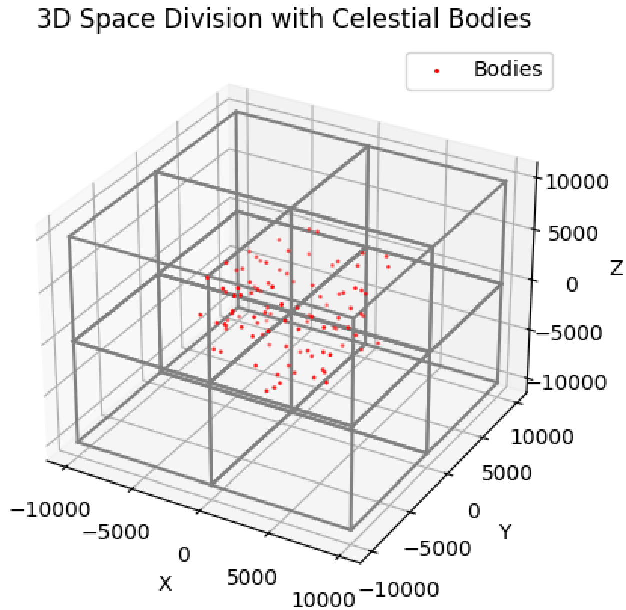

Figure 3: 100 bodies in a chunked 3D space.

However, due to space being chunked on each worker independently to reduce communication, floating point error sometimes causes some chunks to have slightly different centre. Since the program uses the chunk centres to navigate the space, identify the chunks and assign them to workers (e.g. rank 1 is responsible for (x, y, z) chunk) workers might not be able to find their chunks. To eliminate this problem chunk centre map with approximate centres is present on all workers and the centres of chunks are rounded by a given precision (*default 10*).

This parallelization is expected to scale well and almost linearly, however number of workers is limited by number of chunks. It would have been possible to chunk the space depending on the number of workers, but with increasing number of workers, and not increasing number of objects the complexity of calculations would grow as discussed previously. For each simulation set up (number of object, number of chunks), there is optimal number of workers, which was calculated experimentally and will be presented in Ch. 4.

2.3 Barnes-Hut / OctTree

Barnes-Hut algorithm using OctTrees has the approximation mechanism similar to the Space Chunking method.

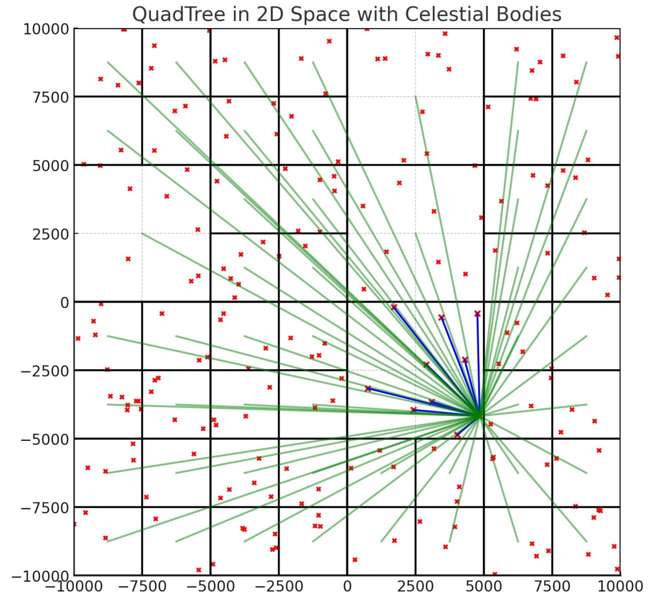

Figure 4: 100 bodies in QuadTree. Max capacity of every branch is 10 bodies. 52 leaf branches.

In a Figure 4 A QuadTree is shown, where each branch can subdivide into 4 branches. When an object is inserted into the tree, the tree checks if it has branches, and if so tries to pass the object to branches. This happens recursively, so every branch will try to add the object. If one of the branches succeed, the process stops. The leaf branches only have objects. Object is added if its coordinates are withing the branch boundaries. When branch capacity is reached (here 10 objects), the leaf branch creates 4 own branches and distributes the objects.

It can be seen that there are much more "chunks" and body to centre of branch interactions. This depends hugely on max capacity. If max capacity is 25 the QuadTree would look more like the Fiqure 2. The difference is, since number of object per branch is limited, the distribution of objects into chunks of spaces is more balanced.

Since number of "chunks" or branches is higher, and number of objects in the same space chunk is smaller this essentially means higher approximations of interactions. Here with 100 objects and max capacity 10 each object has to interact with 9 object in own branch and ~51 (in

this case, the exact number depends on the central object and the current structure of the tree) phantom objects totalling in ~60 interactions per object and $100\times \sim 60 \simeq 600$ interactions and it is close to the theory:

$$100 \times \log 100 = 100\times \sim 6.64 = \simeq 664 \tag{9}$$

The parallelization of this approach however is not straightforward. In Space Chunking the worker had to navigate the space using a map of chunk centres. And after each iteration the chunk centres are distributed between workers. Workers always know which chunks they are responsible for.

In an OctTree after each iterations branches will be pruned or created, and instead of using map `(x, y, z) => chunk_id` the workers traverse the tree (`root => 8×branches`).

Of course the branch has its own centre as its position, however it is mainly used to determine if objects belong in the range from centre to the border of the branch. Mapping the tree would require additional mechanism to keep track of root-branch relationships and tree reconstructions. It would also lower the efficiency of look-ups as the tree doesn't have "search by coordinates" functionality, and traversing it is easier using root-branch relationships.

All these means that while navigating the tree is easier programmatically, it is harder to assign branches to workers and let them communicate the tree structure between themselves.

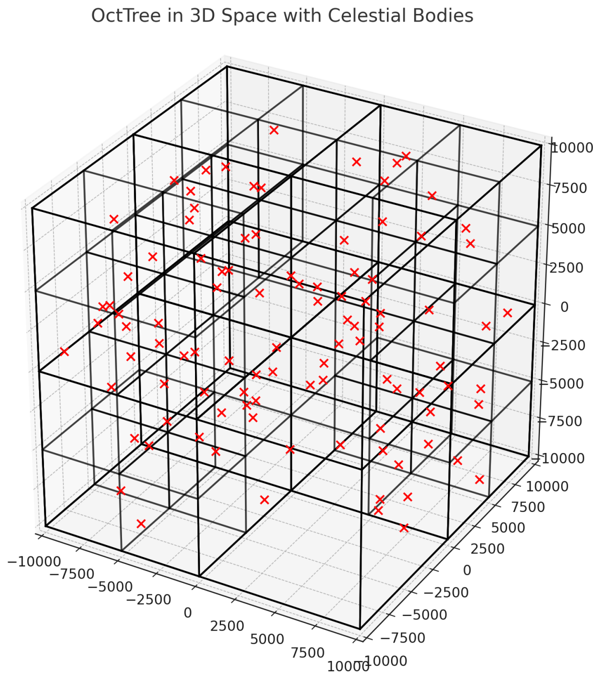

Figure 5: 100 bodies in OCtTree. Max capacity of every branch is 10 bodies. 42 leaf branches.

While the initial approach was to broadcast the tree to the workers and let them calculate which branches they should be using, pruning the tree when objects leave branches would be communication heavy, as first the workers have to pass the objects they have updated to the root worker, which would prune the tree and broadcast it again.

In a simpler version, workers send the updated objects to all workers, not just the root, and each worker reconstructs the tree, instead of looking for branches, pruning them if objects leave, and adding them when objects are inserted.

The first version of the OctTree defines the bodies as objects of the CelestialBody class, while the second one uses numpy arrays. As expected version two was better communication wise and computation speed wise. However, the communication overhead was still present in both versions.

The OctTree has max-capacity of branches and max-depth (how many generations it can have). To calculate max-depth we use:

$$maxDepth = \log_8 n \tag{10}$$

and for max-capacity:

$$maxObject = \frac{n}{8 \times maxDepth} \tag{11}$$

Here 8 is the base because each branch has 8 branches.

2.4 Recordings

The simulation generates several CSV files and also prints out some setting, environment and debugging info, which also can be redirected to a file. All CSVs are comma separated.

Among the generated CSV files the most important is `results_<i>.csv`, where `i` is a file index, in case the same save directory was used several times. This file stores the simulation progress and its state in the following format:

- Time - the current time inside the simulation
- Object_ID - to identify the objects
- Mass
- Position_x,Position_y,Position_z
- Velocity_x,Velocity_y,Velocity_z
- Worker - to keep track which workers did the update

This file can then be used to animate the system, using the positions.

Another generated file is `iterationsFile_<i>.csv` which keeps track on simulation computation times.

- Time - the current time inside the simulation
- Iteration_start,Iteration_end,Iteration_duration
- Sync_start,Sync_end,Sync_duration

Here the iteration is part of the computation where the forces, velocities and new positions are calculated, while sync stage is mainly for MPI versions to keep track how much time it requires for all workers to sync with all other workers their updates.

The `resourcesFile_<i>.csv` is used to keep track of memory usage (recorded by `psutil` on each worker), and `cprofileFile_<i>.csv` is `cProfile` stats ordered by time.

The existing performance meter tools on the cluster were troublesome to use, or would need the user to be interactive. While for debugging this is useful, the simulations take days to hours and days to finish. Thus from practical perspective those tools were not an option.

3 Implementation

3.1 Space Chunking

Space chunking was the easier approach between 2 in terms of communication, but was harder to navigate the space itself. To not complicate the calculations the objects were represented as arrays for each of their properties. The objects are assigned to chunks with:

```
def assignChunks(positions: np.ndarray, chunkCentres: list[np.ndarray], step: float) -> dict:
    chunks = {tuple(centre): [] for centre in chunkCentres}
    for i, position in enumerate(positions):
        distances = np.linalg.norm(chunkCentres - position, axis=1)
        closest_chunk = chunkCentres[np.argmin(distances)]
        chunks[tuple(closest_chunk)].append(i)
    return chunks

def chunkSpace(positions: np.ndarray, gridSize: int, spaceRange: tuple[float, float]) -> tuple[dict, list[np.ndarray], float]:
    chunkCentres, step = calculateChunkCentres(gridSize, spaceRange)
    chunks = assignChunks(positions, chunkCentres, step)
    return chunks, chunkCentres, step
```

Here, instead of looking if the object is inside of chunk using its centre and borders (which might alter from worker to worker), the closest chunk is selected, and the ID of the object (matches the index in positions array) is assigned to the chunk. When chunks are assigned to workers, they also get list of object IDs, which they can use to access other properties like mass, velocity and position from the respective arrays.

Since the data types being transferred are numpy arrays, the communication overhead is reduced:

```
if rank == 0:
    ...
    data = (positions, velocities, masses, chunks, chunkCentres, step)
else:
    data = None

positions, velocities, masses, chunks, chunkCentres, step = comm.bcast(data, root=0)
```

Of course the initial broadcast could be improved, however this is set up broadcast happening before the simulation loop begins so it was decided to keep it simple.

In the main loop the communication happens only after the iteration is done and serves synchronization purpose:

```
localNextPositionsSizes = np.array(len(localNextPositions), dtype='i')
allLocalNextPositionsSizes = np.zeros(size, dtype='i')
comm.Allgather(localNextPositionsSizes, allLocalNextPositionsSizes)

displacements = np.cumsum(allLocalNextPositionsSizes) - allLocalNextPositionsSizes
totalSize = np.sum(allLocalNextPositionsSizes)

allNextPositions = np.empty((totalSize, 3), dtype='d')
allFinalVelocities = np.empty((totalSize, 3), dtype='d')

comm.Allgatherv([localNextPositions, MPI.DOUBLE], [allNextPositions, (allLocalNextPositionsSizes*3, displacements*3), MPI.DOUBLE])
comm.Allgatherv([finalVelocities, MPI.DOUBLE], [allFinalVelocities, (allLocalNextPositionsSizes*3, displacements*3), MPI.DOUBLE])

positions = allNextPositions
velocities = allFinalVelocities
```

Here the workers first communicate the sizes of the packages they are about to transfer. Each worker then computes the displacements so that the result arrays do not mess up object orderings. Then the main packages are communicated, and then the local copies of the global parameters are updated.

Since the local arrays of workers which keep the object parameters that the workers are responsible for are shorter, when accessing the list of object IDs from the chunk, those IDs mismatch with the IDs on the global arrays. To eliminate this problem another map is used, which maps global IDs to local IDs:

```
globalToLocal = {idx: i for i, idx in enumerate
    ([i for chunk in localChunks.values() for i
    in chunk])}
```

Here `idx` is the actual global ID of the object and `i` is the ID of the object on the local chunks. The chunk might have objects [4, 6, 9] but in local array which will have length 3 those IDs will be mapped to [0, 1, 2] as they should be to eliminate out of bound errors.

3.2 OctTree

Initially OctTree based simulation was planned to rely on OOP:

```
# CelestialBody interface
class CelestialBody:
    def __init__(
        self,
        objectId: int=0,
        position: np.ndarray=None,
        mass: float=0.0,
        velocity: np.ndarray=None,
        G: float = parameters.G,
        softening: float = parameters.
            distanceEpsilon
    ):
    def move(self):
    def calculateNewVelocity(self, forceApplied:
        np.ndarray, dt: float):
    def calculateNextPosition(self, dt: float):
    def calculateForce(self, anotherObject: '
        CelestialBody', leapfrogStep: bool =
        False) -> np.ndarray:
    def setWorker(self, worker: int = 0) -> None
        :

# OctaTree interface
class OctaTree:
    def __init__(
        self,
        position: np.ndarray = None,
        size: float = 0.0,
        maxObjects: int = 1,
        depth: int = 0,
        maxDepth: int = 10,
        softening: float = parameters.
            distanceEpsilon
    ):
    def isLastBranch(self) -> bool:
    def belongs(self, position: np.ndarray) ->
        bool:
    def subdivide(self) -> bool:
    def addObject(self, objectToAdd:
        CelestialBody) -> bool:
    def addToBranch(self, obj: CelestialBody) ->
        bool:
    def updateMassAndCenterOfMass(self):
    def getApproximateBodies(self, target:
        CelestialBody, theta: float) -> List[
        CelestialBody]:
```

The tree would store Celestial Bodies (or rather the references to those objects) in the objects array. And the branches array would be list of references to recursively created branches, which also belong to the `OctaTree` class.

Almost all functions in the OctTree are recursive in their nature and mainly targeted at the leaf branches. There was an attempt to keep only leaf branches, however the execution was complicated and the benefit was not clear so the idea was dropped.

The problem with this approach is, while communicating the tree via MPI is acceptable for construction integrity, the bodies as Python objects are too heavy. This caused massive overhead. While the sequential part completed a simulation in 23 hours, the parallel version was only at around 25% of the same simulation. The communication concept however was working. The next step is to refactor the simulation to have only the tree as an object.

```
# OctaTree_v2 interface
class OctaTree_v2:
    def __init__(
        self,
        position: np.ndarray = None,
        size: float = 0.0,
        maxObjects: int = 1,
        depth: int = 0,
        maxDepth: int = 10,
        softening: float = parameters.
            distanceEpsilon,
        positions: np.ndarray = None,
        masses: np.ndarray = None,
        root: 'OctaTree_v2' = None,
    ):
    ...
    # Same as previous Tree
    ...
    def setPossitionsAndMasses(self, positions:
        np.ndarray, masses: np.ndarray):
```

In the new OctTree the tree instead of saving the reference to objects in a list would save indexes of objects, similar to space chunking. It would also save references to position and masses arrays, where it can get parameters of the object in order to calculate total mass, or check if the object belong to the branch.

Since after broadcast the position and masses on each worker will be set to their local memory address, positions and masses in the OctTree are prevented from communication, and after tree broadcast each worker sets the positions and masses reference in their local tree.

To distribute the work between workers the leaf branches of the tree are collected and scattered between workers in equal amounts, similar to Space Chunking.

While in space chunking the distribution could have been improved by assigning the chunk with the least objects to the worker that already got the chunk with the most object, or use other balancing method, in case of an OctTree since all branches have the same capacity the workload is distributed automatically.

```
if rank == 0:
    data = (objectIds, positions, velocities,
        masses)
    timeParameters = (dt, timeline,
        outputInterval)
else:
```

```python
5     data = None
6     timeParameters = None
7
8 objectIds, positions, velocities, masses = comm.
      bcast(data, root=0)
9
10 nextPositions = np.zeros((totalObjects, 3))
11 workers = np.zeros(totalObjects, dtype=int)
12
13 dt, timeline, outputInterval = comm.bcast(
      timeParameters, root=0)
14 theta = comm.bcast(theta, root=0)
15 ...
16 ...
17 tree = comm.bcast(tree, root=0)
18 assignedBranches = comm.scatter(assignedBranches
      , root=0)
19
20 # Nodes have different addresse for arrays
21 tree.setPossitionsAndMasses(positions, masses)
```

After the main calculations are done, each worker saves in the `updatedBodies` a list of indexes of objects they have modified. The new positions are saved in `nextPositions`, where indexes from `updatedBodies` are filled, and the rest are 0s, as another worker is responsible for those. `workers[updatedBodies]` saves the rank ID of the worker at the object ID index in the array. Of course those arrays have lots of zeros, and could have been shorter, similar to Space Chunking. However workers do not communicate the 0s, only the updated parts.

Here local arrays are created just before the communication, and the work is done on global arrays. Global meaning those are the arrays of all n bodies. In comparison Space Chunking simulation was designed to have both global and local arrays. The code for `OctaTree_v2` synchronization step can be found in A.2.

The simulation that in sequential mode took 23 hours and in parallel_v1 toke 24 hours for 25%, with communication and data types replaced in parallel_v2 took 11 hours. For both parallel version 8 workers were used.

With such progress it was decided to focus on scaling results of the version 2. However to make the evaluation fair the sequential version of version 2 was written as well. Soon after it was discovered that the pattern repeats and simulation overall takes longer with increasing number of workers when compared to the new sequential method (details in 4.2) (Fig. 12 see in A.3).

One of the main reasons being the communication overhead when broadcasting the tree to the workers from the root. It was decided to build the tree on each worker instead to reduce the communication. After doing that it was tested to see if the tree structures match on each worker and they did.

The cProfile stat analysis on the next run showed that there are to many recursive calls to the tree functions (mainly `belongs` and `updateMassAndCenterOfMass`) (Fig. 14, 15). More information will be provided in Evaluation.

Further optimization was done to decrease number of calls. For example the simulation that would do 279,949 `belongs` calls now was reduced to 175,449. And for `updateMassAndCenterOfMass` it was reduced from 1,015,408 to 13,700. After those optimization the scaling test was run again.

4 Evaluation

4.1 Space Chunking

In the Figure 6 the graph shows the execution times per iteration for the Space Chunking simulation with fixed grid size and increasing number of objects. It is following the expected pattern.

The Figures 7 and 8 show the optimal number of workers depending on number of object and constant grid size.

It also can be seen that with 16 workers the computations that in Figures 6 took more than 200 seconds in sequential mode is taking 2 times less with 2 workers and 10 times less with 16 workers.

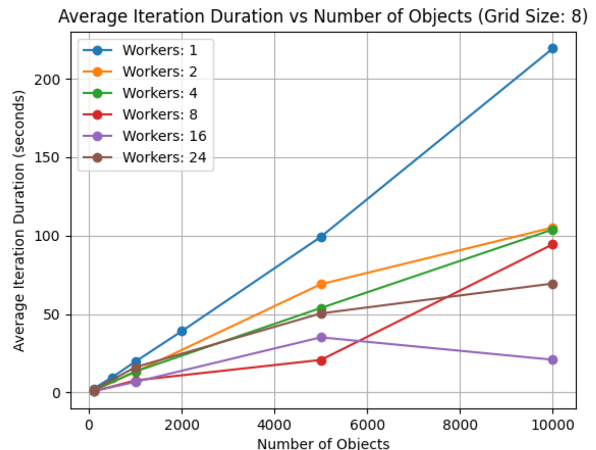

Figure 6: Space Chunking simulation. The time spent on each iteration depending on the number of objects. Grid size is set to 8 (512 chunks).

However the scaling is not that simple. It depends on both number of objects and the grid size. In Figure 9 it can be seen that, for 1000 objects increasing number of workers does reduce the computation time. However increasing number of chunks causes the performance to drop. For 1000 objects the optimal grid size is 3 or 4, 27-64 chunks ($\sqrt{1000} \simeq 31$).

However that also doesn't mean there is no optimal number of workers for such set up. As Figures 7 and 8 show, even with an non-optimal grid size (here 8, making

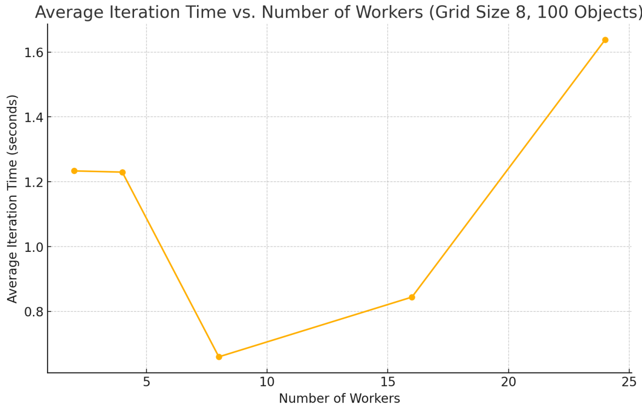

Figure 7: Space Chunking simulation with 100 objects and grid size 8 the optimal number of workers is 2.

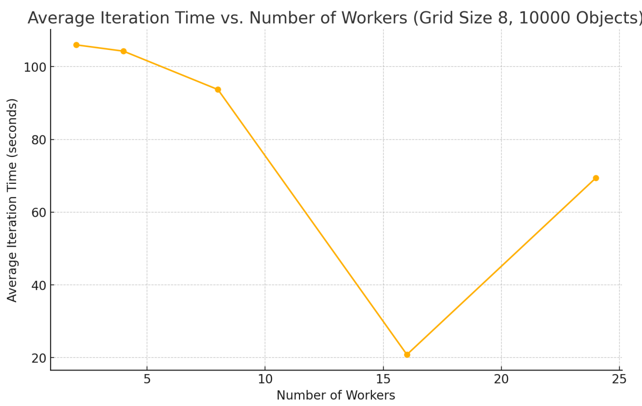

Figure 8: Space Chunking simulation with 10000 objects and grid size 8 the optimal number of workers is 16.

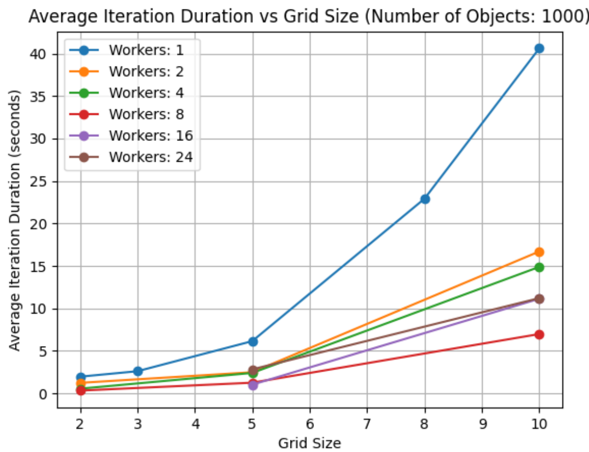

Figure 9: Space Chunking simulation. The time spent on each iteration depending on the grid size. Number of objects is set to 1000.

512 chunks), optimal number of workers can be found (2 and 16 here respectively).

4.2 OctTree

For the OctTree the evaluation will focus on the version 2, as version 1 was taking 4 times longer that the sequential version with 8 workers. However, as said, this trend showed itself when comparing version 2 parallel to sequential version 2 as well (A.3).

After the optimization was done for the MPI communication, the overall performance didn't change for parallel execution. Only after the OctTree functions were optimized (for both sequential and parallel version), the time difference did decrease, however the parallel version still always was slower than the sequential version (Fig. 10).

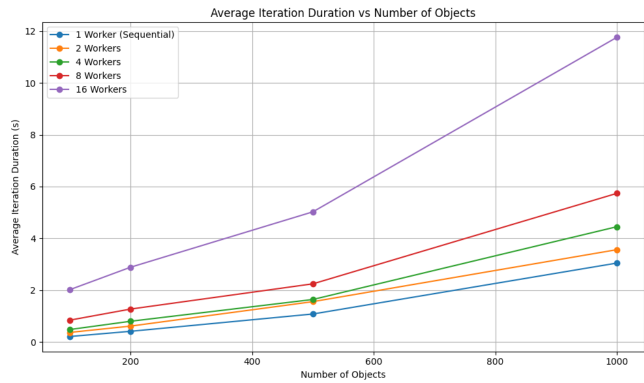

Figure 10: OctTree simulation, comparison of execution time with increasing number of objects for different number of workers. **After Optimization**.

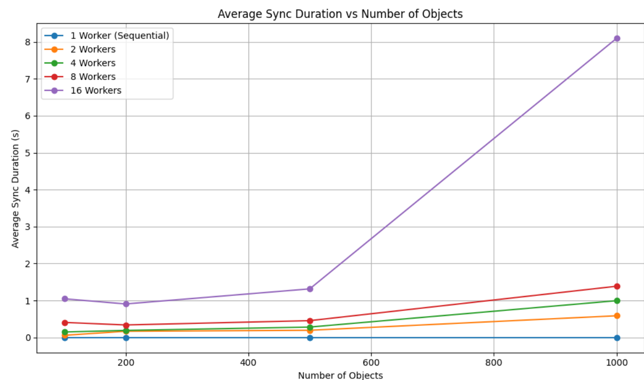

Figure 11: OctTree simulation, comparison of synchronization time with increasing number of objects for different number of workers. **After Optimization**.

As can be seen on Figure 11 the synchronization time increases as number of workers increase. The second round of cProfile showed that the communication during the synchronization process is now the main reason for the increased run time (Fig. 17, 17 see in A.3).

As can be seen, the functional calls of the OctTree remain at around 100 seconds on average, however the

`Allgatherv` increases with increasing number of workers. It was checked if the workers gather all object properties or only the updated ones, and they did collect only the updated objects.

During the synchronization each worker sends only the updated positions and velocities that it have updated. This is similar to the synchronization in the Space Chunking method. However in the OctTree simulation each workers additionally sends the list of updated object IDs and list of ranks, to identify which worker updated which object later during the debugging. To reduce amount of data communicated those parameters were removed and the communication was made similar to Space Chunking method. However this did not affect the end result too much. No further optimizations were conducted.

Besides of the execution time data, memory consumption was measured as well. The measurement showed that with increasing number of objects memory consumption does increase slightly. This is the expected behaviour as all workers always have the total set of data to not communicate them each time (Fig. 13 in A.3).

5 Discussion

This project initially had more developers, however the majority of the project, due to circumstances, had to be completed by a single developer. To compensate for this the functionalities that were not main part of the simulation and were not the main parallelization objective were decided to be outsourced to Generative Artificial Intelligence (ChatGPT)[4]. Outsourced code includes:

- `animator-chatgpt.py` - Generated by ChatGPT debugged by developer
- `visuals.py` - Generated by ChatGPT, almost no changes done by the developer, the code is mostly deprecated
- `gatherCProfileStats` - The function is generated by ChatGPT, only style edits were done by the developer

During the first part of the project, OctTree based simulation was first put on hold and Space Chunking was developed in order to have working simulation due deadlines. The OctTree at the time had sequential simulation working, however the communication was challenging, as described in previous chapters.

The problem was not the lack of idea how to communicate, but the desire to not accept the obvious answer - all workers should have global system data. The initial attempt to develop the simulation (both OctTree and Space Chunking) was focused on communicating only the relevant objects to the workers, and then communicate "phantom" objects between them, which would be the approximation of local subset of objects.

From the memory efficiency perspective this would have been a better approach, however even if it worked, the workers would have been waiting for small amount of data each time to be communicated, those introducing an overhead.

Between memory efficiency and wall clock efficiency the priority was given to the second one. This also simplified the communication mechanism.

The correct visualization is still missing. The current visualization relies on plotting a 3D scatter plot and making the background black. With more time and effort and more nicer animation could have been developed. We can make the animation rotate around z axis or have the trajectories of objects drawn as well. Furthermore the animation can be made parallel as well.

However this need additional time for testing. While the purpose of the project is mainly to write parallel calculations, we still want the calculations to be as accurate as possible.

As the title says, initially it was planned to translate the simulation into C++ after it is feature complete. This was not accomplished. However from the beginning the codebase was written and designed in a way that makes it simple for translation. For example, the only python specific library which was used for computations is numpy. The amount of external libraries or libraries that are hard to replace in C++ or write the functionality manually are reduced to almost none.

As explained before, the use of performance measurement tools preinstalled on the cluster was a challenge. To compensate the measurements were implemented into the simulation (OctTree only).

The project by design involves using MPI and doing the computations on CPUs. However, this kind of simulation is more appropriate to be parallelized on GPUs. This would eliminate major part of the communication latency, and the number of available cores would be much higher. The computation speed will probably be fast enough for a real-time animation as well.

6 Conclusion

The project work involved developing multi-star simulation program and then parallelize it using MPI. 2 kinds of simulations were developed and parallelized and optimized for MPI. The scaling tests for the simulators were conducted.

The parallelization was not always the best choice for

all code parts, but for majority of the cases the parallelization outcomes met the expectations by improving the calculation time. However the communication overhead was not always resolved as in case of OctTree simulation.

While OctTree simulation as expected was more efficient than the Space Chunking method, it did not scale. On the other hand Space Chunking overall is slower than OctTree simulation, however with correct chunking size and number of workers it scales well with the increasing number of objects.

A better approach for OctTree simulation as conclusion would have been first to chunk the space into number of chunks equal to number of workers. After that each worker would only get a subset of objects that are in those chunks, and build local OctTree, that are bounded into those chunks. Each worker also should have a list of total masses of other workers. However the only communication after this set up would be whenever after the iteration an object leaves the chunk that the worker is responsible for. The worker should then pass the object(s) to the worker that is responsible for the receiving chunks. And each receiving worker would broadcast their new total mass to everyone. This could be future optimization. This approach takes advantage of both methods and reduces the communication.

Different optimization techniques were used with varying success. Overall the project goals were not completely reached, as the OctTree parallelization was not optimized.

A Appendix

A.1 Calculation of execution time on Xeon E5-2650 v4

:

The Xeon E5-2650 v4 has clock speed 2.2GHz (2.2×10^9 cycles / second), if we assume there is no latency, overhead and 10 cycles are needed for 1 operation:

$$\text{Operations per second} = \frac{\text{cycles per second}}{\text{cycles per operation}} \tag{12}$$

$$OPS = \frac{2.2 \times 10^9}{10} = 2.2 \times 10^8 \tag{13}$$

with $n = 10^6$ (1M) number of operations:

- $O(n^2) = 10^{6+6} = 10^12$
- $O(n\sqrt{n}) = 10^{6+3} - 10^9$
- $O(n \log n) =\sim 19.93 \times 10^(6) = 1.993 \times 10^7$

To calculate time takes:

$$t = \frac{\text{number of operations}}{\text{operations per second}} \tag{14}$$

$$t_{n^2} = \frac{10^12}{2.2 \times 10^8} \simeq 4545.5 \text{ seconds} \simeq 1.26 \text{ hours} \tag{15}$$

$$t_{n\sqrt{n}} = \frac{10^9}{2.2 \times 10^8} \simeq 4.55 \text{ seconds} \tag{16}$$

$$t_{n \log n} = \frac{1.993 \times 10^7}{2.2 \times 10^8} \simeq 0.09 \text{ seconds} \tag{17}$$

A.2 OctTree_v2 Sync Stage

:

```
1
2 # Create Shorte arrays with only updated
      parameters
3 updatedPositions = nextPositions[updatedBodies]
4 updatedWorkers = workers[updatedBodies]
5
6 # Start sync process (No simulation, only
      communication)
7 if rank == 0:
8     syncStartTime = time.time()
9
10 # Each worker computes the communication size
      itself
11 updatedPositionsFlat = updatedPositions.flatten
      ()
12
13 localSendCounts = np.array(len(updatedBodies),
      dtype='intc')
14 sendCounts = np.empty(size, dtype='intc')
15 comm.Allgather([localSendCounts, MPI.INT], [
      sendCounts, MPI.INT])
16 recvCounts = sendCounts.copy()
17 sendDisplacements = np.insert(np.cumsum(
      sendCounts), 0, 0)[0:-1]
18 recvDisplacements = sendDisplacements.copy()
19
20 # To take into account positions as vectors
21 positionsSendCounts = sendCounts[rank] * 3 # xyz
22 positionsRecvCounts = recvCounts * 3
23 positionsSendDisplacements = sendDisplacements *
        3
24 positionsRecvDisplacements = recvDisplacements *
        3
25
26 sendCounts = np.array(sendCounts, dtype='intc')
27 recvCounts = np.array(recvCounts, dtype='intc')
28 sendDisplacements = np.array(sendDisplacements,
      dtype='intc')
29 recvDisplacements = np.array(recvDisplacements,
      dtype='intc')
30
31 positionsSendCounts = np.array(
      positionsSendCounts, dtype='intc')
32 positionsRecvCounts = np.array(
      positionsRecvCounts, dtype='intc')
```

```python
33 positionsSendDisplacements = np.array(positionsSendDisplacements, dtype='intc')
34 positionsRecvDisplacements = np.array(positionsRecvDisplacements, dtype='intc')
35
36 # To eliminate no data transfer errors
37 if sendCounts[rank] > 0:
38     sendBufBodies = np.array(updatedBodies, dtype='intc')
39     sendBufPositions = updatedPositionsFlat.astype('float64')
40     sendBufWorkers = np.array(updatedWorkers, dtype='intc')
41 else:
42     sendBufBodies = np.empty(0, dtype='intc')
43     sendBufPositions = np.empty(0, dtype='float64')
44     sendBufWorkers = np.empty(0, dtype='intc')
45
46 allUpdatedBodies = np.empty(sum(recvCounts), dtype='intc')
47 allUpdatedPositionsFlat = np.empty(sum(positionsRecvCounts), dtype='float64')
48 allUpdatedWorkers = np.empty(sum(recvCounts), dtype='intc')
49
50 # Assert just in case
51 assert allUpdatedBodies.size == sum(recvCounts), f"Rank {rank}: Mismatch in allUpdatedBodies size"
52 assert allUpdatedPositionsFlat.size == sum(positionsRecvCounts), f"Rank {rank}: Mismatch in allUpdatedPositionsFlat size"
53
54 comm.Allgatherv(
55     [sendBufBodies, sendCounts[rank]],
56     [allUpdatedBodies, recvCounts, recvDisplacements, MPI.INT]
57 )
58
59 comm.Allgatherv(
60     [sendBufPositions, positionsSendCounts],
61     [allUpdatedPositionsFlat, positionsRecvCounts, positionsRecvDisplacements, MPI.DOUBLE]
62 )
63
64 comm.Allgatherv(
65     [sendBufWorkers, sendCounts[rank]],
66     [allUpdatedWorkers, recvCounts, recvDisplacements, MPI.INT]
67 )
68
69 allUpdatedPositions = allUpdatedPositionsFlat.reshape((-1, 3))
70
71 # Sync global values
72 positions[allUpdatedBodies] = allUpdatedPositions
73 workers[allUpdatedBodies] = allUpdatedWorkers
```

A.3 OctTree_v2 Graphs

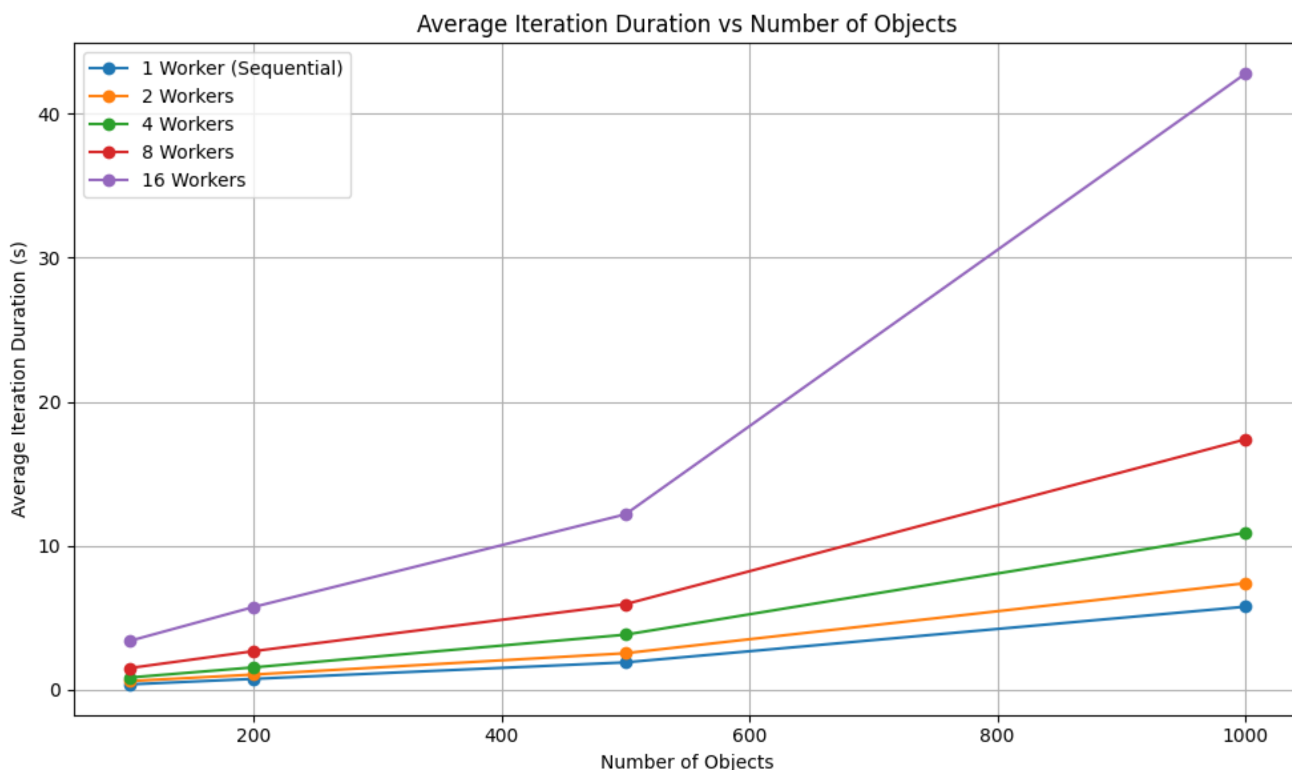

Figure 12: As number of objects grow, the iteration time grows as expected. However, as number of workers grow, the iteration time increases as well, indicating bad workload sharing or communication.

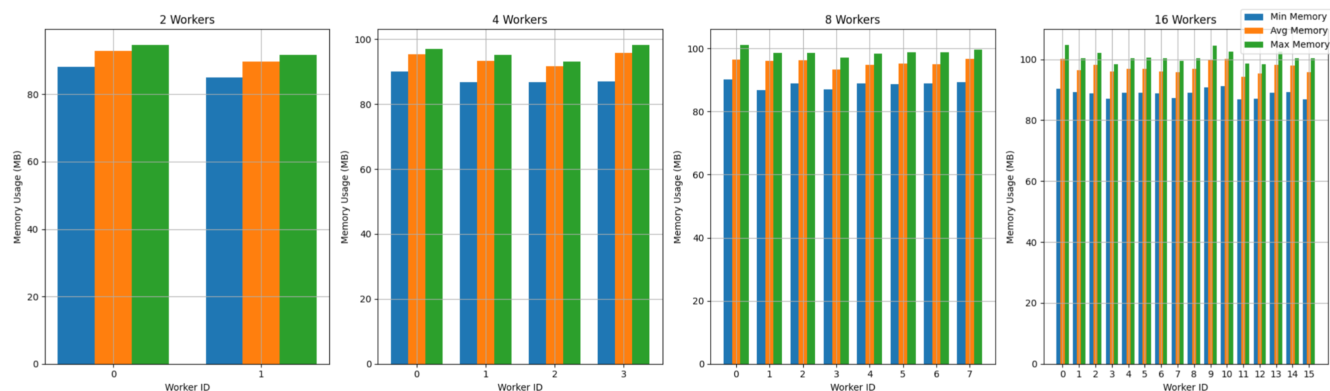

Figure 13: Memory consumption per worker with increasing number of workers (constant number of objects). There is a slight increase. Overall memory consumption is average memory consumption per worker times number of workers.

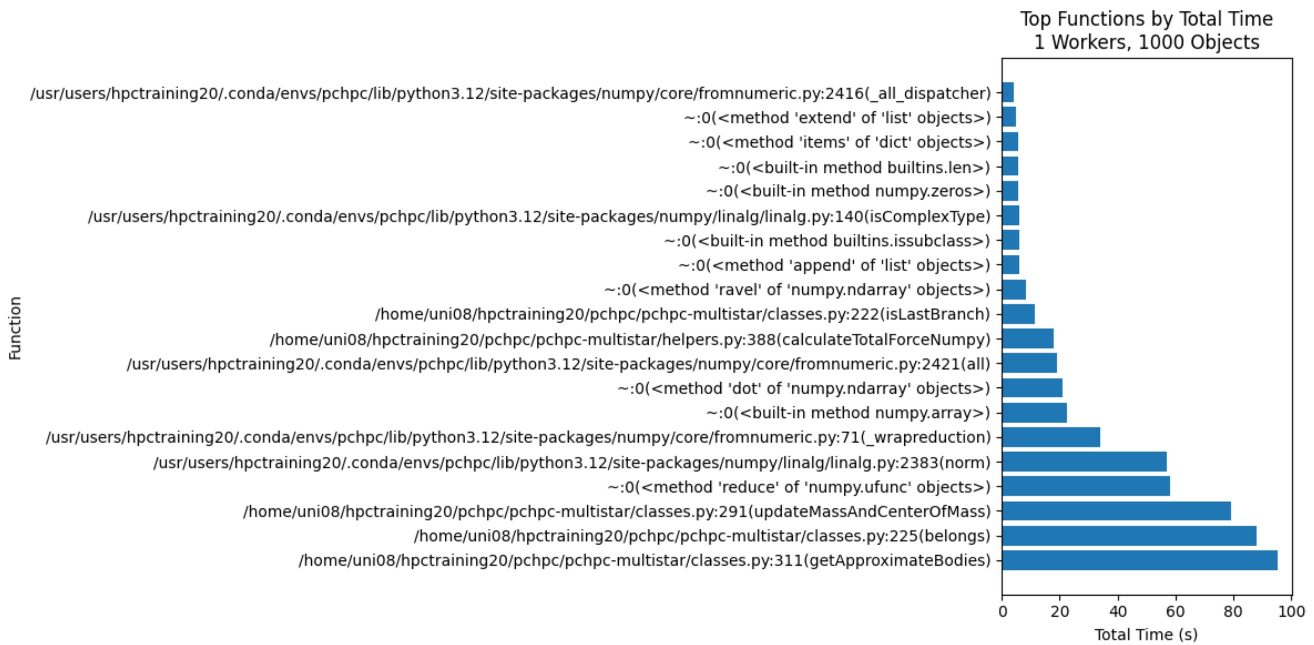

Figure 14: cProfile of simulation with 1000 objects in sequential mode. **Before the optimization**

.

Figure 15: cProfile of simulation with 1000 objects in parallel with 2 workers. The values are averaged by number of workers. **Before the optimization**.

Figure 16: cProfile of simulation with 1000 objects in parallel with 2 workers. The values are averaged by number of workers. **After the optimization**.

Figure 17: cProfile of simulation with 1000 objects in parallel with 4 workers. The values are averaged by number of workers. **After the optimization**.

A.4 Links

GitLab: https://gitlab.gwdg.de/ughur.mammadzada/pchpc-multistar

References

[1] Wikipedia contributors, "Barnes–hut simulation — Wikipedia, the free encyclopedia." https://en.wikipedia.org/w/index.php?title=Barnes%E2%80%93Hut_simulation&oldid=1247823185, 2024.

[2] Gesellschaft für wissenschaftliche Datenverarbeitung mbH Göttingen, "Scc - scientific computing cluster." https://gwdg.de/hpc/systems/scc/.

[3] Wikipedia contributors, "Leapfrog integration — Wikipedia, the free encyclopedia." https://en.wikipedia.org/w/index.php?title=Leapfrog_integration&oldid=1244408783, 2024.

[4] OpenAI, "Chatgpt: A large language model trained by openai." https://chat.openai.com/, 2024.