UNIVERSIDAD
DE LA REPUBLICA
URUGUAY

Parallel multithreading algorithms for self-gravity computation in ESyS-Particle

Nestor Pablo Rocchetti Martínez

Programa de Posgrado en Computación
Facultad de Ingeniería
Universidad de la República

Montevideo – Uruguay
Diciembre de 2020

UNIVERSIDAD
DE LA REPUBLICA
URUGUAY

Parallel multithreading algorithms for self-gravity computation in ESyS-Particle

Nestor Pablo Rocchetti Martínez

Tesis de Maestría presentada al Programa de Posgrado en Computación, Facultad de Ingeniería de la Universidad de la República, como parte de los requisitos necesarios para la obtención del título de Magíster en Computación.

Director:
D.Sc. Prof. Sergio Nesmachnow

Director académico:
D.Sc. Prof. Gonzalo Tancredi

Montevideo – Uruguay
Diciembre de 2020

Rocchetti Martínez, Nestor Pablo

Parallel multithreading algorithms for self-gravity computation in ESyS-Particle / Nestor Pablo Rocchetti Martínez. - Montevideo: Universidad de la República, Facultad de Ingeniería, 2020.

XV, 91 p. 29, 7cm.

Director:

Sergio Nesmachnow

Director académico:

Gonzalo Tancredi

Tesis de Maestría – Universidad de la República, Programa en Computación, 2020.

Referencias bibliográficas: p. 89 – 91.

I. Nesmachnow, Sergio, . II. Universidad de la República, Programa de Posgrado en Computación. III. Título.

INTEGRANTES DEL TRIBUNAL DE DEFENSA DE TESIS

Ph.D. Prof. Rafael Mayo

D.Sc. Prof. Tabaré Gallardo

D.Sc. Prof. Pedro Piñeyro

Montevideo – Uruguay
Diciembre de 2020

Acknowledgements

I would like to thank Dr. Dion Weatherly and Dr. Steffen Abe for their support in implementing the changes in ESyS-Particle. Dr. Derek Richardson and Dr. Steven Schwartz helped us in the comparison with other DEM packages, and in particular Derek kindly welcomed Dr. Tancredi and me in our visit to the University of Maryland. Also, I would like to thank Daniel Frascarelli for helping me understand the self-gravity module of ESyS-Particle. Finally, I would like to thank ANII for the financial support to carry out the thesis.

ABSTRACT

This thesis describes the design, implementation, and evaluation of efficient algorithms for self-gravity simulations in astronomical agglomerates. Due to the intrinsic complexity of modeling interactions between particles, agglomerates are studied using computational simulations. Self-gravity affects every particle in agglomerates, which can be composed of millions of particles. So, to perform a realistic simulation is computationally expensive. This thesis presents three parallel multithreading algorithms for self-gravity calculation, including a method that updates the occupied cells on an underlying grid and a variation of the Barnes & Hut method that partitions and arranges the simulation space in both an octal and a binary tree to speed up long range forces calculation. The goal of the algorithms is to make efficient use of the underlying grid that maps the simulated environment. The three methods were evaluated and compared over two scenarios: two agglomerates orbiting each other and a collapsing cube. The experimental evaluation comprises the performance analysis of the two scenarios using the two methods, including a comparison of the results obtained and the analysis of the numerical accuracy by the study of the conservation of the center of mass and angular momentum. Both scenarios were evaluated scaling the number of computational resources to simulate instances with different number of particles. Results show that the proposed octal tree Barnes & Hut method allows improving the performance of the self-gravity calculation up to 100× with respect to the occupied cell method. This way, efficient simulations are performed for the largest problem instance including 2,097,152 particles. The proposed algorithms are efficient and accurate methods for self-gravity simulations in astronomical agglomerates.

Keywords:
Simulation, high-performance computing, self-gravity, astronomical agglomerates.

Contents

Chapter 1

Introduction

Some astronomical objects, like asteroids and comets, are agglomerates of smaller particles called grains, which are kept together by their mutual gravitational force. Grains are affected by short range interactions (e.g., contact forces) and long range interactions. Long range interactions are a combination of the effect of the influence of the gravity of other objects and the effect of the influence of the other grains that conform the agglomerate itself. The latter is called *self-gravity* of the agglomerate (Harris *et al.*, 2009; Fujiwara *et al.*, 2006). Gravitational potential can cause attraction between astronomical objects and also deformations. This way, self-gravity gives shape to asteroids and comets composed of agglomerates of particles (Rozitis *et al.*, 2014).

Due to the intrinsic complexity of modeling interactions between particles, agglomerates are studied using computational simulations. A straightforward approach to compute the long range interactions between every pair of particles in an agglomerate with N particles has a computational cost of $O(N^2)$ in each step of the simulation. Thus, performing simulations of millions of particles, as usual to model medium-size astronomical objects, is computationally expensive.

The High Performance Computing (HPC) paradigm helps researchers to solve complex problems and perform simulations on big domains. HPC allows dealing with complex problems that demand high computer power in reasonable execution times. Instead of using a single computing resource, HPC proposes using multiple resources in parallel, applying a coordinated approach. This way, a cooperation strategy is implemented, allowing the workload to be divided between the computational units available to solve a complex problem in reasonable execution times.

ESyS-Particle (Abe *et al.*, 2009) is a software library for simulation of geological phenomena using the Discrete Element Method (DEM). ESyS-Particle includes features for execution in parallel and distributed environments.

The first applications of ESyS-Particle in planetary sciences were presented by our research group (Tancredi *et al.*, 2012), including simulations in low-gravity environments (asteroids and comets) and new models to simulate contact forces. A specific shortcoming of ESyS-Particle (and other DEM software) is the lack of models to simulate long-range forces. Our previous work (Frascarelli *et al.*, 2014) proposed a self-gravity module applying HPC techniques to allow performing simulations of thousands of particles efficiently by exploiting multiple computing resources. Strategies to efficiently compute long-range forces were introduced, implemented, and evaluated over realistic scenarios.

In this line of work, this thesis presents parallel multithreading algorithms for self-gravity calculation, including a method that updates the occupied cells on an underlying grid and a variation of the Barnes & Hut method that partitions and arranges the simulation space in both an octal and a binary tree to speed up long range forces calculation. Both methods and its variants are evaluated and compared over two scenarios: two agglomerates orbiting each other and a collapsing cube. The experimental evaluation comprises the performance analysis of the two scenarios using the two methods, including a comparison of the results obtained and the analysis of the numerical accuracy. Both scenarios were evaluated scaling the number of computational resources to simulate instances with different number of particles. Results show that the proposed octal tree Barnes & Hut method (Barnes and Hut, 1986) allows improving the performance of the self-gravity calculation up to 100× with respect to the occupied cell method. This way, efficient simulations are performed for the largest problem instance including 2,097,152 particles.

Three conference papers and a journal article were written, which include the partial results obtained during the development of this thesis. In chronological order, the first work is the conference paper: "Performance improvements of a parallel multithreading self-gravity algorithm" (Rocchetti *et al.*, 2017). After that is the conference paper: "Comparison of Tree Based Strategies for Parallel Simulation of Self-gravity in Agglomerates" (Rocchetti *et al.*, 2018). Then is the conference paper "Large-Scale Multithreading Self-Gravity Simulations for Astronomical Agglomerates" (Nesmachnow *et al.*, 2019). The last is the journal article "High performance computing simulations of self-gravity in astronomical agglomerates" that is under revision at the moment of writing this thesis.

The main contributions of this thesis are: i) the presentation and the experimental evaluation of an upper tree level mass center calculation implemented as an extension of the mass center calculation algorithm included in the Barnes & Hut octal tree construction algorithm, ii) the experimental analysis of the Barnes & Hut octal tree algorithm with a collapsing cube scenario, and iii) the study of the conservation of the center of mass and the angular momentum for the scenarios and the algorithms presented in this article.

The thesis is organized as follows. Next chapter introduces the self-gravity calculation problem and reviews related works on domain decomposition for particle simulators. After that, a parallel self-gravity calculation algorithm is presented which is the base of our work. Then, the adapted Barnes & Hut method for self-gravity calculation is described. Next, the test scenarios and instances used to perform the evaluation are described. Then, the experimental evaluation of the octal tree is presented, followed in the next chapter by the experimental evaluation of the binary tree algorithm and the upper tree level mass center calculation. Finally, the conclusions and main lines for future work are formulated.

Chapter 2

Self-gravity calculation and related work

This chapter starts by the introduction of the problem of self-gravity calculation. Then, follows the explaination of the Discrete Element Method. After that, the chapter continues with a review of previuos works about static, dynamic and combined spatial domain decomposition techniques, used to speed up the calculations of interaction between particles. Finally the chapter presents a parallel algorithm for self-gravity calculation and an implementation of a self-gravity algorithm on a particle interaction simulator called ESyS-Particle.

2.1 Calculating the self-gravity

The self-gravity calculation problem consists of calculating the self-gravity of assemblies of N particles. In this problem, every particle present in the assembly interacts (and therefore is affected by) all the other particles. The mathematical model for computing the gravitational potential resulting of the gravitational interaction with the rest of the particles in the studied system is expressed in Eq. 2.1, where G is the gravitational constant, $\|\overrightarrow{r}\|$ is the norm of the distance vector $\overrightarrow{r}$, M_i is the mass of the particle, and V_j is the gravitational potential of particle.

$$V_j = \sum_{i \neq j} \frac{GM_i}{\| \overrightarrow{r_j} - \overrightarrow{r_i} \|} \tag{2.1}$$

A straightforward implementation of the gravitational potential calculation according to Eq. 2.1 results in a computational cost of $O(N^2)$ when calculating the potential for each particle in each time step. This approach turns to be inefficient when the simulation scenario scales to hundreds of thousands of particles.

Many techniques have been developed in order to overcome the computational inefficiency problem when considering simulations with many particles. The objective of the methods proposed is to develop strategies to efficiently calculate the long range forces based on algorithms that form groups of particles.

2.2 The Discrete Element Method

The Discrete Element Method was first denominated Distinct Element Method and was presented by Cundall and Strack (1979). In this work both methods are adressed with the initials DEM.

DEM consists of calculating the interaction between individual particles, and after that, achieving the representation of the behavior of assemblies by the propagation of the interactions. The simulations are performed in two dimensional spaces and the agglomerates of particles are dry. So, the particles represented are (solid) discs, which can have any size or density. A characteristic of DEM is that contact forces are calculated over time steps of constant length. The velocities and accelerations between time step intervals are considered invariant. According to Cundall and Strack (1979), the time step length has to be chosen small enough that during a single time step the disturbances cannot propagate from any disc further than to the discs that are in contact with it. This way, the net force over a disc is determined only by the interaction with other discs in contact with it. This property of DEM makes it possible to perform simulations without consuming excessive memory.

The net forces for each particle are calculated for each of the time steps of a simulation. The cycle of calculation consists of computing the displacement forces (normal and tangential forces) using a force-displacement law, and then the total force is computed using the second law of Newton. Finally, the displacement vector of a discs is calculated from the total force.

According to Cundall and Strack (1979) the deformation of the assemblies is a consequence of the movement of the particles rather than its deformations.

So, the authors stated that a precise model of the deformation of the particles is not needed. This way, the deformation of two particles is modeled as the overlapping of one another at contact points. A larger overlap implies a larger force that the particles exert to each other due to the deformation. The overlap is small compared to the sizes of the particles. Also, the overlapping is directly related to the time step and the displacement speed of the particles.

The second law of Newton is used for each particle after the forces that are exerted over each of the particles are calculated. The second law is used to calculate the acceleration of a particle for the next time step. The acceleration is calculated from the vectorial sum of the forces of the particles that are in direct contact with it.

The experimental evaluation of the DEM implemented by Cundall and Strack (1979) was performed as a two dimensional execution. The test scenario was composed of 197 discs of different sizes surrounded by four walls that form a square. The experiment consisted of compressing the discs by moving the walls causing the discs to collide and overlap. The purpose of the experiment was to evaluate the resultant forces and the acceleration. The results obtained were compared against a physical experimental setup which was the same as the scenario used for the computational simulation. Cundall and Strack (1979) concluded that the experimental results are subjecive. Nonetheless, the results obtained were sufficient to conclude that DEM is a valid method to perform granular assemblies simulations.

DEM is used to accurately and efficiently perform the calculations of the contact forces over the particles for each time step of a simulation. When performing simulation of astrophysical models, two types of forces are taken into account: short range forces and long range forces. The short range forces are the forces that result from the contact between particles, which are calculated using DEM. The long range forces are the self-gravitational forces and cannot be calculated using DEM as presented by Cundall and Strack (1979). The following section presents and analyzes the problem of calculating the self-gravity of assemblies of particles.

2.3 Related work on domain decomposition techniques

This section reviews the different domain decomposition techniques developed to speed up the self-gravity calculation when performing astrophysical particle simulations. The techniques presented are classified as static or dynamic domain decomposition techniques.

Static hierarchical domain decomposition

Static techniques for particle interaction are based on a domain decomposition that stays invariable during a simulation. According to Hockney and Eastwood (1988), static techniques are divided in three models: Particle-Particle (PP) methods, Particle-Mesh (PM) methods, and Particle-Particle Particle-Mesh (P3M) methods. PP are the simplest methods and consist of computing the forces directly between all the particles in the system. Despite being the most accurate, the PP model lacks the ability to scale in the number of particles due to its $O(N^2)$ execution time. PM methods consist of using a mesh over the simulated space and calculating the potential for each particle that belongs to the mesh. After that, the speed for the particles is calculated via an interpolation. According to Hockney and Eastwood (1988), although PM methods are less accurate than PP methods when computing the forces, PM methods are in general significantly faster than PP methods. However, in many practical applications, PM may need a mesh resolution that should result to be slower than a PP method; thus the PM model is not usually used to calculate contact forces. Finally, P3M methods combine the PP and the PM models. The P3M model divides the forces applied to a particle into short range and long range forces. The short range forces are calculated using a PP method and the long range forces are calculated using a PM method. The combination results in a fast and accurate method to simulate particle interaction. Many implementations and variations of the aforementioned three models are present in the literature. Some of the more relevant implementations are reviewed next.

Couchman (1991) proposed a variation of the P3M algorithm, that applies a selective refinement of the grid depending on the particle density of a cell. The refinement is performed recursively while a density threshold is exceeded.

Given that the refinement is performed recursively, the particle mesh is an spatial division without overlapping zones to be calculated. This way, assigning a similar number of particles to each processing unit, a load balancing scheme is implemented. The results obtained after the refinements are then passed over to the father cell to be integrated via direct summation. There is a correlation between the level of refinement of the grid and the time spent to integrate the results calculated for the individual cells. The refinement level has to be calibrated for this method to be efficient. The method presented by Couchman has the constraint that the domain of the simulation must be cubic and also the space must be divided into an integer number of cells of the same size. Couchman stated that, in this method "The gain is in simplicity" (p. 24). Results presented by Couchman indicate that the algorithm is up to 20 times faster than a P3M algorithm and also requires less memory.

Kravtsov *et al.* (1997) presented an N-body solver called Adaptive Refinement Tree based on the particle-mesh method over a multilevel grid to perform the calculations of the forces on the system. The mesh is created over a cubic space that is divided by regular cells with cubic shape and a predefined size. A multilevel mesh is defined by dividing large cells into eight parts that have the same size depending on the particle density. The multilevel mesh is created at the beginning of the simulation and then it is partially updated when the forces need to be recalculated. The smallest size of an element of the grid is the resolution of the mesh (i.e., a cell). The implementation presented by Kravstov et al. was partially parallel. The section of the application that was parallelized is the one in which the forces are updated for the particles (i.e., the force interpolation). Kravstov et al. stated that the solution is half as fast as the Fourier transform solver with the same number of cells. The method proposed by Kravstov et al. is similar to the method presented in this thesis, in the sense that it spawn an octal tree in which the maximum resolution is a cell in the mesh. On the other hand, in the method that Kravstov et al. proposed, the finer meshes are built when a predefined size threshold is exceeded, but in the method introduced in this thesis, a finer mesh is created when there are particles present in the cell.

An implementation of a DEM simulator using the PM method was presented by Sánchez and Scheeres (2012). The same static domain decomposition technique is used to calculate short range interactions as well as long range interactions. The space of the simulation scenario is divided in cubical cells that are many times bigger than the particles radius of the scenario. Then instead of considering individual particles to calculate the gravitational potential, the whole cell is considered as a particle. The calculations for N particles are still $O(N^2)$, but the authors claimed that the amount of calculations required decreased in one order of magnitude. The short range and the long range interactions are computed concurrently. However, the authors did not propose a parallel implementation. Thus, the method was able to compute efficiently simulations of systems with up to $8,000$ particles. The results presented by the authors show that the algorithm implemented is not suitable to perform large scale simulations that comprise hundreds of thousands of particles. In order to do so, a parallel or distributed implementation of the algorithm is needed.

Dynamic hierarchical domain decomposition

Dynamic techniques for particle interaction calculation use structures that are adapted or reconstructed from scratch during the simulation. A classic technique in this field is the method proposed by Barnes and Hut (1986). In this method, the simulated domain is divided in a hierarchical octal tree to accelerate the calculation of the gravitational potential in a N-body simulation. The tree is composed of nodes that represent a portion of the space of the simulation. The root node of the tree represents the complete space of the simulation. If a node (i.e., the space it represents) contains more than one particle, the space is divided into smaller pieces. In a two-dimension simulation, the space is divided in four smaller pieces. However, in a three-dimension simulation, the space is divided into eight pieces. Applying this domain decomposition Barnes and Hut affirmed that the long-range interactions are calculated in $O(N \log N)$, being N the number of particles in the domain. According to the authors, experimental results indicate that the error increases as the simulation runs. As one of the improvements for the algorithm presented, Barnes and Hut suggest implementing individual time steps for each particle, with the idea of updating more frequently those particles that move faster.

Greengard and Rokhlin (1987) presented the Fast Multipole Method. The method consists of performing a multipole expansion of the space of the simulation to be performed. The expansion is organized as a hierarchy of meshes rather than a tree. However, the elements of meshes with higher resolution that are an expansion of an element of a lower resolution mesh are considered its sons. According to the Greengard and Rohklin, clusters of particles can interact with one another as long as they are "well separated". The range of separation is defined as a configuration parameter before starting a simulation. The expansion is pure spatial. The spatial center of a father mesh box is one expansion center of the next level. The expansion is performed whether there are particles in the box or not. After creating the meshes, the potential is calculated for each center of all the meshes. Then, the potential for each particle is calculated as an extrapolation of the nearest neighbor boxes at the finest mesh level. Greengard and Rohklin implemented the algorithm and performed a comparison with a direct computation in single precision. The experimental evaluation was performed on a VAX-8600. The results on the comparison presented by Greengard and Rokhlin indicate that in an instance consisting of 12,800 particles the execution time of the method presented is 300$\times$ lower than the direct method. In addition, Greengard and Rohklin stated that the execution time grows linearly with the number of particles.

Combined techniques

Xu (1994) presented a variation of the P3M method called Tree Particle Mesh (TPM). The difference with P3M lies in that short-range interactions are calculated using the algorithm presented by Barnes and Hut. However, long-range interactions are calculated using a PM method. The TPM method consists of using the tree code only on the parts of the field that have a density over a certain threshold. Each of the trees created has a different time step. The regions that have low particle density are managed only by the PM method. Using the tree code on high density zones allows the PM method to have a lower resolution (i.e., less cells are created), hence the simulations run faster. The TPM method proposed by Xu was parallelized using a Multiple Instruction Multiple Data model. To provide load balance, the tree code was parallelized in two levels. In the first level, the trees are assigned to the available processors. The second level of the parallelism balances the load when the number of processors is higher than the number of trees. In the second level of parallelism, many processors work on a single tree. After a tree construction is finished, the structure of the tree is broadcasted to a group of processors that are going to perform the tree walk and force summation. Xu explained that the construction of the trees was not parallelized due to the fact that only 2.5% of the time is spent on that activity. The algorithm showed a speedup of 26 when running on 32 processors. The author also performed a comparison with the PM and the tree method. Results indicated that the pure tree method is 12× slower than the TPM code presented by Xu. Also, the PM method turned to be 1.2× faster when the PM box size (i.e., the resolution) is set to 0.010 and the TPM box size is set to 0.003.

Bode *et al.* (2000) presented an algorithm as an improvement of the TPM originally presented by Xu (1994). In the algorithm presented by Bode *et al.* (2000), the forces are calculated as a combination of the PM and a tree algorithm. This algorithm is used to calculate and evolve the short range interactions. On the one hand, the forces internal to the tree are calculated by a tree algorithm. On the other hand, the long range interactions are managed by a PM algorithm. At the beginning of a simulation the time step is the same for PM and the tree algorithm. The tree algorithm can use a smaller time step if needed. According to Bode et al., the algorithm divides the space into regions based on particle density, called clusters. Before creating the clusters, cells of the grid that have a density that is above a defined threshold are assigned a key that identifies them. Then, for each of the cells with a density above the threshold, the surrounding cells that also have a density above the threshold are given the same key and becoming a cluster. The density threshold is defined as a function of the mean density of the cells and its standard deviation. Whereas the focus of the implementation are simulations of clusters of particles, the algorithm can be used to simulate other scenarios. A comparison with the P3M algorithm was performed. Experimental results showed that the TPM algorithm speeds up the simulations "by a factor of 3-4" (Bode et al. 1994 p.566) when the trees have individual time steps. The experimental evaluation was performed on a cluster of computers composed of SGI Origin 2000. Results of a study of computational efficiency showed that the algorithm scales from 4 to 16 processors, then efficiency drops to 70% at 32 nodes.

Bagla (2002) presented a method for performing simulations of large particle agglomerates simulations, called TreePM. The method combines the Barnes and Hut tree code and the PM method. The method presented is similar to the TPM method proposed by Bode *et al.* (2000). The forces are divided into long range and short range forces. The long range forces are calculated using the PM method, whereas the short range forces are calculated using the Barnes and Hut algorithm. For the short range forces, the criteria is to calculate the contribution from particles or from grid cells. A cell is considered when a distance threshold is exceeded, instead if the threshold is not exceeded, the subcells or particles should be considered. According to Bagla, there are two differences between the TreePM method and the TPM method. One difference is that, while TPM uses the PM method to compute the long range forces, TreePM uses an explicit and experimentally defined distance to decide which method is used to compute the forces in a particular region. The other difference is that in TPM the forces are computed for each individual particle only for the particles that belong to high density regions, but in TreePM the forces are computed individually for all the particles. According to Bagla, experimental results indicated that TreePM is about 4.5x faster than a tree algorithm. Also, results show that the simulations scale in O(N log N), being N the number of particles.

Khandai and Bagla (2009) presented a modification of the TreePM algorithm. Khandai and Bagla adapted the suggestion by Barnes and Hut (1986) in the context of the TreePM algorithm previously presented by Bagla (2002). The tree part of the algorithm is used to calculate the short range forces. In the optimization suggested by Barnes, the force is computed in only one tree walk. In addition, the particles are divided into groups, but instead of grouping them by their densities as suggested by Bode *et al.* (2000), the groups are created according to the particles number and volume. The authors performed experiments with scenarios with between 32^3 to 256^3 particles. Results indicated that a TreePM algorithm with group scheme and individual time steps for each group had a speed up of 12.72 compared to an unoptimized TreePM algorithm.

Ishiyama *et al.* (2012) presented a variation of the TreePM method applied to a N-body simulation of one trillion particles which was executed using the full capacity (663552-cores) of the K Computer of the RIKEN institute (`https://www.top500.org/system/177232`). The results obtained were presented as an entry for the Gordon-Bell performance prize. For the short-range forces, the Barnes & Hut algorithm was modified to create trees for groups of particles. Also, Ishiyama et al. performed modifications to implement a list of tree nodes and particles that is shared by a group of particles. By using this method, Ishiyama et al. stated that the algorithm "can reduce the computational cost of tree traversal by a factor of N_i" (p.3) being N_i the average number of particles of the groups. To measure the space of the simulation, Ishiyama et al. used the sampling method to perform a 3D multi-section decomposition. In addition, Ishiyama et al. implemented load balancing by adjusting the size of the cells according to the cost of calculating the potential for the cells. As a complement, the interactions were calculated using the Single Instruction Multiple Data paradigm, combined with loop unrolling. Regarding the PM method, Ishiyama et al. implemented a distributed Fast Fourier Transform (Cochran *et al.* (1967)). Reported results indicated that the algorithm proposed has good scalability in terms of the number of nodes used, which means that there is not a significative overhead in the communications. Using 82,944 computing nodes, Ishiyama et al. stated that the algorithm achieved a performance of 4.45 Pflops and an efficiency of 43%. The algorithm presented is 1.44x faster compared to the Gordon-Bell winner of the previous year.

Chapter 3

Improvements of a parallel self-gravity algorithm

This chapter describes performance improvements of the self-gravity calculation algorithm previously implemented in ESyS-Particle. The improvements establish a new baseline implementation for the self-gravity algorithm to be compared with. Section 3.1 describes a parallel algorithm to calculate self-gravity. Then, section 3.2 explains the implementation of the algorithm in ESyS-Particle including its characteristics. Section 3.3 explains the improvements to reduce the self-gravity calculation time of the self-gravity module of ESyS-Particle. Section 3.4 reports the profiling results and the analysis of the self-gravity calculation before implementing the occupied cells method. Finally, section 3.5 explains the occupied cells method.

3.1 A parallel algorithm for self-gravity calculation

Frascarelli *et al.* (2014) presented an algorithm based on DEM to perform self-gravity calculations and contact forces calculations for the simulation of small solar system bodies. Frascarelli et al. presented details on the implementation, the test scenarios, and the experimental results.

The algorithm developed by Frascarelli *et al.* (2014) makes use of parallel programming techniques based on multithreading in order to accelerate the calculation of long-range forces and also short-range contact forces. In particular, for the long-range forces computation, a domain decomposition technique was implemented following the master-worker model for parallelization. The advantage of using threads lies in the efficiency of the data communication and synchronization via shared memory. A pool of threads was used to avoid the intensive creation and destruction of threads. The algorithm for self-gravity recalculation consists of five stages: initialization, threads creation, self-gravity calculation, interpolation, and output. The initialization phase consists of initializing the shared memory and loading the information of the particles. Then, in the threads creation phase, the self-gravity calculation is divided into smaller tasks to be assigned to the pool of threads. Afterwards, in the self-gravity calculation phase, the tasks are assigned in turn to an idle thread of the pool created in the previous phase. Then, the pool of spawned threads is used to perform an interpolation for each of the particles in the system with respect to the eight surrounding nodes in order to calculate the self-gravity of each particle.

To increase the performance of the algorithm, the acceleration is calculated for a virtual point located at the center of each cell that composes the grid. Thus, a cell of the grid is the minimum processing unit. A hierarchical grouping approximation method (*Mass Approximation Distance Algorithm*, MADA) was introduced. The main goal of MADA is to accelerate the calculation of the gravitational potential of a particle in a given time step, by considering a group of distant particles as a single particle located in the center of mass of the group. The proposed parallel algorithm calculates the self-gravity using MADA and a pool of worker threads that execute the most computing-intensive tasks in parallel sections, following a P3M approach. The principle of MADA is that, when calculating the self-gravity for a given cell, a more refined grid is used for the cells that are near the cell to update. This way, the self-gravity is calculated more accurately. The particles are used individually for the cells that are next to the cell to update.

The experimental evaluation was performed to determine the efficiency and accuracy of the proposed strategies. The infrastructure used to perform the evaluation was a 24-core, 2.1-GHz AMD Opteron 6172 processor with 24 Gbytes of RAM available at Cluster FING (www.fing.edu.uy/cluster). To test the numerical accuracy, an scenario containing an espherical agglomerate of 1,022,208 particles with a radius of 20m was used. The numerical accuracy was determined by calculating the distance of the experimental center of mass versus the theoretical center of mass. Results presented by Frascarelli et al.

showed an error less than 0.1%. The results of the computational efficiency study showed a near linear speed up when using approximately 100,000 particles. Nevertheless, the computational efficiency decreases as the number of particles increases.

Nesmachnow *et al.* (2015) presented several computation and data-assignment patterns to determine the best efficiency and scalability properties of the resulting self-gravity computation method. Four strategies were proposed to dynamically balance the workload assigned to the threads when calculating the self-gravity. The first strategy is *interlocking linear strategy*, in which each worker thread linearly takes a cell to process. The first thread takes the first cell to process, the second thread takes the second, and so on until all the threads have been assigned a cell to process. Then, the first thread that finishes working takes the next available cell to process. The second strategy is *circular concentric strategy*. In this strategy the threads are divided into two groups. One group starts processing the cells from the center of the grid and the other from the border with the main goal of reusing the already calculated centers of mass. The third strategy is *basic isolated linear strategy*, which consists of assigning clusters of cells of equal size to each thread of the pool. The fourth strategy presented is the *advanced isolated linear strategy*. This strategy consists of dividing the workload evenly between the threads spawned. Then, after a thread finishes processing its workload, it starts processing the nearest unprocessed cell. The process ends when there are no cells left to process. Experimental results demonstrated that the best implementation of the algorithm among the presented was the advanced isolated linear strategy. This strategy was able to scale up linearly with the number of particles in the system, and it scaled with an inverse power law (exponent 0.87) with the number of threads used in the computation. The observed speed up was close to linear for systems containing up to 2×10^5 particles.

The self-gravity algorithm presented in the works of this section was not included in a tool that integrated the self-gravity calculations. In order to perform realistic simulations of particle interactions in low gravity environments the work presented in the following section shows how the self-gravity algorithm was included in a particle interaction simulator called ESyS-Particle.

3.2 Implementation of the Self-gravity algorithm on ESyS-Particle

This section describes the main features of a self-gravity algorithm implemented in ESyS-Particle. The work presented in this section is the base for the work presented in this thesis.

Abe *et al.* (2009) presented ESyS-Particle, a particle interaction simulator that implements the DEM. ESyS-Particle was developed to simulate geophysical phenomena, so the particle interacions that were included only comprised the contact forces. ESyS-Particle is fully parallelized and can run in parallel or distributed environments such as multi-core computers or clusters. It is implemented in C++ and is parallelized using Message Passing Interface (MPI) (Walker and Dongarra (1996)).

Weatherley *et al.* (2010) presented a benchmarking of ESyS-Particle to analyze the scalability and the accuracy of the calculations of contact forces algorithm. According to Weatherley et al., in order to replicate some geophysical processes using thousands of particles is not enough to perform realistic simulations. Those scenarios need millions of particles. Depending on the number of particles simulated, every time the total forces of the particles are updated, the self-gravity computation process can last from seconds to hours. The number of calculations to update the self-gravity for N particles and M nodes in the grid is of the order of $O(N \times M)$ for the self-gravity computation method in ESyS-Particle. To deal with that efficiency problem, HPC techniques were applied to speed up the calculations. ESyS-Particle implements spatial domain decomposition using a master-worker model implemented on MPI. The domain decomposition is static and it is defined as a configuration before the simulation starts. The configuration consists of numerical parameters , one for each axis, and is defined to indicate how many times each dimension must be divided. For example, the partition vector (1,2,2) means that, at the beginning of the simulation, the x dimension will not be divided, while dimensions y and z will be divided into two sections. This way, the space is divided into four subdomains. Each of the subdomains is assigned to a process, therefore dividing the total calculations by a factor of four, ideally. An experimental evaluation of the scalability of ESyS-Particle was performed by Weaterley et al. The scenario defined was a cube packed with particles with radii of between 0.2 mm and 1.0 mm. The initial length of the side of the cube was 15 mm. Then, the

length of the side of the cube was increased for larger number of subdomains and particles. Each subdomain had approximately 9,000 particles. The smallest test had one subdomain and 8,647 particles, while the largest subdomain had 1,000 workers and 8,700,121 particles. For every simulation performed, one processor was assigned for each subdomain. Also, all the simulations were executed for 10,000 time steps. Experimental results indicate that when the number of workers increased from 1 to 27, there was a significant performance degradation. According to Weatherlet et al., from 27 workers to 1,000 workers the total execution time was constant. The real execution time for the scenario with 27 subdomains (241,540 particles) is 1104.5 s, while for the the scenario with 1,000 subdomains (8,700,121 particles) was 1617.1 s.
The use of ESyS-Particle was to perform geophysical simualtions. So, only the simulation of short range interaction forces was implemented. The idea for including the self-gravity calculation module in ESyS-Particle was to extend the functionality of ESyS-Particle by enabling the simulation of long-range interactions. The computation scheme in the self-gravity module implemented into ESyS-Particle applies four steps:

1. Compute the gravity acceleration field in a grid of nodes enclosing the limits of the space of the simulation;
2. For every particle at each time step, compute the contact forces over the particle and interpolate the value of the acceleration for the location of the particle using the values of the acceleration on the surrounding nodes;
3. Apply the forces and advance the system;
4. If a large displacement of the particles is found, the gravity field is updated; if not, the previous gravity field is used for the next time step and step 2 is executed again.

The variation of the gravity forces applied to a particle is affected by the velocity of the particles in the system. During the simulation, self-gravity forces are updated after the particles in the system move a distance that is larger than a certain threshold. When particles move faster in a simulation, self-gravity needs to be updated more frequently in comparison to the same system if the particles moved slowly. However, the frequency of update of the contact forces is of the order of the duration of the contact. In low speed simulations, the number of updates of contact forces can be many orders of magnitude more than the number of updates of long range forces.

The implementation of the self-gravity algorithm presented was based on a master-worker model, but including a two-level parallelization scheme that is implemented using multithreading programming techniques. The master-worker model is the one used by ESyS-Particle to calculate and update the forces that affect the particles and it is implemented using a distributed memory approach. The master process is in charge of calling the self-gravity module, which calculates and updates the gravity forces that affect the particles. The calculation of self-gravity forces is implemented using shared memory and multithreading programming techinques. Thus, two different implementations (distributed memory and shared memory) are included.

Before starting a simulation, the self-gravity module builds and overlays a grid to divide the spatial domain of the simulation. The grid is composed of boxes, whose vertexes are called *nodes*. The number of nodes and their location are defined depending on the spatial domain and the size of the boxes. The acceleration along the x-axis (a_x) on a node located at position (x, y, z) due to an ensemble of N particles of individual mass m_j, distance r_j, and positions (x_j, y_j, z_j) is given by Eq. 3.1. Similar equations are formulated for the acceleration along the y-axis in Eq. 3.2 and the z-axis in Eq. 3.3.

$$a_x = \sum_{j=1,N} Gm_j \frac{x_j - x}{r_j^3} \tag{3.1}$$

$$a_y = \sum_{j=1,N} Gm_j \frac{y_j - y}{r_j^3} \tag{3.2}$$

$$a_z = \sum_{j=1,N} Gm_j \frac{z_j - z}{r_j^3} \tag{3.3}$$

Figure 3.1 shows a particle inside its box and its eight surrounding nodes in a step of a simulation. The nodes are numbered from one to eight. The acceleration is updated for the nodes rather than for the particles. To calculate the acceleration of a particle at a given time step of a simulation, an interpolation is applied using the values of the acceleration calculated for the eight nodes that surround the particle.

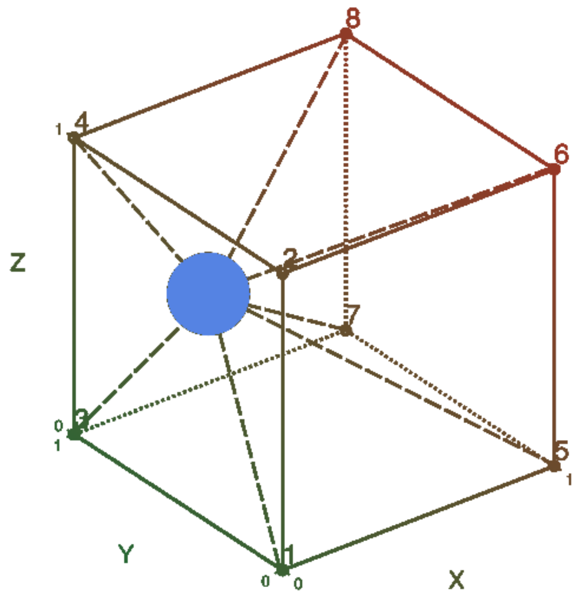


Figure 3.1: Box of the self-gravity grid with eight nodes numbered.

3.3 Reducing the execution time of the self-gravity computation

In the implementation presented by Frascarelli *et al.* (2014), the self-gravity potential on ESyS-Particle was updated on every node of the overlay grid used to calculate the self-gravity field. However, calculating the values of the acceleration of only the occupied nodes is enough to update the acceleration of all the particles in the system.

In order to accelerate the self-gravity computation, a new strategy was proposed and implemented as part of the work presented in this thesis. In the new strategy, only the occupied cells are updated. This strategy improved the utilization of the computational resources available by sparing the update of the acceleration of the unoccupied cells.

In the new strategy, self-gravity is updated when the difference of the position of at least one particle at a given time step, compared to the previous time step, is larger than a predefined distance threshold. In the context of a simulation, a particle that belongs to a box at the start of a simulation can migrate to different boxes during the simulation. However, that migration may not trigger the self-gravity update if the particle just moves a distance that is not larger than the predefined distance threshold. In this situation, and given that the acceleration is only updated on the occupied nodes, some of the eight surrounding nodes needed for the interpolation may be outdated. The use of old self-gravity values may introduce error in the calculations during a simulation. To prevent this situation, the self-gravity is updated for an expanded box that comprises the 64 nodes that surround a particle. Fig. 3.2 shows the 64-node surrounding box for a particle. Usually, the instances of a simulation are composed of agglomerates of particles, for this reason the computational cost of updating the 64 nodes is not significant compared to updating the eight surrounding nodes.

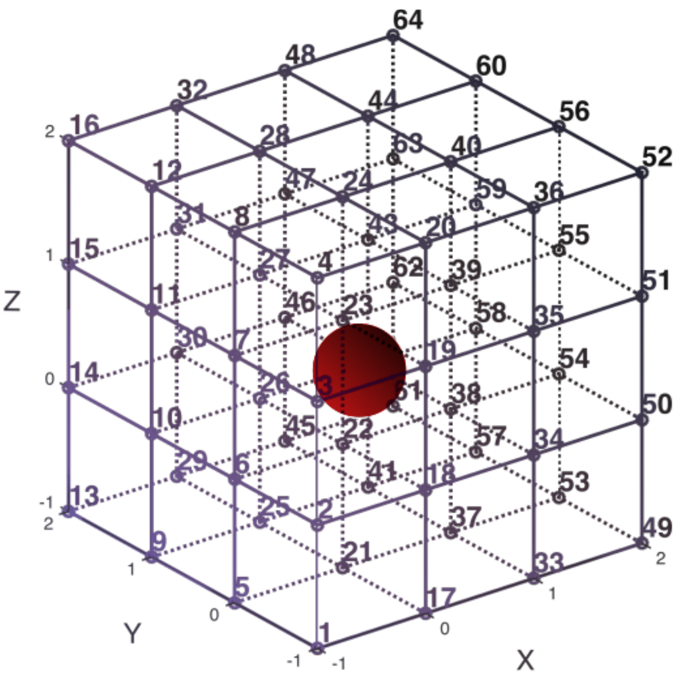


Figure 3.2: The expanded 64 surrounding nodes of a particle.

Before updating the self-gravity, the list of nodes to be updated is determined. This list is built based on the list of currently occupied nodes that is retrieved from the ESyS-Particle context. Using this list as base, the 64-node expansion of the occupied nodes is performed as shown in Fig. 3.2. Performing the expansion results in the determination of the expanded occupied nodes list, which is used when the acceleration is updated. Using the expanded list of occupied nodes results in a reduction in the computational cost of a simulation compared to updating the acceleration on every node of the self-gravity grid.

The nodes that belong to the list of nodes to be updated are assigned using a first-come first-serve policy to the available self-gravity threads. Load balancing is implicitly implemented by assigning the nodes of the list on demand, each time that there are computational resources available. This way, the implementation accounts for the different execution times of each calculation. The values of the acceleration are stored in memory to be used later, when the force on the particles is calculated.

3.4 Profiling the self-gravity calculation

After performing the implementation of the occupied cells algorithm, a profiling was performed using the VTune Amplifier tool by Intel (2017). The purpose of the profiling was to identify bottlenecks on the implementation, to be mitigated before the performance study.

The graphic in Figure 3.3 reports the results of the profiling for the nine most time consuming operations before implementing the improvements. The profiling was performed using the VTune amplifier tool by Intel and using the small instance of a two agglomerates scenario that is presented and described in chapter 5. The profiling test was executed for $10,000$ time steps. In Figure 3.3, routines are represented on the y-axis and the time consumed by the routines (in seconds) is shown on the x-axis. The time consumed by the operations is expressed in seconds. The total time consumed by a routine is represented by a column with two components: the time that the routine is idle waiting for results (grey bar) and the time that the routine is effectively processing (green bar). The grey bar represents when the CPU is assigned to the operation, but no instructions are executed. In turn, the green part reports when the CPU resources reserved for the execution are in use.

According to the profiling results reported in Figure 3.3, the most time consuming routine is `BoxCoords::getZ`, the routine that retrieves the z axis of the position for each particle. This routine is time consuming because it is called in every update of the acceleration for each node. In addition, the routine is also affected by an idle CPU utilization of 335 seconds. The idle CPU utilization could be associated to memory management issues such as memory transfer operations. The cause of this memory issues could be associated to the number of particles involved in a simulation. The large number of particles that is usually used in a simulation may need more memory to be maintained than the cache memory available in a processor. The time required to move the data through all the memory levels could produce the idle CPU time. `BoxCoords::getZ` is the first routine executed (in the self-gravity context) when accessing a node to process. Thus, the time required to transfer data

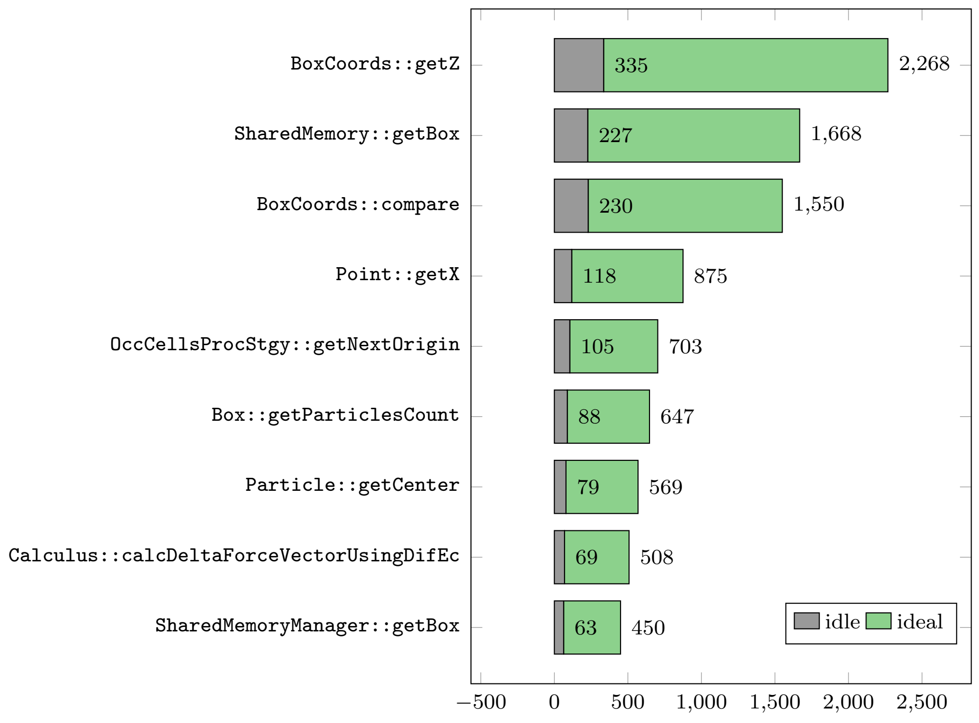


Figure 3.3: Profile report of the self-gravity code before implementing the occupied cells method.

from the main memory increases the total time spent by the routine. A similar behavior is detected for routines `Point::getX`, `SharedMemory::getBox`, `BoxCoords::compare`, `Box::getParticleCount`, and `Particle::getCenter`. The same arguments hold in these cases. A complete analysis on memory issues is beyond the scope of this thesis and is proposed as a line for future work.

The routine `OnlyOccupiedCellsProcessingStrategy::getNextOrigin` occupied the fifth place in terms of time consumption for the profiling performed. This routine is in charge of finding the coordinates of a new node to be updated when all the nodes assigned to a processor have been processed. The routine is time consuming because the next node to be updated is searched through the list of all the nodes. Another time consuming operation is `calculateDeltaForceVectorUsingDifEcuations`, whereas this operation was not selected as a routine to improve. The reason is that the routine implements simple mathematical equations, so the optimizations would have little effect compared to the optimizations implemented on the rest of the aforementioned routines.

According to the aforementioned comments, the performance optimizations were focused on the improve of the routine `OnlyOccupiedCellsProcessingStrategy::getNextOrigin`. This routine performs a search over all the nodes of the grid to find a node to update. The improvement consists on iterating only over the list of occupied nodes that is created every time self-gravity needs to be updated. It is usual that in the scenarios used to perform simulations, the majority of the space is empty. So, the list of occupied cells is significantly shorter than the list of nodes. This way, the routine `OnlyOccupiedCellsProcessingStrategy::getNextOrigin` is called less times using the occupied cells strategy compared to the version before the improvements. The described improvement also affects the overall performance of the routines `BoxCoords::getZ`, `Point::getX`, `SharedMemory::getBox`, `BoxCoords::compare`, `Box::getParticleCount`.

3.5 Implementation on ESyS-Particle and the self-gravity module

This section describes the adaptations performed on ESyS-Particle to implement the occupied cells method. The adaptations include changes on ESyS-Particle and also on the self-gravity module. Algorithm 1 shows a summary of the occupied cells algorithm.

Algorithm 1 Occupied cells algorithm

```
1: procedure RECALCULATE SELF-GRAVITY
2:     occupied cells list ← getOccupiedCells()
3:     expanded occupied cells ← getBoxRecalculateMethod(occupied cells list)
4:     for each occupied cell in expanded occupied cells do
5:         updated value ← updateSelfGravity(occupied cell, occupied cells list)
6:         add(updated cell list, updated cell)
7:     communicateNewValues(updated cell list)
```

The process of adapting ESyS-Particle included to request the developers for the implementation of a new functionality which consists of a function that retrieves the list of occupied cells of the ESyS-Particle overlapping grid in a given moment. The request for the new functionality was included as an issue in a new iteration of the development of ESyS-Particle and was in time available to be used by the self-gravity module. The function name is `getOccupiedCells()` and was included as a method that belongs to the class

`MPSelfGravityMaster` in the module `Model` of the simulator. Line 2 of algorithm 1 corresponds to the call to the `getOccupiedCells()` method.

Along with `getOccupiedCells()`, `getBoxRecalculateMethod()` was implemented. The method is called on line 3 of Algorithm 1. This method takes as input the vector containing the occupied cells that is obtained as output after the execution of the `getOccupiedCells()` method. The output of the `getBoxRecalculateMethod()` method is the vector of boxes to recalculate, that is the 64-node expansion of the occupied nodes vector. The expansion is performed individually for each node. Then, an iteration is performed over the nodes of the individual expansion, and only the nodes that are not present in the general expansion are added to it. After the general extended vector is created it is used as input to perform the update of the gravitational potential of the nodes. The term general refers to the date structure where the information of the system is located.

The self-gravity is updated for each of the cells that belong to the occupied cells list. This process is summarized from lines 4 to 6 of Algorithm 1. The self-gravity of a cell is calculated using the occupied cells list rather than the expanded occupied cells list, because the latter may contain empty cells. Before implementing the occupied cells method, the self-gravity was updated for the full set of boxes that compose the space of a simulation. In the previous implementation, processing the set of boxes consisted of a triple iteration, one for each dimension of the three-dimensional space. After the self-gravity is updated for a cell, the calculated value is added to the updated cells list.

When the algorithm finishes recalculating the self-gravity for all the cells of the expanded cells list, the updated values are communicated to the main module of ESyS-Particle. The procedure `recalculate self-gravity` of Algorithm 1 is called each time that the self-gravity needs to be updated during a simulation.

Chapter 4

Adapted Barnes-Hut method for self-gravity calculation

This chapter describes the characteristics and properties of the Barnes & Hut octal tree used for the self-gravity calculation in ESyS-Particle. In addition, the process of creation of the tree and self-gravity calculation is described. A binary implementation of the tree is presented, alongside a comparison to the octal tree implementation.

4.1 Octal tree structure

The Barnes & Hut tree is implemented as an octal tree in which the root represents the complete space used for the simulation. Leaf nodes of the tree are the boxes of the self-gravity grid. Every non-leaf node has eight sons that have the same size. This way, the space represented by the tree is of cubical shape. Each node also has the following information: the position of the center of mass, the total mass, the spatial coordinates, the coordinates in the self-gravity grid, the level number, the number of particles in it, and an integer that identifies the node in the level it belongs. All nodes of a level are numbered from 0 to $n-1$ being n the number of nodes of the level. The identifiers (id) are assigned to the nodes so that the id of the father of a node satisfies that $id_f = id_s/(10_8 \times (level_s/level_f)$, where id_x is the identifier of the node, $level_x$ is the level of the node, the underscore f denotes a father node and the underscore s denotes a son node. The underscore '8' denotes that the number is in octal base. This way, the procedure applied to know if a node is son of another is a constant time operation performed in $O(1)$. Dividing by 10_8, the identifier of a node is equivalent to performing a shift operation of three bits to the right. Figure 4.1 shows a sample two-dimensional tree partition created for an agglomerate of particles and an illustration example of the partitioning of the space created with the Barnes-Hut method adaptation that is proposed in this thesis. The grid over the agglomerates represents the quadrupole (octapole in three dimensions) tree that is the result of the application of the method of creation of the self-gravity tree. The resolution of the partition is not increased on the nodes that have no particles, by stating that the tree node created is empty after its creation.

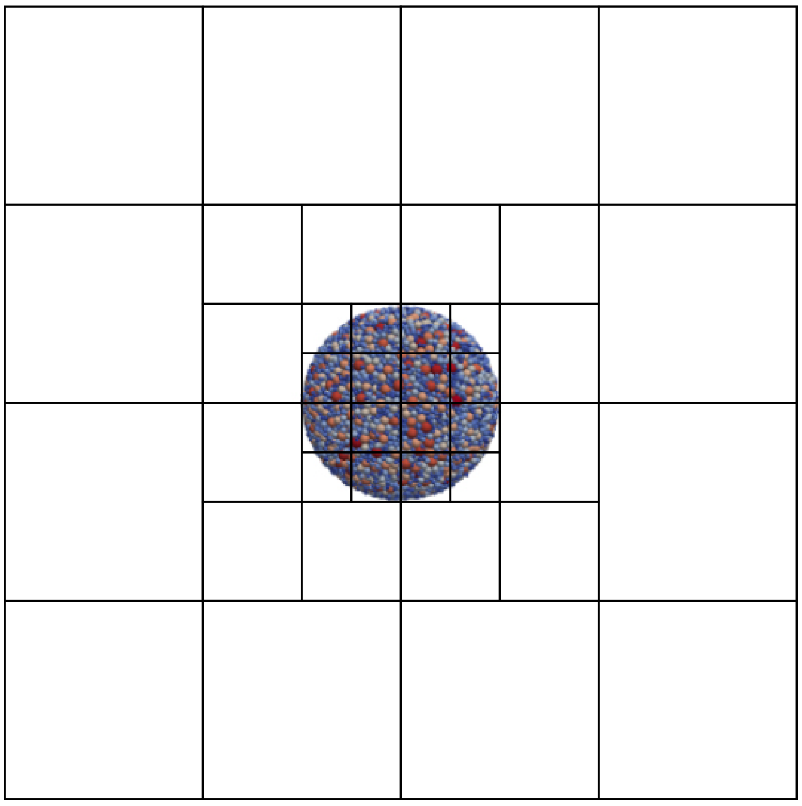


Figure 4.1: Example of tree partition for an agglomerate of particles (a two dimensions projection is shown for better representing the division method).

4.2 Process of creation of the octal tree

The Barnes-Hut method is based in the use of octal trees to divide the space of a scenario of a simulation. The implementation of the Barnes-Hut algorithm presented in this thesis consists of instantiating one octal tree for the complete space of an scenario of a simulation. The octal tree is created and disposed every time the gravitational potential is updated.

The self-gravity update process consists of four steps: i) creating the Barnes & Hut tree applied for the computation (the *self-gravity tree*); ii) building a list of tree nodes for each of the boxes that contain particles (i.e.: *occupied nodes*); this is the list of Barnes & Hut tree nodes that affect the potential of the box to be updated; iii) effectively computing the self-gravity of the occupied nodes using as input the list of tree nodes created in the previous step; iv) finally, deleting the octal tree after updating the potential of the nodes. These steps are explained next.

Algorithm 2 describes the process of calculating the self-gravity potential using the proposed Barnes & Hut algorithm.

Algorithm 2 General octal tree algorithm

```
 1: procedure RECALCULATE SELF-GRAVITY
 2:     occupied cells list ← getOccupiedCells()
 3:     expanded occupied cells ← getBoxRecalculateMethod(occupied cells list)
 4:     octal tree ← createOctalTree()
 5:     calculateCentersOfMass(octal tree)
 6:     for each occupied cell in expanded occupied cells do
 7:         list of tree nodes ← createListOfNodes(occupied cell, octal tree)
 8:         add(list of list of tree node, list of tree node)
 9:     for each occupied cell in expanded occupied cells do
10:         updated value ← updateSelfGravity(occupied cell, list of list of tree nodes)
11:         add(updated cell list, updated cell)
12:     communicateNewValues(updated cell list)
```

In Algorithm 2, lines 2-3 correspond to the process of creation of the algorithm, which is the creation of the expanded occupied cells list. The creation of the list comprises two activities: obtaining the occupied cells list, and creating the expanded occupied cells list from the occupied cells list. The creation of the expanded list consists of performing the 64-node expansion of each of the occupied nodes, but only adding those nodes that belong to the expansion that were not already added to the expanded list. Afterwards, the first step in the calculation of the self-gravity is executed: the octal tree is created and the center of mass of each node of the tree is calculated. The first step is represented in lines 4-5 of Algorithm 2. The second step, the creation of the tree node list of each of the expanded occupied cells, comprises the for loop in line 6 of the algorithm. Then, the third step, which is to actually calculate the potential of the nodes, is performed in the for loop in line 9 of Algorithm 2. The output of step 3 is the new potential of the expanded occupied cells list. That list is communicated to the worker nodes to calculate the potential of the particles.

Creation of the self-gravity tree.

The process of creating a Barnes-Hut tree starts with the instantiation of a root node. The root node represents the complete space defined for the simulation. Due to the nature of the octal tree structure proposed by Barnes and Hut, the space to perform the simulation is cubic shaped. As a consequence, the root node also has cubic shape. After the root node is instantiated, the tree levels are created sequentially. Each new level is created by performing a spatial partition of each of the nodes that belong to the immediate upper level. An expansion of a node is created if that node has at least one particle in it. The new nodes are the child nodes of the node from which they were created by performing the spatial partition. The spatial partition consists of creating eight child nodes by partitioning the space of the father node in eight equal cubic parts. The process of spatial partitioning ends when the node to expand has the same size of a box of the grid used for self-gravity computation, or if the node to expand has no particles.

A node of the Barnes-Hut tree represents a (cubical shaped) part of the space of its father node. For that reason, a node of the tree is represented as a structure which stores the coordinates and also the edge size of the node. In addition, the structure of a node holds the following data: an identifier, the number of particles in the node, the total mass, and the position of the center of mass of all the particles contained inside the node. The identifier is assigned during the creation of the node and is composed by two numbers: the level of the tree where the node belongs, and the number of the node, which is unique in the context of the level. The root node is identified with the number 0 and belongs to the level 0. The identifier is used to establish the location of a node in a given level of a tree. In this way, the identifier is reseted to 0 in every level of the tree. With this identification system, having the level of a node and its identifier as input data, the procedure to know if a node is son of another constitutes an operation of O(1) execution time. This property of the Barnes-

Hut tree implemented improves the time of the calculation of the self-gravity potential at the nodes. After the creation of the tree, the centers of mass of the nodes are calculated. The process of calculation of the centers of mass is bottom up, from the leaves up to the root node. The centers of mass for the leaf nodes are calculated directly from the particles, whereas for the nodes of the upper levels the centers of mass are calculated from their respective son nodes. The center of mass is calculated only for the nodes that have particles. Figure 4.2 shows a sample octal tree created using the described algorithm for a cube composed of 64 boxes. As an example, the center of mass for the node located in the upper left part of the Figure 4.2 is not calculated because it has no particles.

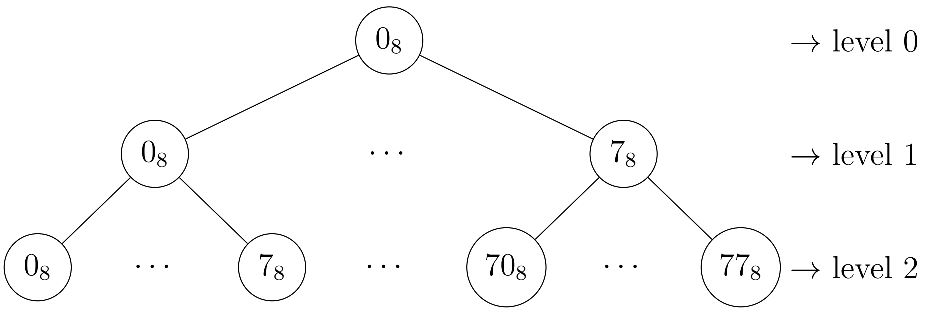


Figure 4.2: Sample of octal tree created using the described algorithm for a self-gravity grid composed of 64 boxes.

Algorithm 4 summarizes the process of creation of the octal tree that was described in the previous paragraphs.

In Algorithm 4, the height of the tree is calculated in line 2 before its creation. The octal tree is created by level, so the height calculated is then used in line 3 to create a queue of node levels that is used as an auxiliary structure in the process of creation of the tree. Then, the creation of the tree starts with the instantiation of the root node in line 4. After that, in line 5 the root node is added to the level queue position that corresponds to the level 0 of the tree. The rest of the algorithm consists of using the created auxiliary structure to create the tree nodes and organize them.

Lines 7-12 of algorithm 4 correspond to the core process of the creation of the octal tree. The process consists of two loops: the outer loop performs an iteration over the level of the tree from the root to the box nodes, whereas the inner loop iterates over all the nodes of a level. In the outer loop, the only operation performed is the change of level. In the inner loop, for each node, the list of sons that correspond to that node is created (line 10). Then, the

Algorithm 3 Octal tree creation algorithm

```
1: procedure CREATE OCTAL TREE()
2:     tree height ← calculateTreeHeight(dimension, box length)
3:     node level queue ← createQueueOfLevels(tree height)
4:     root node ← createNode()
5:     node level queue.at(0).push(root node)
6:     # Create octal tree
7:     for tree level = 0 to tree height - 1 do
8:         lower level queue ← node level queue.at(tree level + 1)
9:         for each node in upper level queue do
10:            lower level node sons ← createSonsList(node queue)
11:            lower level queue ← add(lower level node sons)
12:        node queue ← lower level queue
13:    root node ← getFront(node queue)
14:    # Calculate particle count and center of mass for the tree nodes
15:    for tree level = tree height - 2 to 0 do
16:        level node queue ← node level queue.pop()
17:        for each node in level node queue do
18:            calculateParticleCount(node)
19:            mass center ← calculateMassCenterOfNode(node)
20:    return root node
```

list is associated to the father node (line 11). The inner loop finishes adding the son nodes recently created to the lower level queue.

Lines 15-19 of algorithm 4 correspond to the process of counting the particles and also calculating the center of mass of the nodes. The previously created octal tree is covered by level from the nodes of the lowest level (i.e., the nodes that match with the boxes) up to the root node. The results of particle count and mass center of the lower level nodes are used to calculate the values for the upper levels.

Finally, in line 20 of algorithm 4 the root node is returned as the representant of the complete octal tree. The tree is returned in this format because in all cases where the octal tree needs to be covered, the process always starts from the root node to the box nodes.

Algorithm 5 summarizes the process of creation of the eight sons of a not leaf node of an octal tree. The process start by creating the list of sons to be returned at the end of the procedure. The id of the first son is calculated by performing a shift of 3 places to the father id. The numeration strategy was previously explained in this chapter. The size of all the nodes is the same

Algorithm 4 Octal tree creation algorithm

```
1: procedure CREATE OCTAL TREE()
2:     tree height ← calculateTreeHeight(dimension, box length)
3:     node level queue ← createQueueOfLevels(tree height)
4:     root node ← createNode()
5:     node level queue.at(0).push(root node)
6:     # Create octal tree
7:     for tree level = 0 to tree height - 1 do
8:         lower level queue ← node level queue.at(tree level + 1)
9:         for each node in upper level queue do
10:            lower level node sons ← createSonsList(node queue)
11:            lower level queue ← add(lower level node sons)
12:        node queue ← lower level queue
13:    root node ← getFront(node queue)
14:    # Calculate particle count and center of mass for the tree nodes
15:    for tree level = tree height - 2 to 0 do
16:        level node queue ← node level queue.pop()
17:        for each node in level node queue do
18:            calculateParticleCount(node)
19:            mass center ← calculateMassCenterOfNode(node)
20:    return root node
```

and is half the size of the father node. Finally, for each of the eight sons, the position of the node is calculated, then the node is created and added to the *sons list.*

Algorithm 5 Create list of sons

```
1: procedure CREATE SONS LIST(node)
2:     sons list ← new List()
3:     first son id ← node.id < < 3
4:     son node dimension ← node.size/ 2
5:     for i = first son id to first son id + 8 do
6:         calculate son node position (x, y, z, son node dimension)
7:         node ← create node (x, y, z, i)
8:         sons list.add(node)
9:     return sons list
```

Creation of the list of tree nodes.

After creating the expansion tree, the second step in the simulation is to create a list of tree nodes for each of the occupied nodes of the grid, called *objective* *nodes*. To help achieving that purpose, the concept of *neighborhood* of a node is introduced. A neighborhood of a specific node is conformed by its surrounding nodes that do not exceed a certain distance threshold. The distance threshold that delimits the neighborhood is modeled as a parameter that is user defined. Each list of tree nodes is composed of the highest level nodes that are not father of any node that belongs to the neighborhood. For each objective node, the algorithm that creates each tree node list starts with the root node. A node is added to the tree node list if it is not father of any member of the neighborhood. Otherwise, if the node is father of at least one member of the neighborhood, then the sons of the node are added to a queue to be evaluated later as candidates to the tree node list. The list of tree nodes is used as input for the third step, that is the update of the potential of a node.

Algorithm 6 presents a summary of the `getProcessableNodes` routine. This procedure implements the process of creation of the list of processable nodes for the neighborhood of an objective node. The procedure starts by creating a list of neighboring nodes from the objective node and the neighborhood radius, that is user defined before starting a simulation. Then, the octal tree is covered, starting from the root node, with the objective of creating the list of processable nodes (lines 6-14). The covering consists of obtaining a node from the queue of nodes to process and then checking if the node is father of any node in the neighborhood of the objective node. If the node being processed is not a father of any node in the neighborhood, then the node is pushed to the list of processable nodes, else the sons of the node are added to the queue of nodes to process.

Self-gravity calculation.

The final step is to calculate the total self-gravity force vector for every node of the occupied nodes list. The total self-gravity force is calculated based on the lists built in step two, instead of using the occupied cells that are used on the baseline implementation. After updating the potential for each objective node, the new vector values are transferred to the main force calculation module to be integrated with the contact forces. Algorithm 7 presents the self-gravity calculation process of an occupied node. The process is called after the creation of the octal tree, and after the list of processable nodes for the occupied node neighborhood has been created. In the algorithm, for each of the processable

Algorithm 6 Get processable nodes

```
1: procedure GET PROCESSABLE NODES(objective node, octal tree, neighborhood radius)
2:     neighbor nodes ← get neighbor cells(objective node , neighborhood radius)
3:     processable nodes ← create queue()
4:     node queue ← create queue()
5:     node queue.push(octal tree)
6:     while !node queue.isEmpty() do
7:         node ← node queue.pop()
8:         if is father of neighborhood member(node, neighbor nodes) then
9:             if !node.is box() then
10:                node queue.push(node.get sons queue())
11:            else
12:                processable nodes.push(node)
13:        else
14:            processable nodes.push(node)
15:    return processable nodes
```

nodes, the force vector with respect to the occupied node is calculated and added to the *total force vector*. The *total force vector* is then returned to be broadcasted to all the MPI worker processes of ESyS-Particle.

Algorithm 7 Self-gravity calculation of a node

```
1: procedure UPDATE SELF-GRAVITY(occupied node, octal tree, processable nodes)
2:     total force vector ← createEmptyForceVector()
3:     for each processable node in processable nodes do
4:         force vector ← processBarnesHutNode(occupied node, processable node)
5:         sumForceVector(total force vector, force vector)
6:     return total force vector
```

4.3 Implementation of the Barnes and Hut method on ESyS-Particle

The self-gravity module on ESyS-Particle was extended to implement the Barnes & Hut method. The extension process added two new classes: the `Barnes_And_Hut_Node` class, and the `Barnes_And_Hut_Manager` class. The `Barnes_And_Hut_Node` class implements the basic functionalities of the node.

This class is responsible for the creation of a new node. It holds the getters and setters of the attributes of the class, and the dispose functions. The `Barnes_-And_Hut_Manager` class hosts the core functions of the Barnes & Hut method. This class is responsible of the algorithm that creates and disposes of the octal tree in each update of the self-gravity. It implements the algorithm that creates the list of tree nodes, and the algorithm that retrieves the list of neighbors of an objective node. The self-gravity calculation code in ESyS-Particle was adapted to use the list of nodes retrieved from the `Barnes_And_Hut_Manager` instead of using the list of occupied nodes that is used by the occupied cells method in the standard self-gravity module.

4.4 The binary tree

This section presents the binary tree. The changes introduced in the octal tree algorithm to implement the binary tree and the main differences between both implementations are described.

Structure and process of creation of the binary tree

Figure 4.3 shows a sample binary tree for a self-gravity grid composed of 64 boxes. The generated tree has seven levels, including the root level. Each node has a unique number that identifies it in the corresponding level, which is an integer in binary code. A node is the father of another node of the binary tree if it satisfies the condition that $id_f = id_s/(10_2 \times (level_s/level_f)$, where id_x is the identifier of the node and $level_x$ is the level of the node. The underscore 2 denotes that the number is in binary base. This way, the procedure to know if a node is son of another is an operation of $O(1)$. This condition is analog to the condition used in the octal tree to check if a node is father of another node. Instead of dividing by 10_8, the division is performed by 10_2 which comprises a shift operation to the right.

To build the tree, the space represented by a node is divided in two by its largest axis. So, the partitions are not necessarily cubic. This way, the binary tree has the advantage that the space represented does not need to be cubic. Performing the partitions over the largest axis guarantees that the leaf nodes are of the same size and position of the self-gravity grid boxes.

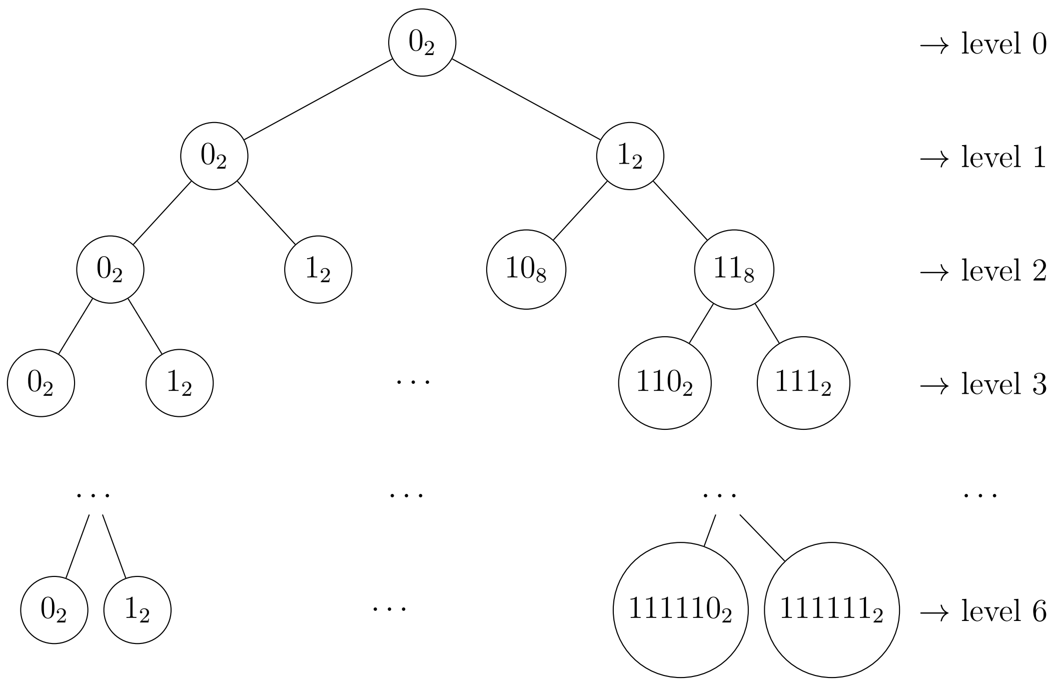


Figure 4.3: Example of enumeration for a binary tree with seven levels for a self-gravity grid composed of 64 boxes.

Comparison of the binary tree and the octal tree

Node 77_8 of Figure 4.2 is taken as an example to perform the comparison of the trees. This node corresponds to 111111_2 in the binary tree. Assuming that all the boxes are occupied and the neighborhood size is zero, in the binary tree in Figure 4.3 the list of tree nodes is comprised of node 0_2 (level 1), node 10_2 (level 2), node 110_2 (level 3), node 1110_2 (level 4), and node 11110_2 (level 5). In this example, the list of tree nodes has five elements. On the other hand, the list of tree nodes of the octal tree has 13 elements. Despite having more levels, the list of tree nodes for the binary tree has fewer elements then the octal tree and the resolution of the partition for the binary tree grows slower when moving closer to the objective node.

Regarding the implementation of the algorithm in ESyS-Particle, instead of performing a shift of tree bits, the shift is performed in one bit to multiply the nodes identifiers by 2. Another difference in the implementation of the binary tree with respect to the octal tree implementation is the tree height calculation process. The calculation process involves the three axis instead of only considering the x axis as in the octal tree. The largest of the three axis is divided by two up to the stage where the three axis have the same size of a box. Each time that the division is performed, the level count is incremented by one.

Except for the aforementioned differences, the structure is the same as the octal tree. The algorithm to update self-gravity in the binary tree is the same as the octal tree. After creating the lists of nodes for the occupied nodes, the gravitational potential is calculated and delivered to the ESyS-particle module.

4.5 Increasing the numerical accuracy of the octal tree algorithm

This section describes a different approach to increase the numerical accuracy of the simulations using the octal tree strategy. This approach aims to diminish the error in the calculation of the potential at the nodes by increasing the precision of the calculation of the center of mass of the tree nodes. The increase in the precision of the center of mass is intended to be achieved by calculating the center of mass in higher levels of the tree using the particles, rather than using the potential of the lower levels.

Each node of the octal tree is modeled as a point like particle. For the particular case of nodes that belong to the lower level of the tree, the center of the node is located at the center of mass of the particles that belong to the space delimited by the node. Then, the total mass of a node is the sum of all the nodes of the particle. The center of mass and the total mass of the nodes that belong to the higher levels of the tree are calculated using the values obtained for the son nodes of the immediate lower level. This characteristic means that the center of mass of the higher level is calculated based on a previous calculation of a center of mass. A consequence of using this approach is a loss of precision of the calculation of the potential.

The octal tree algorithm was modified in order to prove that the precision of the algorithm increases by calculating the center of mass in higher levels of the tree. The modification of the algorithm consisted of calculating the center of mass using the particles for up to the level $n-1$ of the tree, which is the first level of nodes that have son nodes.

Chapter 5

Description of the experimental setup

This chapter describes the test scenarios and the instances used in the experimental evaluation of the parallel self-gravity calculation implemented in ESyS particle. Also, the hardware platform used to perform the tests is presented. Then, the profiling results of the optimized version are presented and discussed.

5.1 Two agglomerates scenario

The first test scenario is composed of two agglomerates of particles. Both agglomerates have the property that each particle of one agglomerate has an identical copy of it on the other agglomerate (but the position of both particles is central mirrored). The agglomerates of particles are symmetrical with respect to the origin of the coordinates system (point (0,0,0)). This way, the center of mass is located in the point (0,0,0) as well, and it is halfway the center of mass of each agglomerate.

The creation of the test scenario starts by the generation of one agglomerate with the tool GenGeo that is included in ESyS-Particle. After that, a central symmetry is performed so that the agglomerates are separated by five kilometers. This way, the center of mass of the two agglomerates together is located halfway to the center of mass of the agglomerates taken separately. The collisions between the particles are configured to be pure elastic. The initial speed of the particles is set to be 5 m/s. The velocity has opposite direction for each agglomerate. The velocity direction is also tangential to the Z axis and is perpendicular to the line that passes through the center of mass of each agglomerate. The density of the individual particles is 3000 g/cm^3.

Table 5.1 presents details about the instances generated based in the two agglomerates scenario. Three instances of the two agglomerates scenario were created to perform the study of the computational efficiency of the self-gravity implementations presented in this article. The first instance defined is a small instance, composed of $3,866$ particles, each one having a radius from 50 m to 100 m. The second instance is a medium size instance with $11,100$ particles with a radii from 35 m to 70 m. Finally, the large instance is composed of $38,358$ particles with a radii from 20 m to 60 m. The mass of the instances is not the same for all the instances and oscillates from 1.2×10^{12} kg to 1.7×10^{12} kg. However, the masses of the instances are of the same order of magnitude. Figure 5.1 shows a representation of the large instance of the two agglomerate scenario. The space was configured to be of cubic form and measuring 4096 m in each axis direction. For the small instance, the box length is 256 m long, and for the medium and large instances the box length is 128 m long.

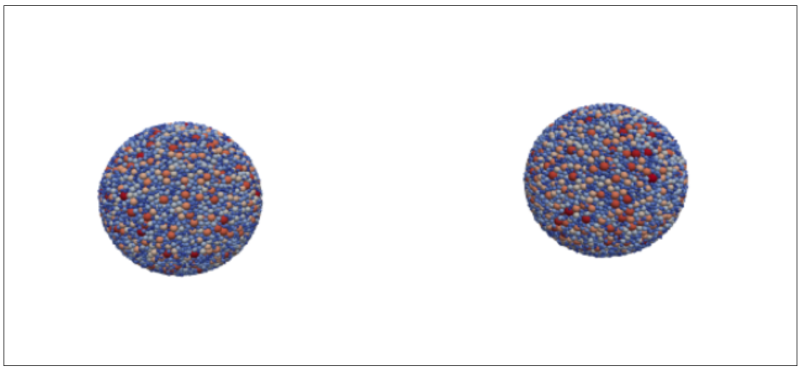


Figure 5.1: The two agglomerate scenario

Table 5.1: Instances generated based on the two agglomerates scenario.

instance name	*# particles*	*particle radius (m)*	*# octal tree levels*	*# binary tree levels*
small instance	3,866	50-100	6	16
medium instance	11,100	35-70	7	19
large instance	38,358	20-60	7	19

All instances were simulated for $100,000$ time steps of 0.01 seconds long. The simulations were executed using a varying number of computational resources to evaluate the speedup and scalability of the proposed implementation. In all simulations, the self-gravity is updated after at least one particle has moved more than a certain distance threshold that was configured to be two times the radius of the biggest particle. So, the execution of the scenarios using small particles will have more updates of the self-gravity than the instances with large particles.

The size of the grid box in ESyS-Particle must satisfy $box_l \geq 2 \times r_{max}$, where box_l is the box length and r_{max} is the maximum radius of a particle. Using a bigger box implies lower accuracy of the calculations. So, the value of box_l has to be as close as possible to $2 \times r_{max}$ and also be a power of two. This way, the box size for the small instance is 256 m, and for both medium and large instances is 128 m. The total number of boxes for the small instance is 32,768, while for medium and large instances the number of boxes is 262,144.

For the small instance, the octal tree has six levels and 37,449 nodes. On the other hand, the binary tree for the small instance has 16 levels and 65,535 nodes. For the medium and large instances the octal tree has seven levels and 299,593 nodes, while the binary has 19 levels and 524,287 nodes. So, for all instances executed in this work, the memory used by the binary tree is roughly twice the memory used by the octal tree. This feature shows that the octal tree can scale to a larger number of boxes compared to the binary tree. Simulations were executed for 10,000 time steps of 0.01 seconds each (a total time of 100 seconds). The neighborhood was configured to be of length five. This way, when creating the list of nodes that correspond to an objective node, the defined neighborhood is a cube of 11 boxes long centered in the objective node.

5.2 Free falling symmetric cube

The second scenario consists of a cube of 1 km on each side, filled with spherical particles that have the same radius and density. The particles that conform the cube are separated from one another at least a distance that is equivalent to 1/6 of a particle radius. In addition, the particles are located in a cubical box. The dimension of the cubical box is determined beforehand. Also, the particle radius is bounded by the dimension of a box. The particle radius is 1/2 of a box edge. The particles are located in a way that the resultant cube is symmetric respect to its center. As a consequence, the center of mass of the cube matches the geometrical center of the cube.

The cube is generated in two stages: i) a small cube is filled with particles at random locations bounded to the constraints previously detailed in the paragraph, ii) then, the cube is copied eight times to create a bigger cube. The copies are arranged in a way that satisfies that the center of mass of the bigger cube is located at the position (0,0,0) of the space. The particles are given an initial speed of $0\,\mathrm{m/s}$ and a density of $3000\,\mathrm{g/cm^3}$. In this scenario, the cube should collapse in the direction of its center of mass.

Table 5.2 presents the details about the instances generated based in the free falling symmetric cube scenario. The table reports the instance name, the total number of particles, the particle radius, and the free-fall time for each instance.

Table 5.2: Instances generated based on the free falling symmetric cube scenario.

instance name	*# particles*	*particle radius (m)*	*free fall time (s)*
small cube instance	32,768	10.42	3079
medium cube instance	262,144	5.21	3079
large cube instance	2,097,152	2.60	3088

The free-fall equation of a sphere is defined by Binney and Tremaine (2011), and it is used to calculate the time needed for an spherical agglomerate of particles to collapse under its own gravity. This equation is valid under the supposition that the agglomerate is only affected by its own gravity (i.e., the gravity generated by its own particles).

Three scenarios were generated with different number of particles, classified in small, medium, and large. The cube for all the instances created has 1 km long on each side, so the particles radii are smaller as the particle number increases. The small scenario has 32,768 particles, the medium scenario has 262,144 particles, and the large scenario has 2,097,152 particles. Figure 5.2 shows an example representation for the small instance of the cube scenario.In the case of the executions using the Barnes & Hut method for self-gravity calculation, the neighborhood is composed of the nodes that are at least at $3\times$ the size of a box.

The box size for the small instance is 256 m, and for both medium and large instances is 128 m. The total number of boxes for the small instance is 32,768, while for medium and large instances the number of boxes is 262,144.

For the small instance, the octal tree has six levels and 37,449 nodes. On

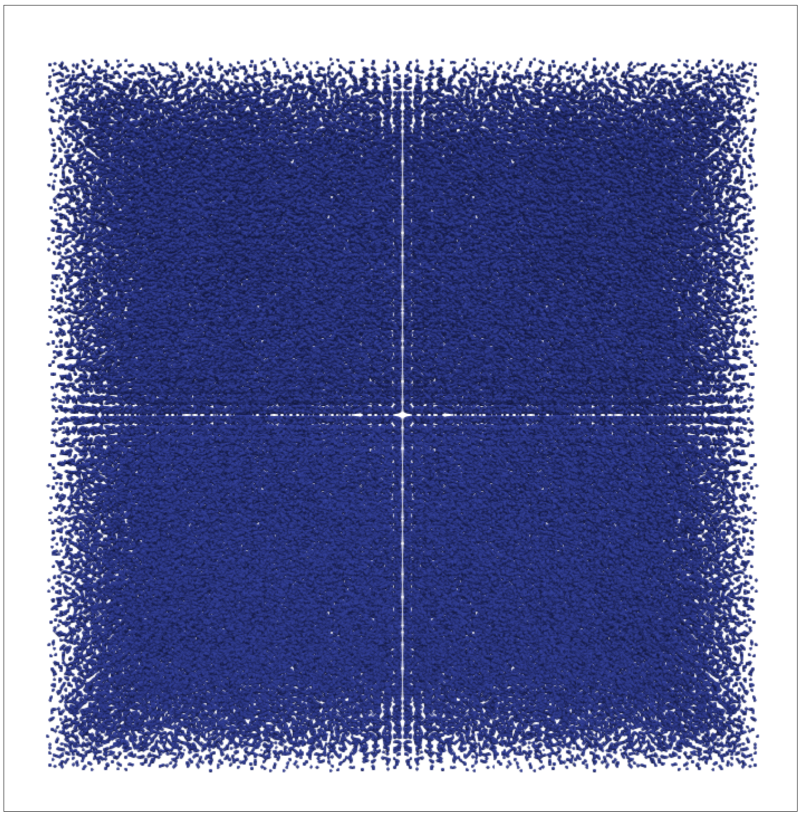


Figure 5.2: Example of tree partition for the cube scenario. The example corresponds to the initial state of the small instance of 32,768 particles.

the other hand, the binary tree for the small instance has 16 levels and 65,535 nodes. For the medium and large instances the octal tree has seven levels and 299,593 nodes, while the binary has 19 levels and 524,287 nodes. So, for all instances executed in this thesis, the memory used by the binary tree is appoximately twice the memory used by the octal tree. This feature shows that the octal tree can scale to a larger number of boxes compared to the binary tree. Simulations were executed for 10,000 time steps of 0.01 seconds each (a total time of 100 seconds). The neighborhood was configured to be of length five. This way, when creating the list of nodes that correspond to an objective node, the defined neighborhood is a cube of 11 boxes long, centered in the objective node.

5.3 Hardware platform

ESyS-Particle allows the execution of realistic scenarios and instances that can scale up to millions of particles. The execution of large size scenarios requires a large number of calculations that translate into many hours of CPU use. This makes a cluster of high-performance computers a neccesary environment to run realistic simulations in a reasonable amount of time.

In addition, both ESyS-Particle and the self-gravity module are implemented to take advantage of both parallel and distributed computing environments. This is because in ESyS-Particle a two-level parallelism is implemented. On the one hand, the main module of ESyS-Particle, in which the contact forces are calculated, uses a distributed algorithm to improve the performance of the simulations. In the distributed algorithm, a spatial partitioning of the simulation domain is performed. This partitioning is static and is performed at the beginning of the simulation. After this, the partitions communicate only with those that are adjacent to share information on particle migration. In this way, it is guaranteed that communication is carried out efficiently, consuming the least amount of time possible. Efficient communication allows most of the time to be used for effective calculations.

On the other hand, the self-gravity calculation module uses a parallel algorithm to achieve the high performance necessary to perform simulations in a reasonable execution time. The parallel programming techniques used in the self-gravity calculation module include the partitioning of the workload to be performed and the assignment of that workload to a pool of threads. The threads are assigned a part of the total workload, when they finish the execution of a part they are assigned another part that has not yet been processed. This process is repeated until the entire workload is completed. The workload to be partitioned is the calculation of the potential on the particles at the time of updating the self-gravity values. Unlike the static spatial partition used by the main module (where the contact forces are calculated), the workload in the self-gravity module is dynamically distributed among the available threads.

The experimental evaluation was performed on a AMD Opteron Magny Cours Processor 6272 @ 2.09GHz, with 64 cores and 48GB of RAM. The server is part of Cluster FING, the High Performance Computing facility from Universidad de la Republica, Uruguay, Nesmachnow (2010).

5.4 Profiling the optimized version of self-gravity calculation

After performing the implementation of the occupied cells algorithm, a profiling was performed using the VTune Amplifier tool by Intel (2018). The purpose of the profiling was to identify bottlenecks on the implementation, to be mitigated before the performance study. The profiling was performed using the small instance of the two agglomerates scenario and was executed for $10,000$ time steps.

Figure 5.3 graphically summarizes the results of the profile performed to the self-gravity calculation after implementing the occupied cells method. In the figure, the X axis represents the time spent by each routine in the implemented method (in seconds). In addition, the Y axis reports the eight most time consuming routines during a simulation. The time spent by each routine is represented by a column which in turn is composed by two subcolumns. These subcolumns represent different utilization level of the CPU. The yellow subcolumn, which was named as *ok* CPU usage by Intel (2018) and represents an utilization level of the CPU between 51 % to 85 %. Whereas, the green subcolumn (the *ideal* CPU usage as defined by Intel (2018)) represents an utilization level of the CPU from 85 % to 100 %.

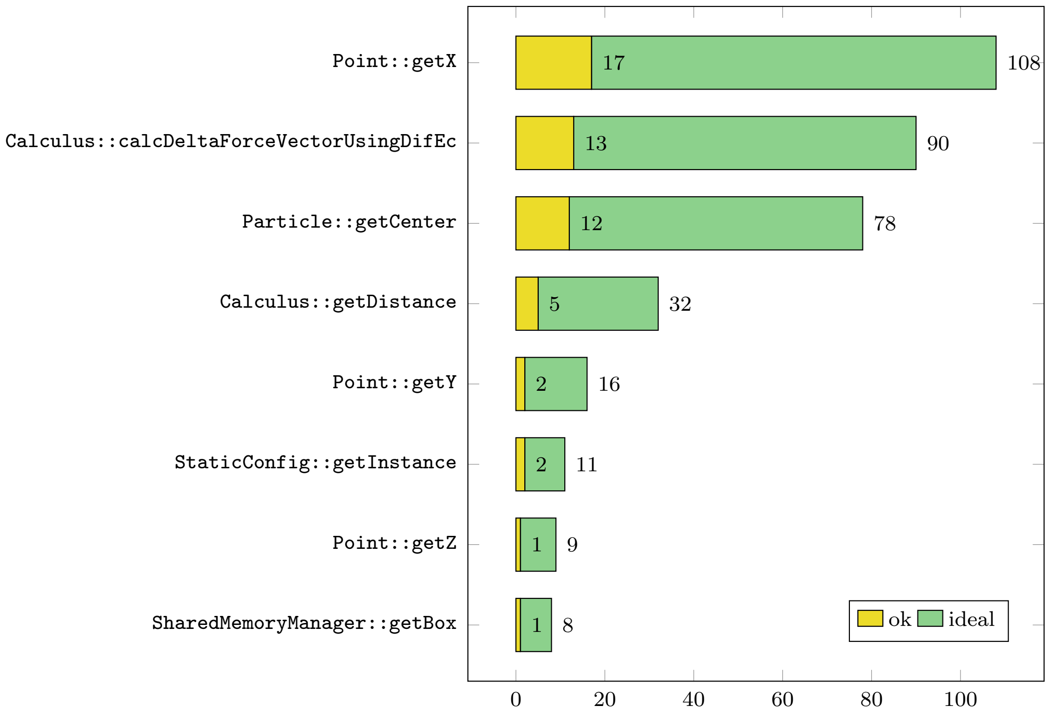


Figure 5.3: Profile report using the occupied cells method.

To highlight the performance improvements, Table 5.3 reports the execution time of the eight most time consuming routines identified before the implementation of the occupied cells method and a comparison of the time consumed by those routines after the implementation of the occupied cells method.

Results in Table 5.3 show that the method `BoxCoords::getZ`, which consumed 2269 seconds in the non-optimized version, has a negligible contribution to the execution time in the occupied cells implementation. A similar behavior is identified for routines `SharedMemory::getBox`, `BoxCoords::compare`, `SharedMemoryManager::getBox`, and `Box::get- ParticlesCount`. The modifications in routine `OnlyOccupiedCellsProcessingStrategy::getNextOrigin`, that searches for the next node to process, improve the execution time from consuming 704 seconds to consume a negligible time compared to the other routines. Finally, improvements on routine `Point::getX` allow the routine to execute 8 times faster (from 876 seconds to 109 seconds). These results account for notably performance improvements when using the new proposed schema for computing self-gravity. The new implementation allows scaling up to

Table 5.3: Comparison of the most time consuming routines before and after implementing the performance improvements.

routine	*execution time (s)*	
	before	*after*
`BoxCoords::getZ`	2269	*negligible*
`SharedMemory::getBox`	1669	*negligible*
`BoxCoords::compare`	1551	*negligible*
`Point::getX`	876	109
`:: getNextOrigin`	704	*negligible*
`Box::getParticlesCount`	649	*negligible*
`Particle::getCenter`	570	80
`::calculateDeltaForceVectorUsingDifEcuations`	509	91
`SharedMemoryManager::getBox`	451	9
`std::vector<>::size`	370	*negligible*

perform larger simulation, i.e., involving millions of particles, in reasonable execution times. Next section reports the full experimental evaluation of the implemented methods.

Chapter 6

Experimental evaluation: occupied cells versus octal tree

This chapter describes the performance and numerical analysis of the occupied cells and the octal tree self-gravity calculation methods implemented in ESyS-Particle. The experimental evaluation was performed for the two agglomerate scenario and the collapsing cube scenario described in Chapter 5. Section 6.1 reports and discusses the performance results obtained for the simulation of different instances of the two agglomerates scenario, and Section 6.2 reports and discusses the performance results obtained for the simulation of different instances of the collapsing cube scenario. Finally, both Section 6.3 and Section 6.4 analyze the numerical accuracy of the results by studying the position of the center of mass and the conservation of the angular momentum for the two studied scenarios and the two methods evaluated.

6.1 Results for the two agglomerates scenario

This section reports the performance evaluation results for the two agglomerates scenario. The evaluation was performed for the small, medium, and large instances of the aforementioned scenario.

6.1.1 Small size instance

Table 6.1 reports the performance results for the small instance in simulations using the occupied cells method. The table reports the configuration of processes and threads used for execution, the execution time in seconds,

Table 6.1: Performance results for the two agglomerate scenario with 3,866 particles (small instance) using the occupied cells method.

#particle processes	*#gravity threads*	*execution time(s)*	*time computing self-gravity*	*# self-gravity updates*	*avg. self-gravity time(s)*
1 (1,1,1)	1	$1.06{\times}10^4$	86%	1131	8.15
1 (1,1,1)	2	$7.17{\times}10^3$	81%	1131	5.14
2 (1,1,2)	1	$1.18{\times}10^4$	87%	1131	9.06
2 (1,1,2)	2	$7.79{\times}10^3$	78%	1131	5.39

the percentage of the overall execution time spent on self-gravity calculation, the number of self-gravity updates performed, and the average time spent in self-gravity calculations.

Using the occupied cells method, the lowest execution time for the small instance was $7.17{\times}10^3$ s, using one process for ESyS-Particle and two threads for self-gravity calculation. The execution using two processes and two threads took $7.79{\times}10^3$ s to finish. When executing in the distributed mode of ESyS-Particle, a rule of thumb recommends assigning at least $5,000$ particles to each process. The rationale behind this rule is that the time spent on processes communications via MPI is high compared to the time needed to calculate the forces and particle displacements. In the small scenario, only one process is recommended according to the rule of thumb. As a consequence, using two processes is a less efficient configuration. The lowest percentage of time computing self-gravity was 78%, when using a configuration with two processes and two threads. These values indicate that the opportunities for improving the performance must be focused on the self-gravity module rather than in the ESyS-Particle core. The lowest average self-gravity time obtained was 5.14 s, using one process and two threads.

Table 6.2 reports the results of the Barnes & Hut method. The lowest execution time was $3.79{\times}10^3$ s, using a configuration of one process and two threads. Using more processes resulted in longer execution times. The aforementioned argument about the rule of thumb to define the number of processes also holds in this case.

Regarding the average self-gravity calculation time, Barnes & Hut has a lowest value of 2.30 s, which is 2.24× faster than the occupied cells method. In addition, using Barnes & Hut, the lower percentage of time computing the self-gravity was 68%, 13% lower than the occupied cells method. Comparing the

Table 6.2: Performance results for the two agglomerate scenario with 3,866 particles (small instance) using the Barnes & Hut method.

#particle processes	*#gravity threads*	*execution time(s)*	*time computing self-gravity*	*# self-gravity updates*	*avg. self-gravity time(s)*
1 (1,1,1)	1	4.09×10^3	74%	1167	2.60
1 (1,1,1)	2	3.79×10^3	72%	1167	2.32
2 (1,1,2)	1	4.55×10^3	73%	1167	2.84
2 (1,1,2)	2	3.95×10^3	68%	1167	2.30

overall execution time of the small instance, the best result using the Barnes & Hut method was 1.89× faster than the best time using the occupied cells method.

6.1.2 Medium size instance

Table 6.3 reports the performance results for the medium instance using the occupied cells method. The configuration with one process and four threads allowed obtaining the lowest execution time (1.31×10^5 s). The lower percentage of time spent on self-gravity calculation was 96%. The number of gravitational force interactions grow in a super-linear manner compared to the linear growth of the contact forces, which caused the increase in the self-gravity computation time. The best average self-gravity calculation time was 73.25 s using the configuration with one process and four threads.

Table 6.4 reports the performance results of the Barnes & Hut method. The lowest execution time was 1.31×10^5 s, using one process and four threads, and using two process and four threads. The lowest percentage of time calculating self-gravity was 74%, using one process and two threads. The lowest average time calculating self-gravity was 8.80 s, using two processes and four threads. The best average self-gravity calculation time using Barnes & Hut was 10.11× lower than using the occupied cells method.

Comparing the results of the medium and small instances using the Barnes & Hut method, the average self-gravity calculation time grew slower than when using the occupied cells method. These results show that the Barnes & Hut method implemented as part of this thesis is faster and more scalable than the occupied cells method. The efficiency of the Barnes & Huy implementation opens the opportunity to perform simulations with larger particle amounts. In

Table 6.3: Performance results for the two agglomerate scenario with 11,100 particles (medium instance) using the occupied cells method.

#particle processes	*#gravity threads*	*execution time(s)*	*time computing self-gravity*	*# self-gravity updates*	*avg. self-gravity time(s)*
1 (1,1,1)	1	$3.13{\times}10^5$	98%	1733	177.90
1 (1,1,1)	2	$1.76{\times}10^5$	97%	1733	99.18
1 (1,1,1)	4	$1.31{\times}10^5$	96%	1733	73.25
2 (1,1,2)	1	$3.12{\times}10^5$	98%	1755	175.49
2 (1,1,2)	2	$2.12{\times}10^5$	98%	1755	118.47
2 (1,1,2)	4	$1.61{\times}10^5$	97%	1755	88.99

Table 6.4: Performance results for the two agglomerate scenario with 11,100 particles (medium instance) using the Barnes & Hut method.

#particle processes	*#gravity threads*	*execution time(s)*	*time computing self-gravity*	*# self-gravity updates*	*avg. self-gravity time(s)*
1 (1,1,1)	1	$2.80{\times}10^4$	79%	1828	12.18
1 (1,1,1)	2	$2.53{\times}10^4$	74%	1828	10.27
1 (1,1,1)	4	$2.14{\times}10^4$	79%	1828	9.27
2 (1,1,2)	1	$2.43{\times}10^4$	81%	1866	10.50
2 (1,1,2)	2	$2.33{\times}10^4$	81%	1866	10.06
2 (1,1,2)	4	$2.14{\times}10^4$	77%	1866	8.80

addition, the execution time of the medium instance using the occupied cells method was one order of magnitude higher than when using Barnes & Hut.

6.1.3 Large size instance

Table 6.5 reports the performance results for the large instance using the implemented Barnes & Hut method. Executions using the occupied cells method were not performed for this instance due to the large execution times required. The lowest execution time was $4.96{\times}10^4$ s, using the configuration with four processes and four threads. The percentage of time spent on gravity calculations varied from 68% (using (1,1,1) configuration and 16 gravity threads) to 88% (using (1,2,2) configuration and 4 gravity threads), meaning that most of the execution time is spent on self-gravity calculations rather than on contact forces calculation. With regard to the average self-gravity calculation time, the lowest value was 11.01 s, using one process and eight threads. Despite

Table 6.5: Performance results of the Barnes & Hut method for the two-agglomerate scenario with 38,538 particles (large instance).

#particle processes	*#gravity threads*	*execution time(s)*	*time computing self-gravity*	*# self-gravity updates*	*avg. self-gravity time(s)*
1 (1,1,1)	1	$8.58{\times}10^4$	71%	3813	16.06
1 (1,1,1)	2	$7.45{\times}10^4$	76%	3813	12.22
1 (1,1,1)	4	$6.00{\times}10^4$	74%	3813	11.67
1 (1,1,1)	8	$5.48{\times}10^4$	77%	3813	11.01
1 (1,1,1)	16	$8.32{\times}10^4$	68%	3815	14.84
2 (1,1,2)	1	$7.74{\times}10^4$	77%	3807	15.65
2 (1,1,2)	2	$6.34{\times}10^4$	69%	3807	11.48
2 (1,1,2)	4	$5.96{\times}10^4$	71%	3813	11.13
2 (1,1,2)	8	$6.52{\times}10^4$	76%	3807	13.09
2 (1,1,2)	16	$7.36{\times}10^4$	76%	3807	14.69
4 (1,2,2)	1	$7.32{\times}10^4$	83%	3781	16.08
4 (1,2,2)	2	$6.31{\times}10^4$	75%	3781	12.43
4 (1,2,2)	4	$4.96{\times}10^4$	88%	3781	11.50
4 (1,2,2)	8	$6.14{\times}10^4$	84%	3781	13.67
4 (1,2,2)	16	$6.77{\times}10^4$	83%	3781	14.84
8 (2,2,2)	1	$7.43{\times}10^4$	83%	3781	16.26
8 (2,2,2)	2	$7.14{\times}10^4$	76%	3781	14.35
8 (2,2,2)	4	$6.35{\times}10^4$	84%	3781	14.03
8 (2,2,2)	8	$6.77{\times}10^4$	80%	3781	14.37
8 (2,2,2)	16	$6.35{\times}10^4$	82%	3781	13.79

that the large instance has almost four times more particles than the medium instance, the average self-gravity calculation time is only 1.26× longer. These results indicate that the growth in the average self-gravity calculation time is sub-linear when using the Barnes & Hut method. The reason is that the method is based in cells rather than the individual particles. This way, if a scenario grows in number of particles but the cell size stays unchanged, the increment in execution time will be sub-linear. Anyway, if more precision is needed, smaller cells should be used, hence resulting in larger execution times.

6.1.4 Overall discussion for the two-agglomerate scenario

For the execution using the Barnes & Hut method, in the small instance the self-gravity was updated 1167 times, whereas on the medium instance it was updated from 1828 to 1866 times, and on the large instance it was updated between 3781 and 3813 times. So, the number of gravity updates increases when the size of the particles decreases.

In general, using the Barnes & Hut method to simulate the two agglomerates scenario lowered the percentage of time calculating the self-gravity in up to 23%. This effect is caused by the up to 10× lower average self-gravity calculation time of the Barnes & Hut method in comparison with the occupied cells method. In addition, the overall execution time of the scenario of the two agglomerates using the Barnes & Hut method was affected by the 10× acceleration of the average self-gravity calculation time. Using the Barnes & Hut method, the execution time reported is one order of magnitude lower than the occupied cells method. This is a very relevant performance result that allows the execution of instances with a larger number of particles, and also for a larger number of time steps.

Results show a sub linear increase in the performance of the proposed implementation of the Barnes & Hut method when increasing the number of computational resources. The reason for this behavior is twofold: i) the average self-gravity calculation time is usually too short to exploit the benefits of a parallel environment, ii) the process of creation and deletion of the Barnes & Hut tree is not implemented in parallel, so most of the time updating the self-gravity is spent performing sequential operations. This last issue of the Barnes & Hut based method is proposed be addressed in future work.

6.2 Collapsing cube scenario

This section reports and analyzes the performance results of both the occupied cells and the Barnes & Hut methods for the defined instances of the collapsing cube scenario.

6.2.1 Small instance

Table 6.6 reports the experimental results of the execution of the occupied cells method for the small cube instance. The number of self-gravity updates was 11 for all the tested configurations of processes and threads. The lowest execution time was 1.50×10^4 s using four processes and eight threads. The lowest average self-gravity calculation time (1238.24 s) was also achieved with the same configuration of processes and threads. However, the lowest percentage

Table 6.6: Performance results for the collapsing cube scenario with 32,768 particles (small instance) executed with the occupied cells implementation.

#particle processes	*#gravity threads*	*execution time(s)*	*time computing self-gravity*	*# self-gravity updates*	*avg. self-gravity time(s)*
1 (1,1,1)	1	$3.26{\times}10^4$	82.79%	11	2451.75
1 (1,1,1)	2	$2.90{\times}10^4$	77.60%	11	2046.41
1 (1,1,1)	4	$2.45{\times}10^4$	71.51%	11	1591.73
1 (1,1,1)	8	$2.10{\times}10^4$	65.10%	11	1241.98
2 (1,1,2)	2	$2.24{\times}10^4$	84.86%	11	1725.69
2 (1,1,2)	4	$2.22{\times}10^4$	84.77%	11	1711.92
2 (1,1,2)	8	$1.76{\times}10^4$	80.02%	11	1279.81
4 (1,2,2)	1	$3.34{\times}10^4$	98.65%	11	2998.07
4 (1,2,2)	2	$2.41{\times}10^4$	91.42%	11	2001.9
4 (1,2,2)	4	$2.06{\times}10^4$	81.48%	11	1524.34
4 (1,2,2)	8	$1.50{\times}10^4$	91.10%	11	1238.24

of time computing self-gravity (71%) was obtained using one process and four threads.

In turn, Table 6.7 reports the results of the Barnes & Hut method for the small instance of the collapsing cube scenario. The number of self-gravity updates was 13 for all the configurations. Results indicate that the variation in the self-gravity computation time when increasing the number of self-gravity threads is of about 50% being the lowest value reported 26.14 s, using one process and four threads, and the highest value 39.28 s, using four processes and one thread. However, the lowest execution time measured was $2.25{\times}10^3$ s using eight processes and eight threads. The lowest percentage of time calculating the self-gravity was 4.51% using one process and two threads. Most of the simulation time was spent on the calculation of the contact forces, varying from 81.9% of the time to 96.5% of the time.

The percentage of time calculating self-gravity during the simulations was analyzed. Results presented on Table 6.6 report that using the selected implementation of the occupied cells method, most of the execution time was spent calculating self-gravity, with values varying from 65% to 98% of the total execution time of a simulation. On the other hand, for the Barnes & Hut method the percentage of time spent calculating self-gravity varied from 4% to 19%, as it is shown in Table 6.7. In addition, a comparison was performed regarding the average self-gravity calculation time of the occupied cells and the Barnes & Hut method. The lowest value of the average self-gravity calculation time

Table 6.7: Performance results for the collapsing cube scenario with 32,768 particles (small instance) executed with the Barnes & Hut method implementation.

#particle processes	*#gravity threads*	*execution time(s)*	*time computing self-gravity*	*# self-gravity updates*	*avg. self-gravity time(s)*
1 (1,1,1)	1	$8.89{\times}10^3$	4.78%	13	32.65
1 (1,1,1)	2	$8.79{\times}10^3$	4.51%	13	30.51
1 (1,1,1)	4	$7.52{\times}10^3$	4.52%	13	26.14
1 (1,1,1)	8	$7.64{\times}10^3$	4.59%	13	26.99
2 (1,1,2)	1	$4.24{\times}10^3$	8.49%	13	27.68
2 (1,1,2)	2	$7.41{\times}10^3$	5.60%	13	31.90
2 (1,1,2)	4	$7.83{\times}10^3$	5.35%	13	32.21
2 (1,1,2)	8	$6.41{\times}10^3$	6.34%	13	31.24
4 (1,2,2)	1	$5.10{\times}10^3$	10.01%	13	39.28
4 (1,2,2)	2	$3.62{\times}10^3$	11.47%	13	31.90
4 (1,2,2)	4	$3.30{\times}10^3$	12.99%	13	32.93
4 (1,2,2)	8	$3.85{\times}10^3$	11.21%	13	33.18
8 (2,2,2)	1	$2.35{\times}10^3$	19.11%	13	34.54
8 (2,2,2)	2	$2.37{\times}10^3$	18.52%	13	33.80
8 (2,2,2)	4	$2.39{\times}10^3$	18.31%	13	33.69
8 (2,2,2)	8	$2.25{\times}10^3$	16.58%	13	28.65

for the occupied cells method was 1238.24 s, which is two orders of magnitude higher than the lowest value of 26.14 s obtained using the Barnes & Hut method. The self-gravity is calculated **up to 47.37×** faster using the Barnes & Hut method rather than using the occupied cells method. The reduction of the calculation time of the self-gravity allows the simulation of larger and more realistic scenarios.

6.2.2 Medium instance

Table 6.8 reports the experimental results of the Barnes & Hut method for the medium instance. The number of self-gravity updates was 21 for all the configurations used. The lowest average self-gravity calculation time was 251.06 s, using eight processes and four threads. According to the results, the lowest percentage of time computing self-gravity was 7.69%, using one process and two threads. The lowest execution time obtained was $1.84{\times}10^4$ s using 27 processes and 16 threads. Partial simulations were performed for the occupied cells method on the medium size instance, for the reasons explained below. Results showed that the time needed to perform a single self-gravity update

Table 6.8: Performance results for the collapsing cube scenario with 262,144 particles (medium instance) executed with the Barnes & Hut method implementation.

#particle processes	*#gravity threads*	*execution time(s)*	*time computing self-gravity*	*# self-gravity updates*	*avg. self-gravity time(s)*
1 (1,1,1)	1	8.01×10^4	8.24%	21	314.69
1 (1,1,1)	2	7.38×10^4	7.69%	21	270.38
1 (1,1,1)	4	1.04×10^5	11.66%	21	575.64
1 (1,1,1)	8	7.89×10^4	9.71%	21	364.81
1 (1,1,1)	16	6.79×10^4	10.41%	21	336.64
8 (2,2,2)	2	3.56×10^4	20.78%	21	352.40
8 (2,2,2)	4	1.85×10^4	28.50%	21	251.06
8 (2,2,2)	8	2.80×10^4	35.42%	21	472.47
8 (2,2,2)	16	2.50×10^4	27.53%	21	327.87
27 (3,3,3)	16	1.84×10^4	36.34%	21	320.09

was 2.08×10^5 s. This self-gravity calculation time is three orders of magnitude larger than the result reported for the average self-gravity calculation time using the Barnes & Hut method. Due to the large execution time required to perform a complete simulation, the complete performance study for the occupied cells method was not performed in the context of this research.

6.2.3 Large instance

Table 6.9 reports the results of the Barnes & Hut method for the large instance. Executions using one process and one thread, and using the occupied cells method were not performed due to the large execution times required: the estimated execution time using a configuration of one process and one thread is 1.5×10^6 s, while the estimated execution time using the occupied cells method is 9.0×10^7 s. The large instance simulation performed 41 or 42 self-gravity updates, depending on the configuration of processes and threads used. The percentage of time of the simulation spent updating self-gravity varied from 6.60% to 20.21%. In addition, the lowest execution time was 1.25×10^5 s using 36 processes and 16 threads. The lowest average self-gravity calculation time was 313.14 s.

6.2.4 Overall discussion for the collapsing cube scenario

A comparative analysis of the results after the execution of the simulations using the Barnes & Hut method for the large instance indicates that the exe-

Table 6.9: Performance results for the collapsing cube scenario with 2,097,152 particles (large instance) executed with the Barnes & Hut method implementation.

#particle processes	*#gravity threads*	*execution time(s)*	*time computing self-gravity*	*# self-gravity updates*	*avg. self-gravity time(s)*
8 (2,2,2)	8	$1.43{\times}10^5$	20.21%	42	689.01
8 (2,2,2)	16	$1.95{\times}10^5$	6.60%	41	313.14
27 (3,3,3)	8	$1.57{\times}10^5$	8.65%	41	331.41
36 (3,3,4)	16	$1.25{\times}10^5$	11.09%	41	336.72

cution time grows one order of magnitude with respect to the medium instance. However, the average self-gravity calculation time registered for the medium and large instances is of the same order of magnitude. The self-gravity calculation time did not vary significantly because the Barnes & Hut method performance is not bounded to the number of particles but to the number of boxes. In the medium and large instances, the same box size was used to perform the calculations. This means that, if the number of boxes is fixed, increasing the number of particles of a scenario does not affect the self-gravity calculation time.

Results show that the acceleration for the simulations of the cube scenario using the Barnes & Hut method was more than **50**× for the small instance, and was more than **100**× for the medium instance compared to the occupied cells method. In addition, the average self-gravity calculation time was of the same order of magnitude for the medium and large instance of the cube scenario using the Barnes & Hut method. Thus, performance improvements of up to **100**× are also expected in this case. These performance improvements allow scaling up to perform realistic simulations with a large number of particles (tens of millions) in reasonable execution times, which constitutes the main algorithmic contribution of this thesis.

6.3 Numerical accuracy: analysis of the position of the center of mass

This section studies the numerical accuracy of the results obtained in the simulations performed using the occupied cells and the Barnes & Hut methods.

6.3.1 Description of the studies

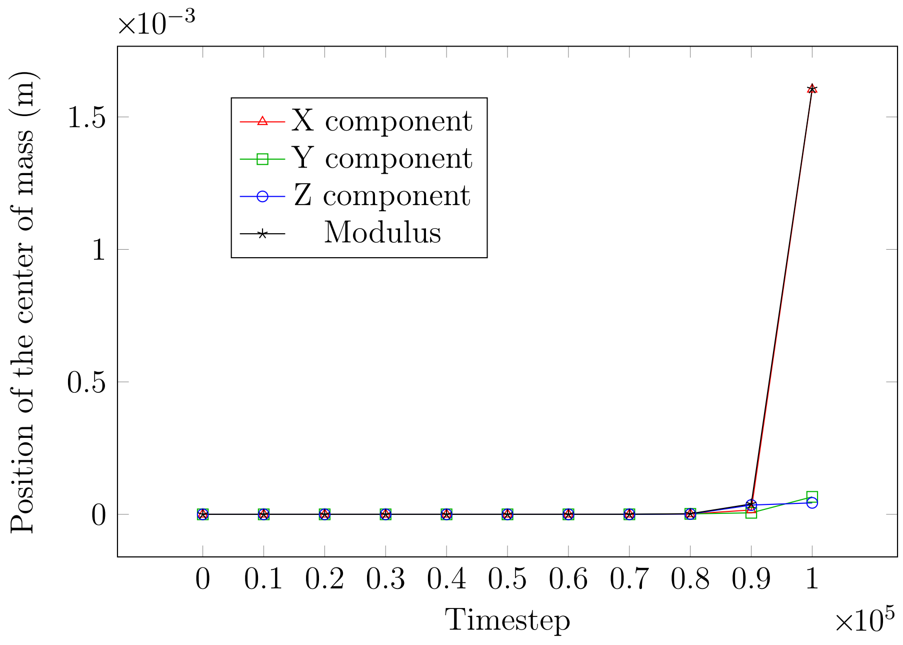
The analysis is organized in two parts: the study of the position of the center of mass for the two agglomerates scenario and for the collapsing cube scenario. Two analysis are presented for each scenario. One is the comparison of the position of the center of mass for the small size instance using both the occupied cells method and the Barnes & Hut method. The second is an analysis of the scalability of the Barnes & Hut method by performing a comparison and discussion of the movement of the position of the center of mass for the small, medium, and large instances of both scenarios. Both studied scenarios were configured to be symmetrical, so the center of mass of all particles stays in the same position for the complete simulation. The less the center of mass moves during the simulation, the more accurate to the reality represented. In an ideal scenario where the initial velocity of the center of mass is zero, and due to the symmetrical characteristics, the total movement of the center of mass should be zero during the complete simulation. The analysis of the variation of the position of the center of mass is performed in order to determine the numerical accuracy of the proposed methods. This analysis allows verifying that the approximations in the proposed methods do not introduce significant errors on the particles movement, thus allowing performing accurate simulations of the systems studied.

Regarding the figures presented in this subsection, four sets of data are reported in each figure: red, green, and blue lines represent the evolution of the X, Y, and Z components of the position of the center of mass for a simulation, respectively. The variation of the modulus of the position of the center of mass for the same simulation is represented in black. Values are reported for different time step values for a representative simulation.

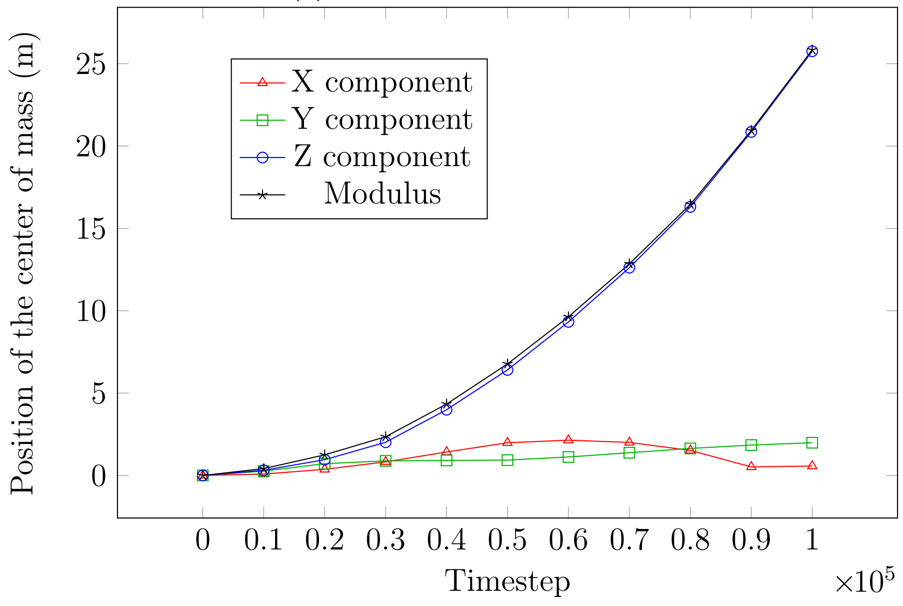
6.3.2 Results for the two agglomerate scenario

Figure 6.1 shows the variation of the position of the center of mass for the small instance of the two agglomerates scenario. Results for the occupied cells method are presented in Figure 6.1a and results for the Barnes & Hut method are presented in Figure 6.1b.

Results show that the position of the center of mass varies in the order of $1{\times}10^{-3}$ m when using the occupied cells method and in the order of $1{\times}10^{1}$ m when using the Barnes & Hut method. So, a loss of precision of the calculation



(a) Occupied cells method.



(b) Barnes & Hut method.

Figure 6.1: Position of the center of mass over time for the small instance of the two agglomerates scenario.

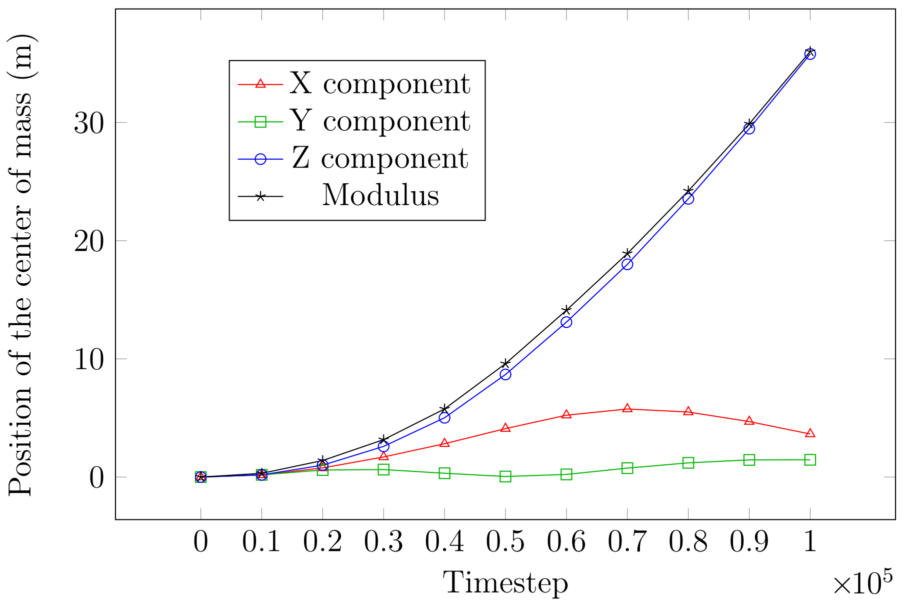
of the self-gravity is perceived when the Barnes & Hut method is used with a neighborhood of size 3. The cause of the loss of precision lies in the strategy implemented to traverse the octal tree and to create the list of tree nodes from which the self-gravity of a node is calculated. The strategy consists of creating groups of tree nodes, that do not contain the node to be calculated, into bigger nodes. The action of grouping the nodes into bigger nodes causes a loss of precision in the calculation. In addition, bigger nodes cause a larger loss of precision in the calculation of the self-gravity. In systems where the effect of the self-gravity of the particles has significance, rounding the calculation of the acceleration produced by grouping the particles into a single major particle introduces changes in the net force applied to the particles.

Figure 6.2 shows the position of the center of mass for the medium (Figure 6.2a) and large (Figure 6.2b) scenarios using the Barnes & Hut method. Results show that the position of the center of mass in both scenarios varies in the same order of magnitude as in the small scenario. The X and Y components vary between zero and five meters, while the Z component moves up to 36.01 m in the medium scenario and up to 40.26 m in the large scenario at the time step $1{\times}10^5$. Recall that the two agglomerates are separated by $5,000\,m$. So, movement the Z component for the medium scenario (36.01 m) represents 0.72% of the distance between the agglomerates, while the movement for the large scenario (40.26 m) represents 0.81% of distance between the agglomerates. The less the center of mass moves during the simulation, the more accurate to the reality represented is the simulation. In an ideal scenario where the initial velocity of the center of mass is zero, and due to the symmetrical characteristics, the total movement of the center of mass should be zero during the complete simulation. The two agglomerates move along the plain that contains the X and Y axis. So, the center of mass moves mainly over the Z component by less than 1% of the distance between the agglomerates for all the scenarios studied.

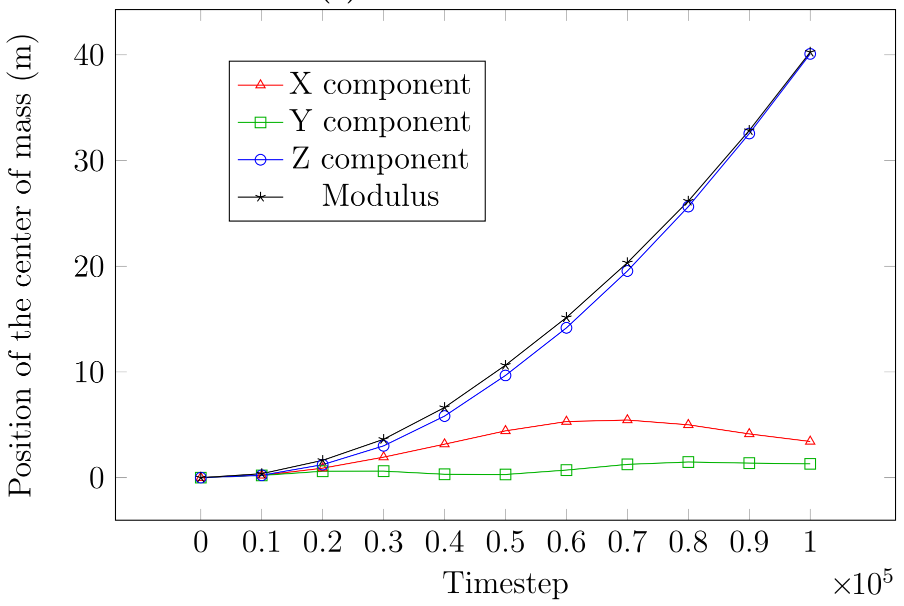
6.3.3 Results for the collapsing cube scenario

Figure 6.3 shows the variation of the position over time of the center of mass for the small cube scenario. Figure 6.3a shows the results for the occupied cells method. Figure 6.3b shows the results for the Barnes & Hut method. Results indicate that using either algorithm, the modulus of the center of mass increases over time.

For the small instance, the position of the center of mass varies in the order of $1{\times}10^{-4}$ m in the case of the occupied cells method, and in the order of $1{\times}10^{-2}$ m for the Barnes & Hut method. The values obtained confirm that the occupied cells method is more precise than the Barnes & Hut method. How-



(a) Medium scenario.



(b) Large scenario.

Figure 6.2: Position of the center of mass over time for the medium and large instances of the two agglomerates scenario using the Barnes & Hut method.

ever, the precision of the algorithm is higher for the collapsing cube scenario rather than for the two agglomerates scenario. These results suggest that the Barnes & Hut algorithm performs best in scenarios with slow movements of the particles.

Figure 6.4 shows the position of the center of mass for the medium (Figure 6.4a) and large (Figure 6.4b) scenarios using the Barnes & Hut method. For the medium scenario, the maximum value for the position of the center of mass is 36.0 m and for the large scenario is 40.3 m. Both values were registered at the $1{\times}10^5$ time step. Results indicate that the variation of the position of the center of mass for the medium and large instances is of the same order of magnitude to the one obtained for the small instance. Thus, the reported results show that scaling the number of particles does not affect the numerical accuracy of the results obtained when a simulation is performed. The reason of this behavior is that the loss in precision is introduced mainly by the calculation of the self-gravity, which is calculated based on the nodes and boxes rather than on the particles.

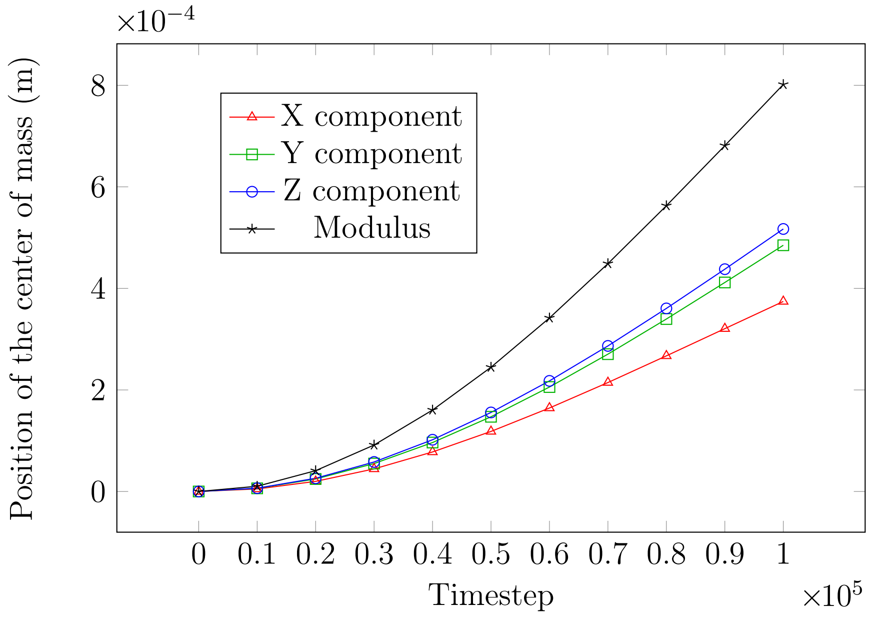
6.4 Numerical accuracy: analysis of the angular momentum

This section studies the conservation of the angular momentum measured for both scenarios and their small, medium, and large instances using a neighborhood of size three. The results obtained are reported, commented, and discussed in the following subsections.

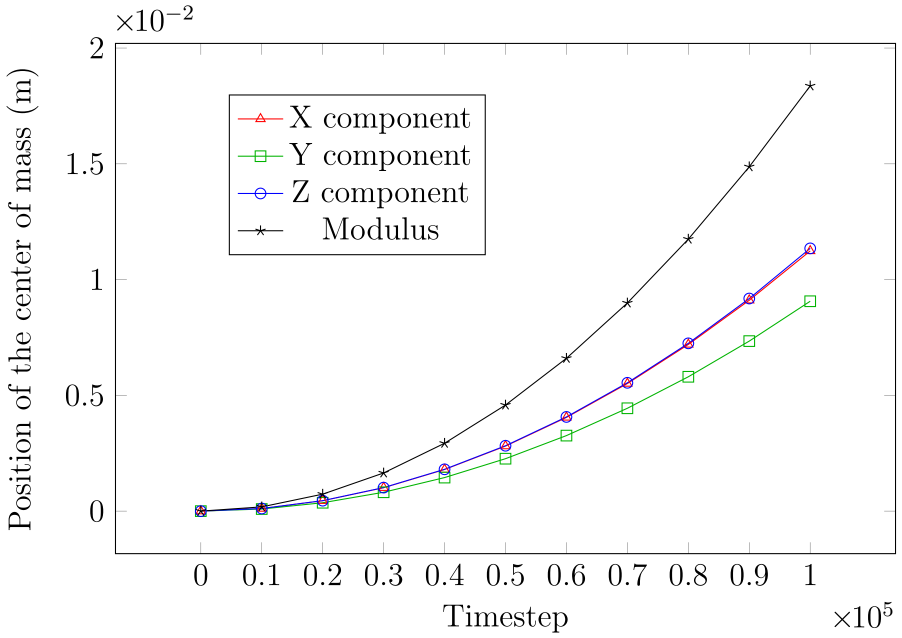
6.4.1 Angular momentum and its relevance

The angular momentum is used to calculate and describe the rotational momentum of a particle or group of particles. The importance of angular momentum resides in the fact that it has the property that it is conserved with respect to a point and in this experiment it is not affected by external forces. The angular momentum of a point particle with respect to the origin is described by Equation 6.1, where $\vec{r_o}$ represents the distance of the particle from the origin, m represents the mass of the particle, and $\vec{v}$ is the velocity of the particle. The angular momentum is measured in $kg\frac{m^2}{s}$.

$$L_o = \vec{r_o} \times m\vec{v}, \tag{6.1}$$



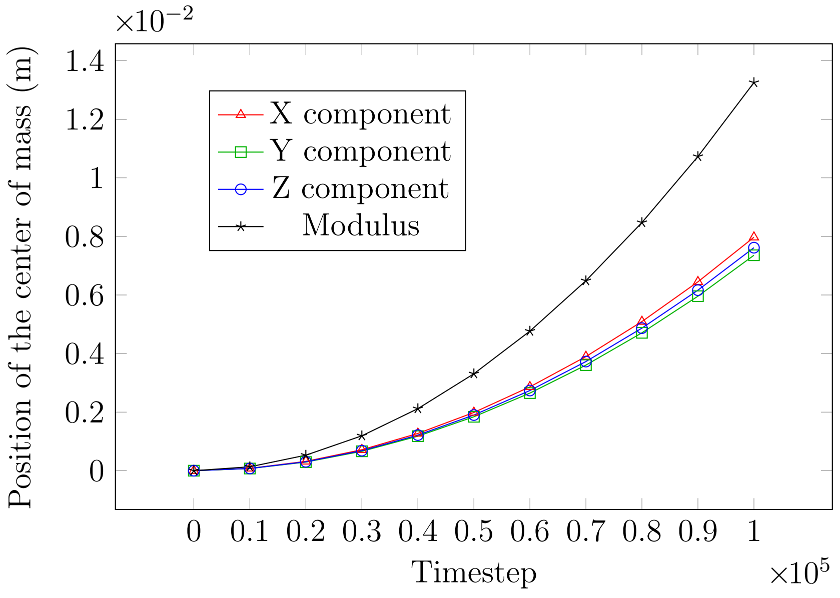
(a) Occupied cells method.



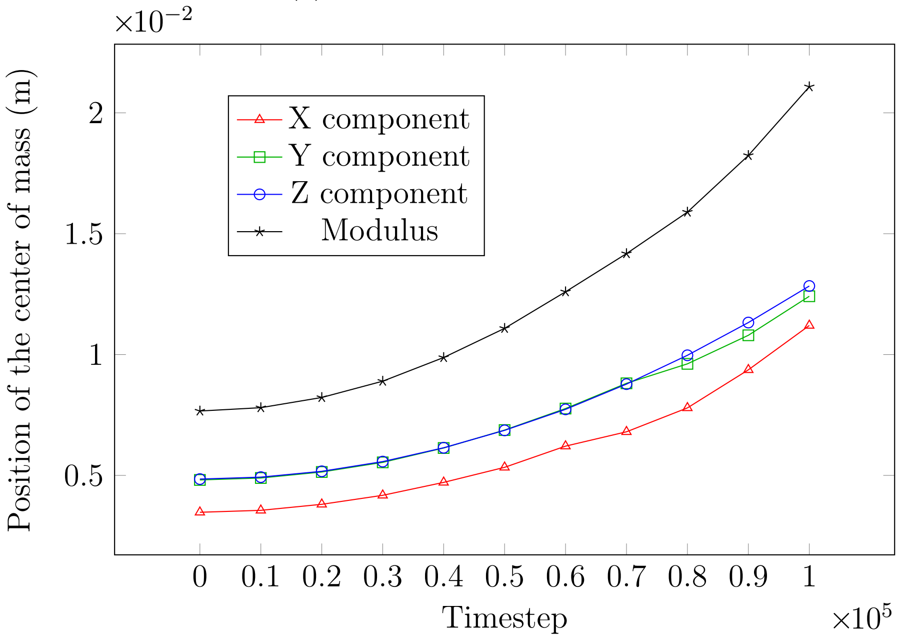
(b) Barnes & Hut method.

Figure 6.3: Position of the center of mass over time for the small instance of the cube scenario.

The aim of the study of the angular momentum is to measure its conservation throughout the simulations. To that extent, the angular momentum is calculated based on the results obtained from the simulations performed. Then, the modulus of the angular momentum vector is calculated, and the variation is studied and commented.



(a) Medium cube scenario.

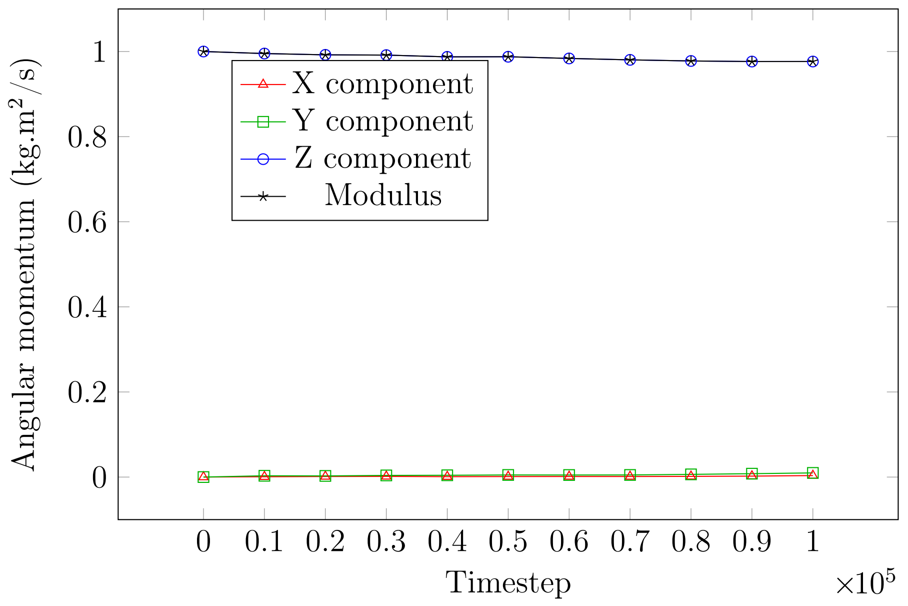


(b) Large cube scenario.

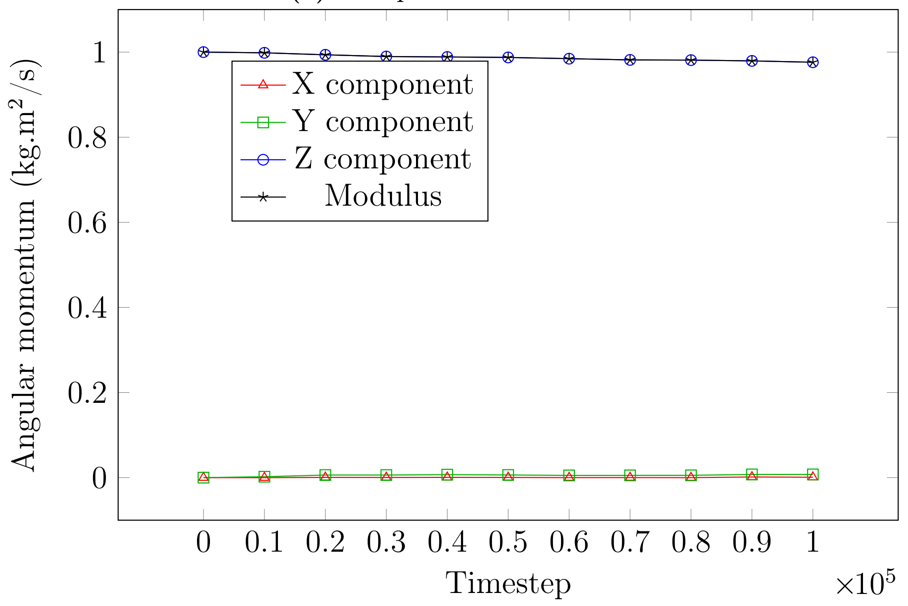
Figure 6.4: Position of the center of mass over time for the medium and large instances of the cube scenarios using the Barnes & Hut method.

6.4.2 Results for the two agglomerates scenario

Figure 6.5 shows the shows the variation of the angular momentum over time for the small instance of the two agglomerates scenario. Figure 6.5a shows the angular momentum variation using the occupied cells algorithm, while Figure 6.5b shows the variation using the Barnes & Hut octal tree algorithm.



(a) Occupied cells method.



(b) Barnes & Hut method.

Figure 6.5: Angular momentum over time for the small instance of the two agglomerates scenario.

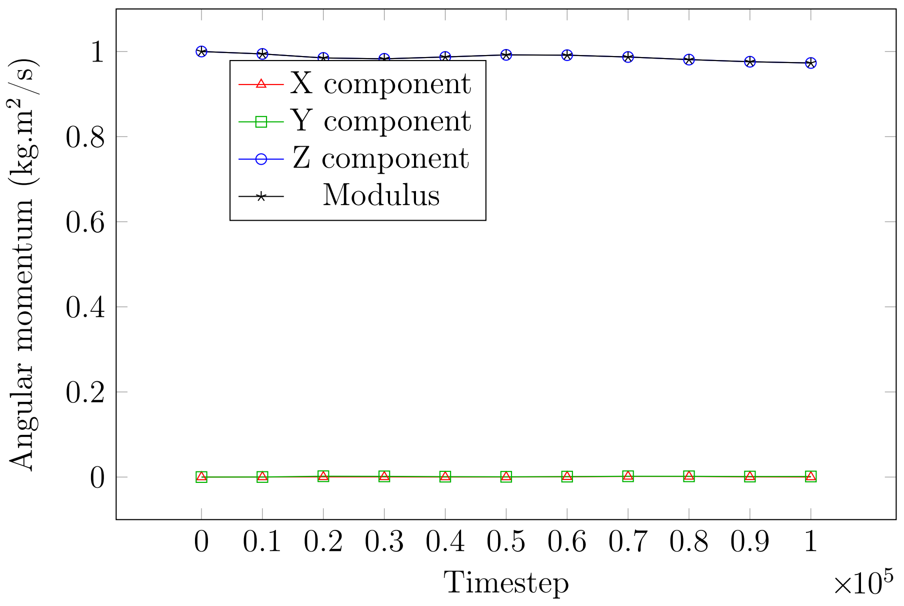
Results for the two agglomerates scenario indicate that the values for the X axis, Y axis, Z axis, and modulus using the occupied cells method are similar to the values using the Barnes & Hut method. Comparing the results obtained, the value for the angular momentum is 6.13×10^{18} kg.m^2/s using either method for the last time step of the simulation, which is time step 1×10^5. So, the obtained results suggest that the angular momentum for the small instance is conserved with the same precision for the occupied cells method and the Barnes & Hut method.

Figure 6.6 shows the variation of the angular momentum over time for the two agglomerates scenario using the Barnes & Hut method. Figure 6.6a shows the results obtained for the medium instance, while Figure 6.6b shows the results for the large instance. The initial value for the modulus of the angular momentum is $6.20{\times}10^{18}$ kg.m^2/s and the value at the end of the simulation is $6.03{\times}10^{18}$ kg.m^2/s. The difference between the initial value and the value at the last time step is of $0.17{\times}10^{18}$ kg.m^2/s, which is similar to the difference for the same interval for the small scenario using both methods ($0.14{\times}10^{18}$ kg.m^2/s).

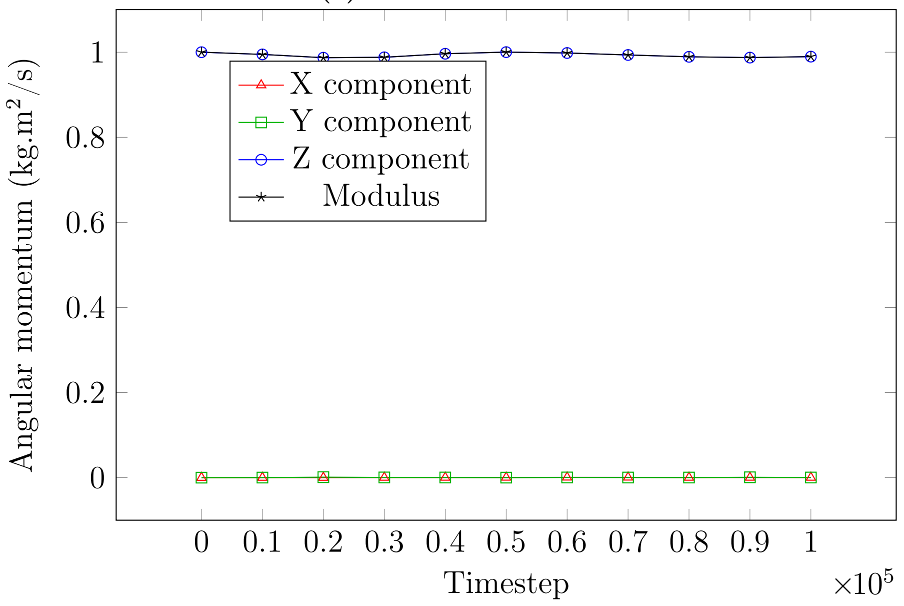
In the case of the execution of the large instance using the Barnes & Hut method, the results obtained were similar to the ones obtained for the medium instance. In this case, the values of the angular momentum modulus oscillated between $8.42{\times}10^{18}$ kg.m^2/s and $8.33{\times}10^{18}$ kg.m^2/s, therefore in this case the difference of the angular momentum modulus is $0.09{\times}10^{18}$ kg.m^2/s. The results presented for the medium and large instances suggest that for the two orbiting agglomerates the angular momentum varies in the same order of magnitude for the Barnes & Hut method and the occupied cells method.

6.4.3 Results for the collapsing cube scenario

Figure 6.7 shows the variation of the angular momentum for the execution of the small instance of the collapsing cube scenario. The results using the occupied cells method are presented in figure 6.7a, while the results for the Barnes & Hut method are presented in figure 6.7. In this scenario, the modulus of the angular momentum should ideally remain with a value of 0 kg.m^2/s for the complete simulation. Given that both the occupied cells and the Barnes & Hut are approximated methods, the results deviate from the ideal values. The modulus of the angular momentum increases with the succession of the time steps. The maximum value obtained for the small instance using the occupied cells method is $1.48{\times}10^{6}$ kg.m^2/s , while the maximum value using the Barnes & Hut method is $1.97{\times}10^{8}$ kg.m^2/s. Results show that in this case the modulus for the Barnes & Hut method is two orders of magnitude larger than for the occupied cells method. In addition, the behavior of the increase of the modulus is different for both methods: for the occupied cells method the speed of the increase slows down as the time steps increase, while the results for



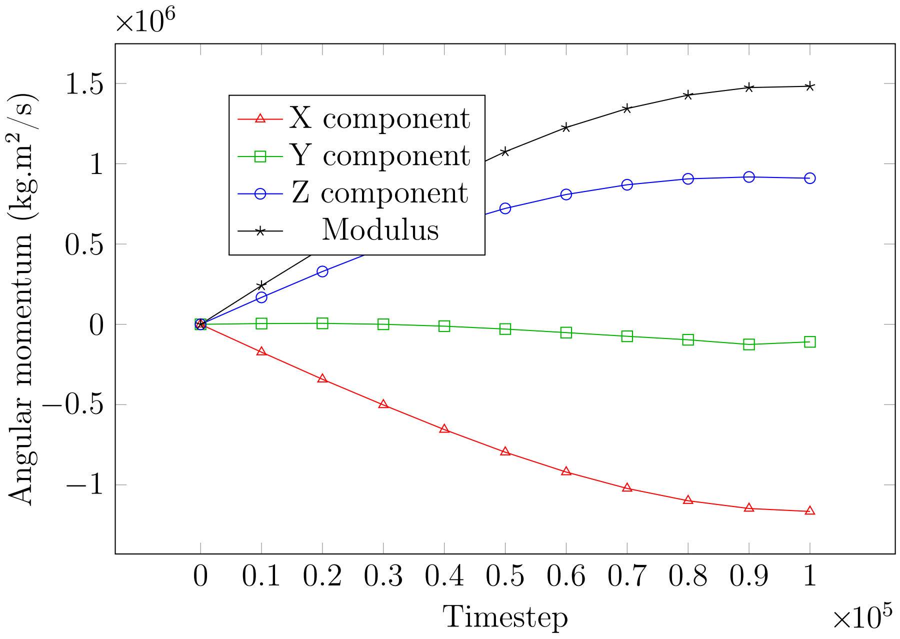
(a) Medium scenario.



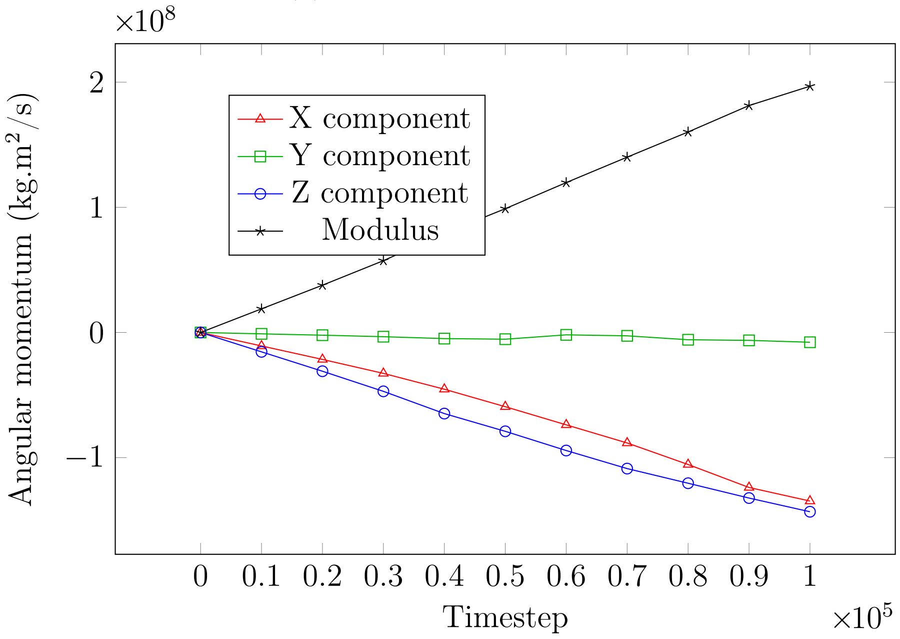
(b) Large scenario.

Figure 6.6: Angular momentum over time for the medium and large instances of the two agglomerates scenario using the Barnes & Hut method.

the Barnes & Hut method show an steady increase of the modulus throughout the simulation. In spite of the difference in precision, the acceleration using the Barnes & Hut method is calculated 47.37× faster than using the occupied cells method (see section 6.2.1).



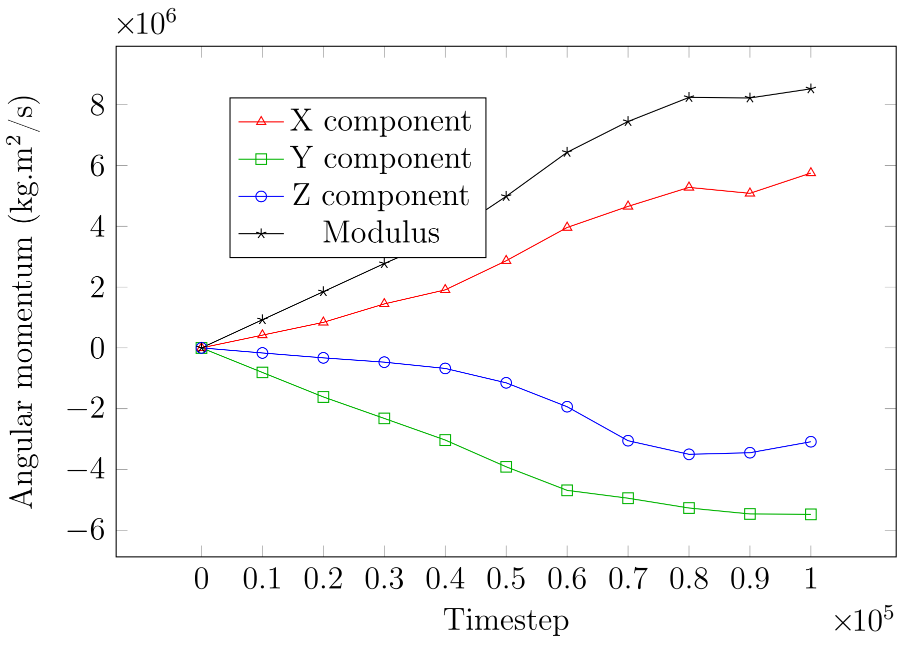
(a) Occupied cells method.



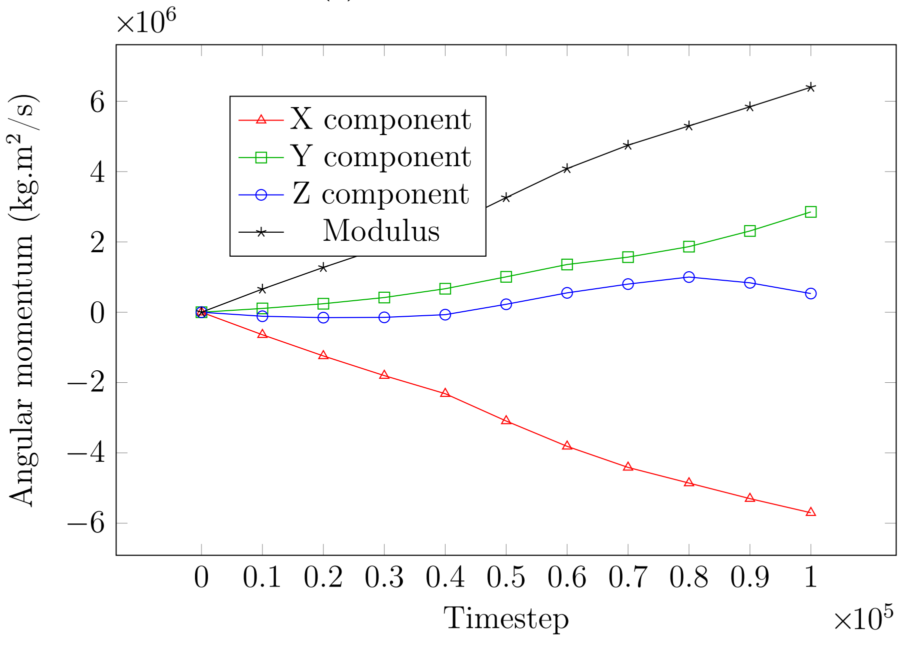
(b) Barnes & Hut method.

Figure 6.7: Angular momentum over time for the small instance of the collapsing cube scenario.

Figure 6.8 shows the results obtained for the study of the evolution of the angular momentum over time for the medium and large instances of the collapsing cube using the Barnes & Hut method. The results presented in Figure 6.8a show that the modulus of the angular momentum also increases



(a) Medium scenario.



(b) Large scenario.

Figure 6.8: Angular momentum over time for the medium and large instances of the collapsing scenario using the Barnes & Hut method.

for this instance, even though the derivative decreases for the last three time steps measured. The larger value for the modulus of the angular momentum for the medium scenario is $8.52{\times}10^6$ kg.m^2/s.

Figure 6.8b shows the results of the study of the angular momentum performed to the large instance of the collapsing cube scenario using the Barnes & Hut method. For this instance, the large value for the modulus of the angular momentum is $6.40{\times}10^6$ kg.m^2/s; this value is of the same order of magnitude as the larger value obtained for the small instance using the occupied cells method. Also, the values of the modulus of the angular momentum increase over time. This increase follows the same pattern that was previously commented for the small and medium instances and for both the occupied cells and the Barnes & Hut method.

Overall, the results obtained for the study of the angular momentum performed to the collapsing cube suggest that as the number of particles in the instance increases, the numerical results for the modulus of the angular momentum using the Barnes & Hut method approximate to the values obtained using the occupied cells method. In addition, results for all the instances tested showed that the modulus of the angular momentum increased over time. This last result show that the angular momentum behaved differently for the collapsing cube compared to the two agglomerates scenario, in which the variation of the modulus of the angular momentum was such that all the values were of the same order of magnitude.

Chapter 7

Experimental evaluation: binary tree algorithm and upper tree level mass center calculation

This chapter reports the results of executions performed with the two agglomerates scenario using the binary tree algorithm and a comparison of performance against the Barnes & Hut octal tree. The results include the numerical accuracy of the performance of the compared methods. In addition, a study of the performance of a variation of the octal tree is presented, in which the mass center is calculated using the particles of the system at upper tree levels rather than center of mass of the lower levels.

7.1 Performance results of the binary tree algorithm

The performance of the binary tree algorithm was studied by means of a comparison against the octal tree algorithm. Results are reported for the executions of the three instances defined in Section 5.1, using different configurations of processes and threads. The processes are related to the workload distribution of the contact forces calculation, while the threads are related to the self-gravity update process. All results correspond to the average of five executions for each configuration.

Table 7.1 reports the total execution time and the average time of a self-gravity update when using the octal tree and the binary tree algorithms for the small instance of the two agglomerates scenario.

Table 7.1: Performance results for the two agglomerate scenario with 3,866 particles (small instance).

		octal tree		*binary tree*	
#particle processes	*#gravity threads*	*execution time(s)*	*avg. self-gravity time(s)*	*execution time(s)*	*avg. self-gravity time(s)*
1 (1,1,1)	1	$9.15{\times}10^2$	10.11	$1.30{\times}10^3$	14.55
1 (1,1,1)	2	$6.64{\times}10^2$	6.99	$1.03{\times}10^3$	11.65
2 (1,1,2)	1	$9.59{\times}10^2$	10.58	$1.76{\times}10^3$	18.78
2 (1,1,2)	2	$7.08{\times}10^2$	7.40	$1.47{\times}10^3$	15.18

For the small instance, experiments were executed for up to two processes and two threads, taking into account the rule-of-thumb that recommends assigning at least $5,000$ particles to each process on the distributed mode of ESyS-Particle. When using either tree algorithm, self-gravity was updated 82 times.

For self-gravity update, results show that the octal tree algorithm is up to 2×faster than the binary tree algorithm. Results confirm the rule-of-thumb, since the lowest execution time was obtained using one process and one thread. When increasing the number of gravity threads from 1 to 2, the small instance ran approximately in 30% less time for the octal tree, whereas in the case of the binary tree the instance finished the execution in approximately 25% less time. Thus, the small instance ran faster using the octal tree algorithm than using the binary tree.

For the medium instance, the evaluation was performed for six configurations of gravity processes and gravity threads. When using either tree algorithm, the self-gravity was updated for a total of 127 times. Table 7.2 reports the results obtained for the execution of the medium instance of the two agglomerate scenario when using the octal tree and the binary tree algorithm. The lowest execution time was achieved using the octal tree algorithm with a configuration of two processes and four threads, which supports the rule of thumb. For the medium instance, the best binary tree execution time was approximately 20% slower than the best octal tree time.

Table 7.2: Performance results for the two agglomerate scenario with 11,100 particles (medium instance).

		octal tree		*binary tree*	
#particle processes	*#gravity threads*	*execution time(s)*	*avg. self-gravity time(s)*	*execution time(s)*	*avg. self-gravity time(s)*
1 (1,1,1)	1	$6.87{\times}10^3$	51.37	$8.02{\times}10^3$	59.55
1 (1,1,1)	2	$4.75{\times}10^3$	31.22	$6.32{\times}10^3$	46.41
1 (1,1,1)	4	$4.30{\times}10^3$	31.09	$5.27{\times}10^3$	38.19
2 (1,1,2)	1	$7.14{\times}10^3$	54.23	$9.76{\times}10^3$	72.70
2 (1,1,2)	2	$4.57{\times}10^3$	33.85	$7.31{\times}10^3$	53.70
2 (1,1,2)	4	$4.10{\times}10^3$	30.35	$5.70{\times}10^3$	41.26

The large instance was studied considering experiments performed with 20 different configurations of processes and threads. In the tests performed for the large instance, the gravitational potential was updated 264 times for both algorithms. Table 7.3 reports the results obtained for each of the studied configurations.

For the large instance, the best execution time was obtained using the binary tree with the configuration of eight processes and four threads. This result supports the rule of thumb. Also, for configurations with the same number of processes, the configurations using eight or 16 threads performed slower than the configurations using four threads. Results obtained suggest that the binary tree algorithm performs faster than the octal tree for large instances. This is a relevant result from the research reported in this document, as using a binary tree has not been previously proposed and is a direct contribution of this thesis.

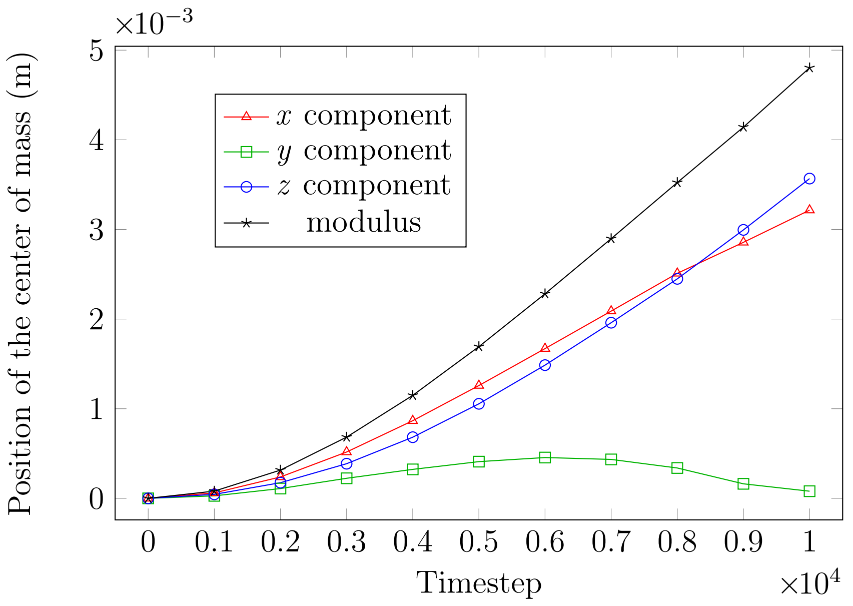
7.2 Numerical accuracy of the binary tree algorithm: analysis of the center of mass

Due to the symmetrical characteristics of the scenario, the center of mass of the system should remain static at the center of the space as the agglomerates move. However, the simulation calculates the interactions of the particles in discrete steps, which introduces error in the calculations. So, a study of the numerical accuracy was performed analyzing of the position of the center of mass during time for the three instances considered in the experimental analysis of the binary tree algorithm.

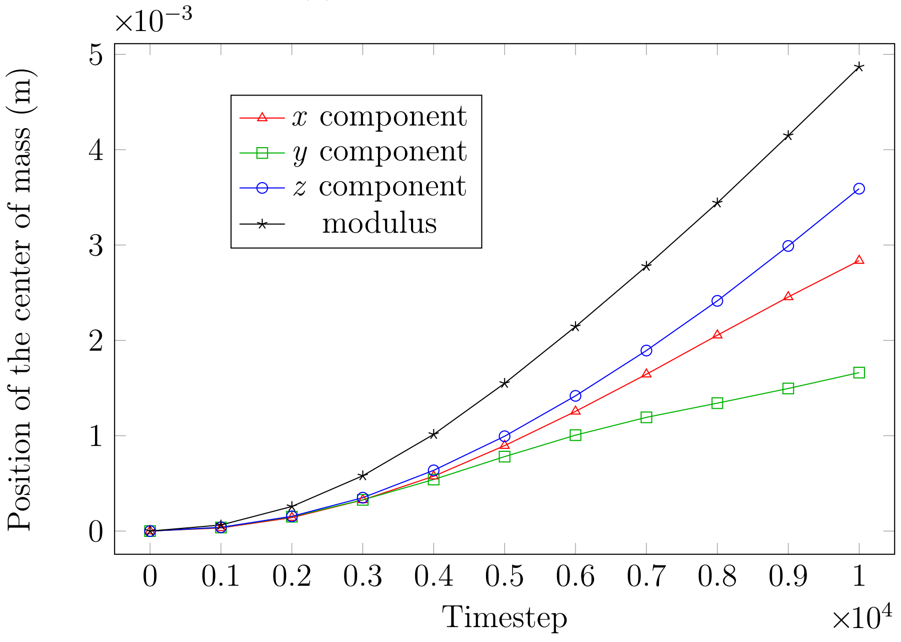
Table 7.3: Performance results for the two agglomerate scenario with 38,358 particles (large instance).

		octal tree		*binary tree*	
#particle processes	*#gravity threads*	*execution time(s)*	*avg. self-gravity time(s)*	*execution time(s)*	*avg. self-gravity time(s)*
1 (1,1,1)	1	$1.49{\times}10^4$	49.79	$1.89{\times}10^4$	64.40
1 (1,1,1)	2	$1.04{\times}10^4$	32.86	$1.34{\times}10^4$	42.94
1 (1,1,1)	4	$9.21{\times}10^3$	28.08	$1.10{\times}10^4$	35.38
1 (1,1,1)	8	$9.59{\times}10^3$	29.60	$1.10{\times}10^4$	35.37
1 (1,1,1)	16	$1.09{\times}10^4$	34.81	$1.16{\times}10^4$	36.75
2 (1,1,2)	1	$1.43{\times}10^4$	49.58	$1.90{\times}10^4$	65.97
2 (1,1,2)	2	$1.07{\times}10^4$	35.54	$1.27{\times}10^4$	42.79
2 (1,1,2)	4	$1.01{\times}10^4$	32.62	$1.10{\times}10^4$	35.81
2 (1,1,2)	8	$1.09{\times}10^4$	35.79	$1.02{\times}10^4$	33.92
2 (1,1,2)	16	$1.06{\times}10^4$	34.95	$1.12{\times}10^4$	36.32
4 (1,2,2)	1	$1.62{\times}10^4$	57.63	$1.88{\times}10^4$	65.91
4 (1,2,2)	2	$1.07{\times}10^4$	36.56	$1.49{\times}10^4$	52.46
4 (1,2,2)	4	$9.56{\times}10^3$	32.27	$9.72{\times}10^3$	32.64
4 (1,2,2)	8	$1.04{\times}10^4$	35.07	$9.82{\times}10^3$	33.09
4 (1,2,2)	16	$1.07{\times}10^4$	36.20	$1.09{\times}10^4$	37.28
8 (2,2,2)	1	$1.65{\times}10^4$	60.27	$1.74{\times}10^4$	62.80
8 (2,2,2)	2	$1.12{\times}10^4$	39.78	$1.11{\times}10^4$	39.23
8 (2,2,2)	4	$1.03{\times}10^4$	36.72	$8.83{\times}10^3$	30.94
8 (2,2,2)	8	$9.69{\times}10^3$	34.29	$9.58{\times}10^3$	33.86
8 (2,2,2)	16	$1.02{\times}10^4$	36.38	$1.06{\times}10^4$	37.10

Figure 7.1 shows the position of the center of mass (x, y, z components and its module) and its variation over time for the small instance for (a) the executions using the octal tree, and (b) the executions using the binary tree. Results confirm that the numerical accuracy using the binary and octal trees are of the same order of magnitude. However, the octal tree presented a slightly lower change in the position of the center of mass compared to the binary tree algorithm. The study of the numerical accuracy for the medium and large instances are reported in Figure 7.2 and Figure 7.3, respectively. Results support the commented trends for the small instance. In addition to the differences in accuracy, differences in the position of the components of the center of mass were detected when using the different tree structures. An example is shown in Figure 7.1, the position of the center of mass when using the octal tree moved away from the origin in the direction of the x component up to step $6,000$, but then went back to the origin, while this movement did



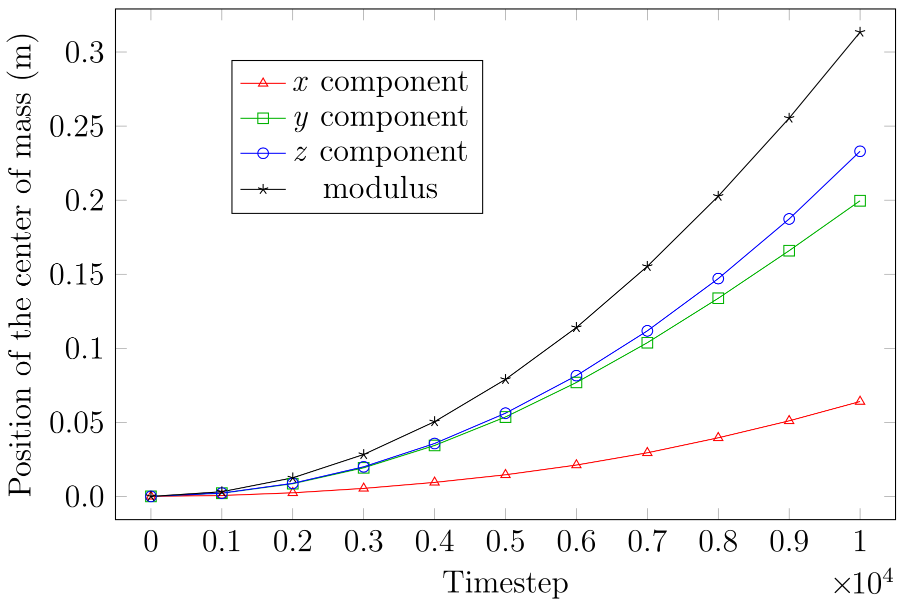
(a) Octal tree algorithm.



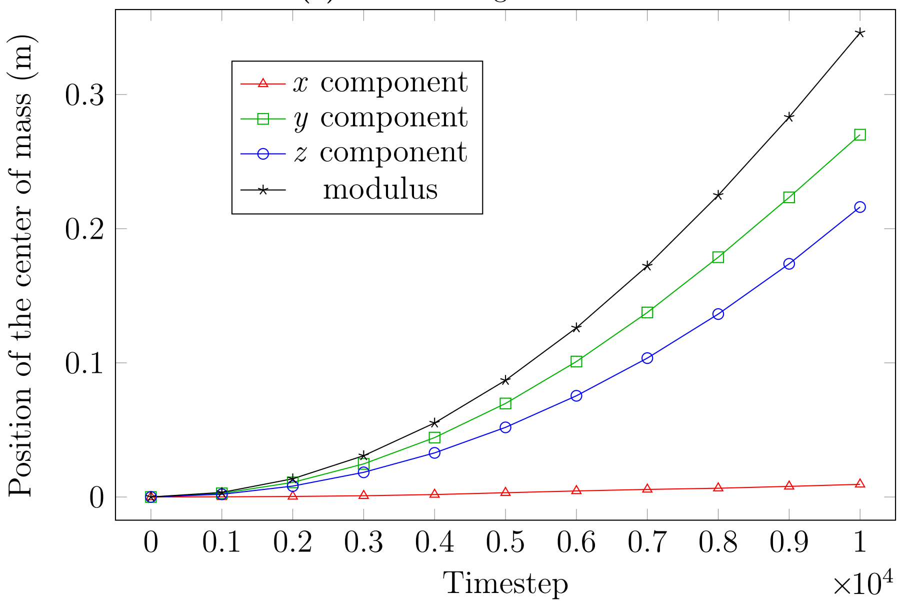
(b) Binary tree algorithm.

Figure 7.1: Position of the center of mass over time for the small instance of the two agglomerates scenario using the Barnes & Hut method with octal and binary tree.

not occur when using the binary tree structure. Either way, the modulus of the center of mass behaves in a similar way for the binary and octal trees. From the reported results, the method based on the binary tree emerges as robust alternative to the standard octal tree proposed by Barnes & Hut.

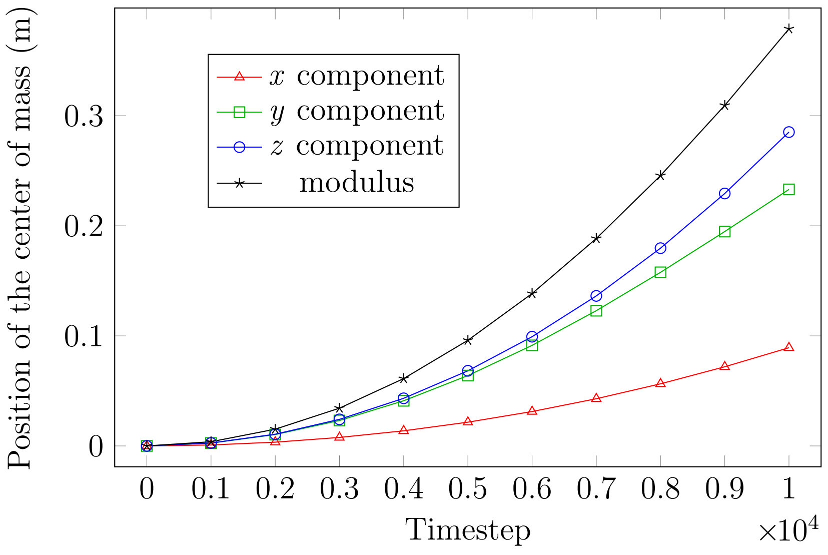


(a) Octal tree algorithm.

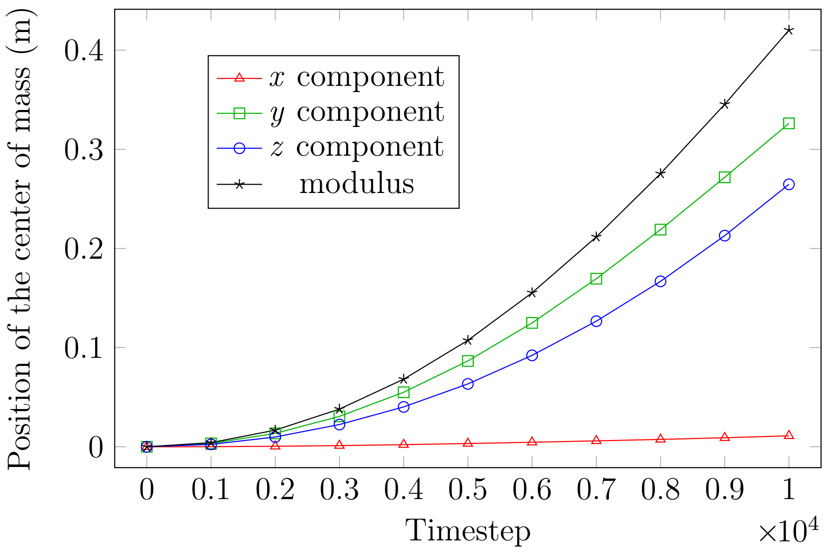


(b) Binary tree algorithm.

Figure 7.2: Position of the center of mass over time for the medium instance of the two agglomerates scenario using the Barnes & Hut method with octal and binary tree.



(a) Octal tree algorithm.



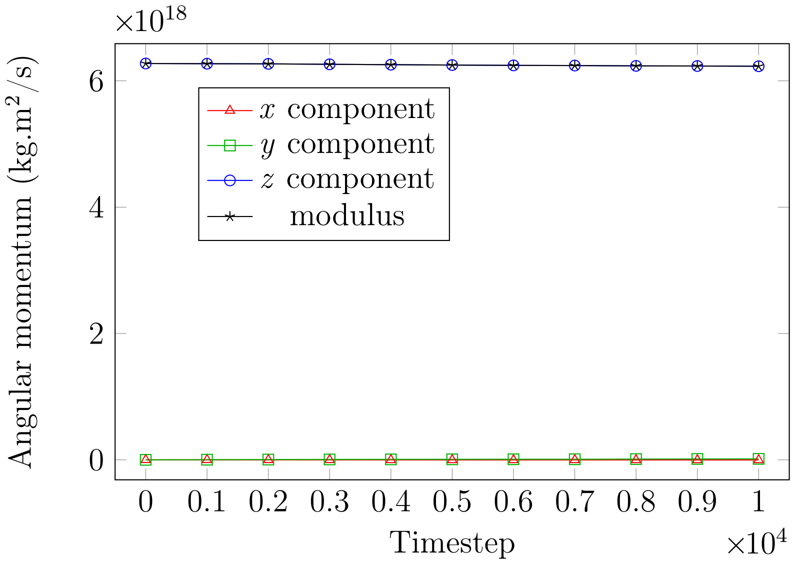
(b) Binary tree algorithm.

Figure 7.3: Position of the center of mass over time for the large instance of the two agglomerates scenario using the Barnes & Hut method with octal and binary tree.

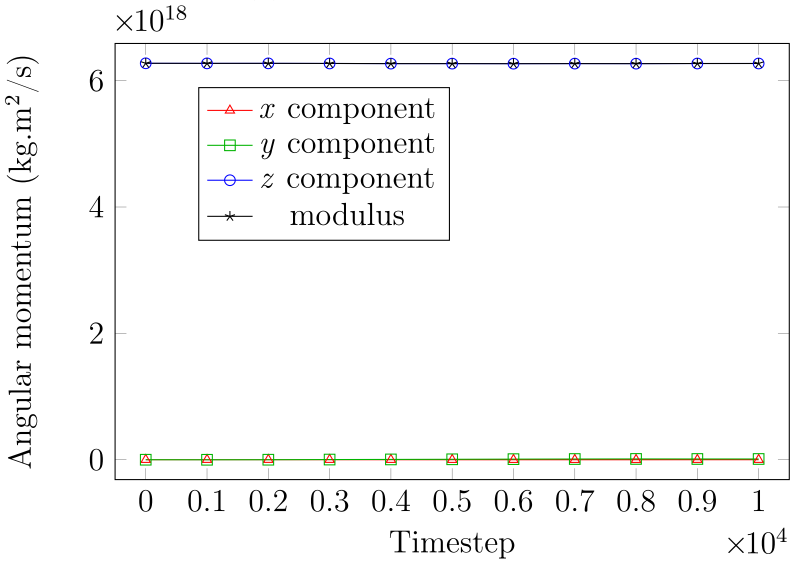
7.3 Numerical accuracy of the binary tree algorithm: analysis of the angular momentum

This section reports the results of the study of the angular momentum of the execution of the small, medium, and large instances of the two agglomerates scenario using the octal tree and the binary tree methods.

Figure 7.4a, and Figure 7.4b report the results of the study of the angular momentum for the execution of the two agglomerates scenario using the octal tree and the binary tree for the small instance.



(a) Octal tree algorithm.



(b) Binary tree algorithm.

Figure 7.4: Angular momentum over time for the small instance of the two agglomerates scenario using the Barnes & Hut method with the octal tree and the binary tree algorithm.

Then, Figure 7.5 and Figure 7.6 report the results for the medium instance. Finally, Figure 7.7 and Figure 7.8 report the results for the large instance. Results indicate that the angular momentum is of the same order of magnitude either using the octal tree or the binary tree. This results confirm that the simulation results are of the same quality in terms of conservation of the angular momentum of the system.

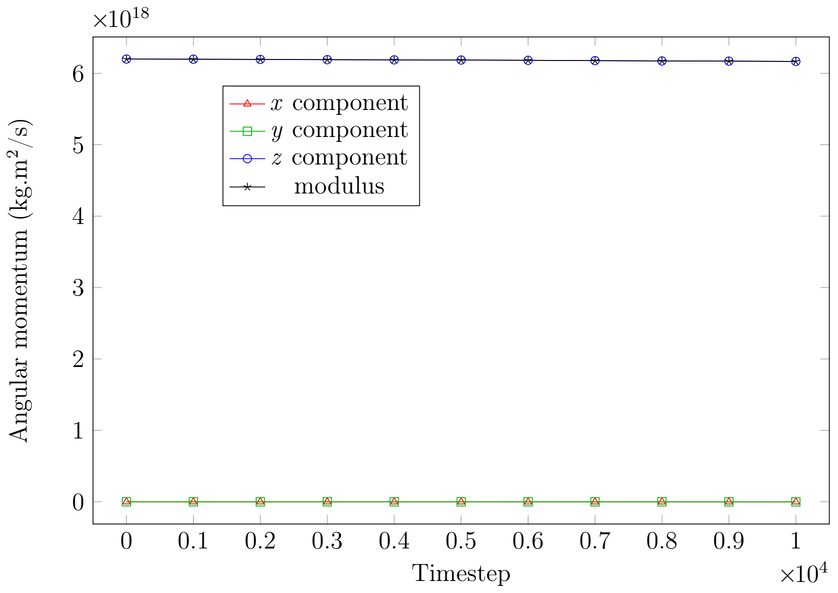


Figure 7.5: Angular momentum over time for the medium instance of the two agglomerates scenario using the Barnes & Hut method with the octal tree algorithm.

7.4 Analysis of the upper level direct calculation of the center of mass

This section reports the experimental results of the application of the higher level direct calculation of the center of mass introduced at the end of chapter 4. The results presented include the execution time of the simulations performed and the numerical accuracy of the calculation by the study of the movement of the center of mass of the system. In addition, a discussion of the results obtained is presented.

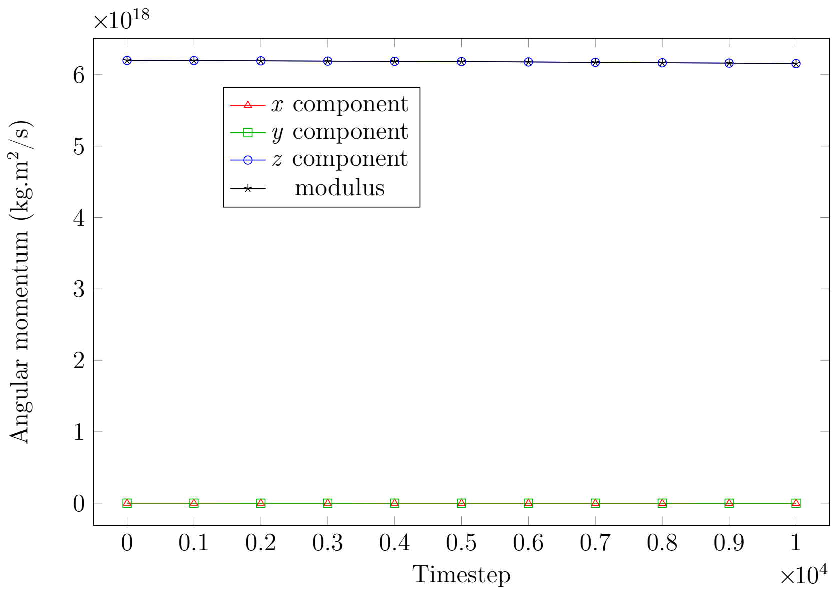


Figure 7.6: Position of the center of mass over time for the medium instance of the two agglomerates scenario using the Barnes & Hut method with the binary tree algorithm.

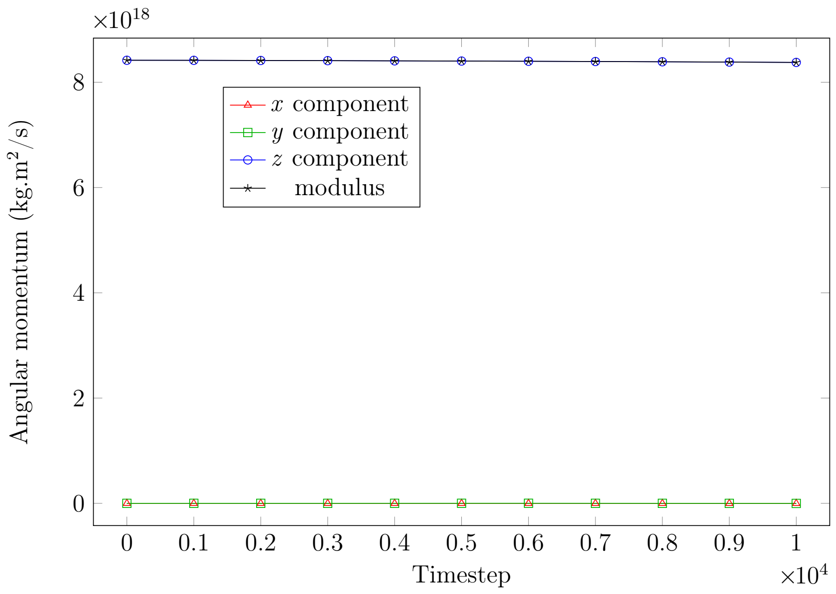


Figure 7.7: Angular momentum over time for the large instance of the two agglomerates scenario using the Barnes & Hut method with the octal tree algorithm.

The aim of the study is to prove that the precision of the calculation of the potential increases by calculating the center of mass in higher levels using particles. So, a study of the position of the center of mass was performed. The large instance of the two agglomerates scenario was considered. The study consisted of varying the neighborhood size of the boxes and then compare the

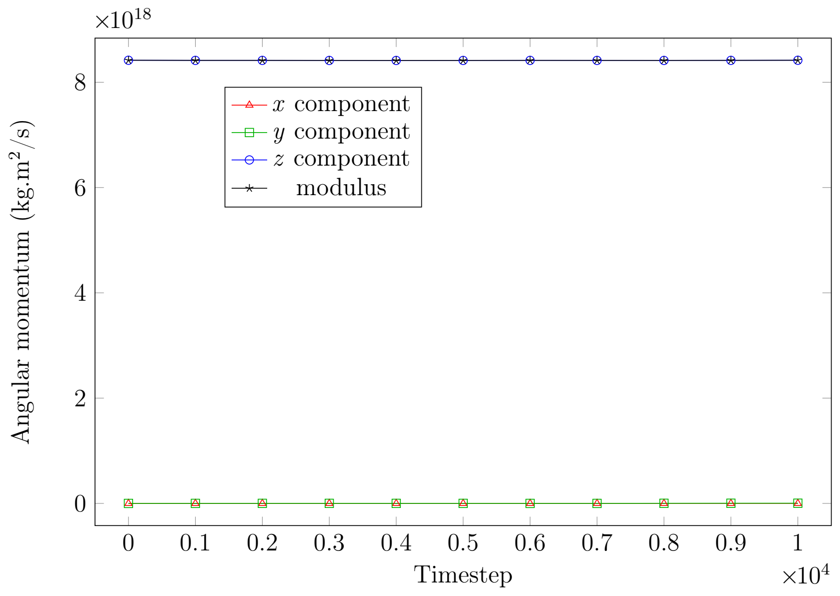


Figure 7.8: Angular momentum over time for the large instance of the two agglomerates scenario using the Barnes & Hut method with the binary tree algorithm.

movement of the position of the center of mass of the system over time.

Table 7.4 shows the execution time results obtained for the set of configurations defined and tested. All the executions were performed for 10, 000 time steps using the large instance of the two agglomerates scenario and a configuration of 8 processes and 8 threads. The first row of the table corresponds to the execution of the baseline configuration. Then, from the second row on, the results correspond to the execution of the Barnes & Hut octal tree using the adaptation introduced in Section 4.5 and increasing the neighborhood size by one for each configuration.

Figure 7.9 shows the movement of the center of mass over time for the configurations studied. Results indicate that the movement of the center of mass reduces as the neighborhood size increases. For the scenarios using a neighborhood size 0 to 3, every measure of the distance of the center of mass towards the center of the system is greater than the measure for the previous time steps. On the other hand, for the configuration with a neighborhood size 4, the center of mass starts to move nearer to the center of the system for the last part of the simulation.

The results presented for the study of the center of mass for the octal tree on Figure 7.3a show that using the octal tree algorithm, the center of mass of the system, for the scenario studied, moves $3.79{\times}10^{-1}$ m on time step 10, 000. On the other hand, using the higher level direct calculation, the center of

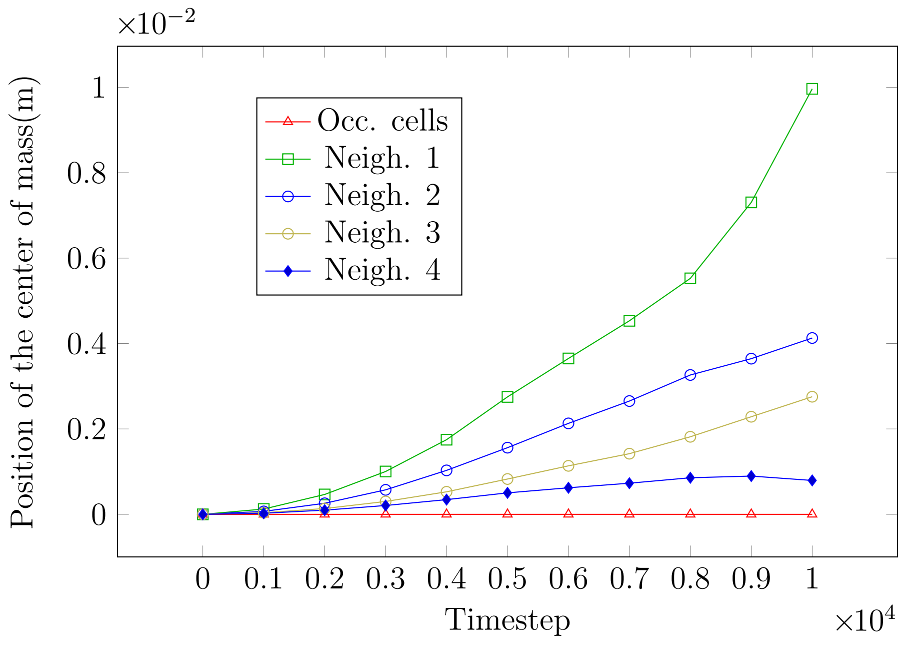


Figure 7.9: Position of the center of mass over time for the large instance of the two agglomerates scenario.

Table 7.4: Performance results for the large instance of the two agglomerate scenario.

neighborhood size	*execution time(s)*	*time computing self-gravity*	*# self-gravity updates*	*avg. self-gravity time(s)*
0	$1.80{\times}10^3$	$1.28{\times}10^3$	264	4.86
1	$1.73{\times}10^3$	$1.16{\times}10^3$	264	4.39
2	$1.96{\times}10^3$	$1.44{\times}10^3$	263	5.49
3	$2.99{\times}10^3$	$2.16{\times}10^3$	263	8.21
4	$3.00{\times}10^3$	$2.50{\times}10^3$	263	9.49

mass moves $7.94{\times}10^{-}4$ m on time step 10,000. This way, results shows that by applying the strategy presented in this section, the precision of the calculation improves up to four levels of magnitude.

Chapter 8

Conclusions and future work

This thesis presented the design, implementation, and evaluation of efficient parallel algorithms for self-gravity simulations in astronomical agglomerates. The algorithms are implemented as a self-gravity module in ESyS-Particle, a DEM simulator for geological phenomena. Three methods are presented and compared: the occupied cells method, and two variations of the Barnes & Hut method.

Before implementing the occupied cells method, a profiling of the self-gravity calculation was performed. The purpose of the profiling was to identify bottlenecks on the implementation, to be mitigated before the performance study. The profiling was performed using the two agglomerates scenario and executed for 10,000 time steps. After the profiling was performed, time consuming routines were identified and then specific modifications were included to reduce the number of invocations of those routines. The implemented modifications resulted in the occupied cells method, which consists of updating the acceleration of the occupied nodes of an overlying grid.

The two variations of the Barnes & Hut method proposed apply an octal and a binary tree for domain partitioning. Both methods consist of four stages: the creation of the self-gravity tree, the construction of the occupied nodes list, the computation of the self-gravity of the occupied nodes using the self-gravity tree, and the deletion of the tree. For the stage of the tree creation, the root of the tree is the whole simulation domain and tree nodes are created by refining the space of the simulation into eight regular cubical cells (octal tree) or into two non-cubical ones (binary tree). The refinement is recursively performed for each of the resultant nodes that have particles. The occupied nodes list is created using the occupied nodes and extending the list by adding the neighboring nodes for each occupied cell. Then the force vector is calculated for each of the occupied nodes plus the neighbors and the self-gravity tree. Finally, the tree is deleted and the calculated data is broadcasted to all the worker processes. A specific variation was also proposed at the stage of the creation of the self-gravity tree, in which the mass center of the tree nodes is calculated directly from the particles in an upper level of the octal tree.

The proposed methods for self-gravity calculation were evaluated over two realistic scenarios: two identical agglomerates orbiting each other and a collapsing cube scenario, including a instance with more than two million particles. Results showed that the Barnes & Hut method was 10× faster than the occupied cells method in simulations to compute self-gravity for the two agglomerates scenario and up to 100× faster for the collapsing cube scenario. The numerical accuracy was evaluated by studying the position of the center of mass and the angular momentum over the simulations. Results for two agglomerates scenario showed that using the occupied cells method the position of the center of mass varies in the order of 10^{-3} m, whereas it varies in the order of 10^{1} m when using Barnes & Hut. For the small instance of the collapsing cube scenario, the position of the center of mass using the occupied cells algorithm varied in the order of 10^{-4} m, while using the Barnes & Hut algorithm the position of the center of mass varied 10^{-2} m. Regarding the angular momentum, the difference in values had the same order of magnitude for both algorithms. The Barnes and Hut method is substantially faster than the busy occupied cells method. However, Barnes and Hut is less accurate than the occupied cells method in terms of numerical accuracy.

The comparison of the binary tree the octal tree variations of Barnes & Hut showed that the octal tree algorithm was up to 100% faster for the small instance, and 20% faster for the medium instance compared to the binary tree. On the other hand, the fastest execution time for the large instance was computed for the binary tree algorithm, suggesting that the binary tree algorithm performs faster than the octal tree for large instances. The numerical accuracy comparison indicated that both the position of the center of mass and the angular momentum vary in the same order of magnitude for both algorithms. Finally, results for the upper level calculation improved the accuracy of the calculations compared to the original Barnes & Hut. The center of mass moved in the order of 10^{-1} m using Barnes & Hut. However, using the upper level direct calculation, the center of mass moved on the order of 10^{-4} m.

The main lines for future work include extending the performance evaluation to consider larger, more realistic, and more diverse problem instances and scenarios, and proposing more strategies to improve the precision of the calculations over a simulation.

The future work also includes to propose strategies to improve the performance of the self-gravity module. One strategy is to update the Barnes & Hut tree partially during the simulation in order to increase the self-gravity update performance. Also, another strategy includes adapting the implemented Barnes & Hut algorithm to allow that the parameter that sets the box size to be any given value instead of only a power of two. Making the size of the boxes more flexible by accepting a wide range of values will allow performing simulations with tighter box sizes and hence allowing to reduce the bounding box, which in turn can help to improve the performance of simulations. Finally, another strategy is to improve the self-gravity overlaying grid to allow the cells to be of a different size than the cells that belong to the grid of contact forces that is located in the main module of ESyS-Particle. By decoupling the sizes of the boxes from these two grids, the size of the box of the contact forces grid can be close to or equal to the recommended two and a half times the radius of the largest particle. While, for the calculation of the self-gravity, a larger cell size could be used and thus improve the performance of the simulator.

Bibliography

Abe, S., Altinay, C., Boros, V., Hancock, W., Latham, S., Mora, P., Place, D., Petterson, W., Wang, Y., and Weatherley, D. (2009). Esys-particle: Hpc discrete element modeling software. *Open Software License version*, **3**.

Bagla (2002). Treepm: A code for cosmological n-body simulations. *Journal of Astrophysics and Astronomy*, **23**(3), 185–196.

Barnes, J. and Hut, P. (1986). A hierarchical O(N log N) force-calculation algorithm. *Nature*, **324**(6096), 446–449.

Binney, J. and Tremaine, S. (2011). *Galactic dynamics*. Princeton university press.

Bode, P., Ostriker, J. P., and Xu, G. (2000). The tree particle-mesh n-body gravity solver. *The Astrophysical Journal Supplement Series*, **128**(2), 561.

Cochran, W. T., Cooley, J. W., Favin, D. L., Helms, H. D., Kaenel, R. A., Lang, W. W., Maling, G. C., Nelson, D. E., Rader, C. M., and Welch, P. D. (1967). What is the fast fourier transform? *Proceedings of the IEEE*, **55**(10), 1664–1674.

Couchman, H. M. P. (1991). Mesh-refined p3m-a fast adaptive n-body algorithm. *The Astrophysical Journal*, **368**, L23–L26.

Cundall, P. and Strack, O. (1979). A discrete numerical model for granular assemblies. *Geotechnique*, **29**(1), 47–65.

Frascarelli, D., Nesmachnow, S., and Tancredi, G. (2014). High-performance computing of self-gravity for small solar system bodies. *Computer*, **47**(9), 34–39.

Fujiwara, A., Kawaguchi, J., Yeomans, D., Abe, M., Mukai, T., Okada, T., Saito, J., Yano, H., Yoshikawa, M., and Scheeres, D. (2006). The rubble-pile asteroid itokawa as observed by hayabusa. *Science*, **312**(5778), 1330–1334.

Greengard, L. and Rokhlin, V. (1987). A fast algorithm for particle simulations. *Journal of computational physics*, **73**(2), 325–348.

Harris, A., Fahnestock, E., and Pravec, P. (2009). On the shapes and spins of “rubble pile” asteroids. *Icarus*, **199**(2), 310–318.

Hockney, R. W. and Eastwood, J. W. (1988). *Computer simulation using particles*. CRC Press.

Intel (2017 (accessed July, 2017)). *Intel® VTune™ Amplifier 2017*. Available online at https://software.intel.com/en-us/intel-vtune-amplifier-xe.

Intel (2018 (accessed March, 2018)). *Intel Developer Zone - CPU Usage - Metric Description*. Available online at https://software.intel.com/en-us/vtune-amplifier-help-cpu-usage.

Ishiyama, T., Nitadori, K., and Makino, J. (2012). 4.45 pflops astrophysical n-body simulation on k computer: the gravitational trillion-body problem. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, page 5. IEEE Computer Society Press.

Khandai, N. and Bagla, J. S. (2009). A modified treepm code. *Research in Astronomy and Astrophysics*, **9**(8), 861.

Kravtsov, A. V., Klypin, A. A., Khokhlov, and Alexei, M. (1997). Adaptive refinement tree: a new high-resolution n-body code for cosmological simulations. *The Astrophysical Journal Supplement Series*, **111**(1), 73.

Nesmachnow, S. (2010). Computación científica de alto desempeño en la Facultad de Ingeniería, Universidad de la República. *Revista de la Asociación de Ingenieros del Uruguay*, **61**(1), 12–15.

Nesmachnow, S., Frascarelli, D., and Tancredi, G. (2015). A parallel multithreading algorithm for self-gravity calculation on agglomerates. In *International Conference on Supercomputing*, pages 311–325. Springer.

Nesmachnow, S., Rocchetti, N., and Tancredi, G. (2019). Large-scale multi-threading self-gravity simulations for astronomical agglomerates. In *2019 Winter Simulation Conference (WSC)*, pages 3243–3254. IEEE.

Rocchetti, N., Frascarelli, D., Nesmachnow, S., and Tancredi, G. (2017). Performance improvements of a parallel multithreading self-gravity algorithm. In *Latin American High Performance Computing Conference*, pages 291–306. Springer.

Rocchetti, N., Nesmachnow, S., and Tancredi, G. (2018). Comparison of tree based strategies for parallel simulation of self-gravity in agglomerates. In *Latin American High Performance Computing Conference*, pages 141–156. Springer.

Rozitis, B., MacLennan, E., and Emery, J. (2014). Cohesive forces prevent the rotational breakup of rubble-pile asteroid (29075) 1950 DA. *Nature*, **512**(7513), 174–176.

Sánchez, P. and Scheeres, D. (2012). Dem simulation of rotation-induced reshaping and disruption of rubble-pile asteroids. *Icarus*, **218**(2), 876–894.

Tancredi, G., Maciel, A., Heredia, L., Richeri, P., and Nesmachno, S. (2012). Granular physics in low-gravity environments using discrete element method. *Mnras*, **420**, 3368–3380.

Walker, D. W. and Dongarra, J. J. (1996). Mpi: a standard message passing interface. *Supercomputer*, **12**, 56–68.

Weatherley, D., V., B., Hancock, W., and Abe, S. (2010). Scaling benchmark of ESyS-Particle for elastic wave propagation simulations. In *IEEE Sixth International Conference on e-Science*, pages 277–283. IEEE.

Xu, G. (1994). A new parallel n-body gravity solver: Tpm. *arXiv preprint astro-ph/9409021*.