# High performance computing simulations of self-gravity in astronomical agglomerates

Néstor Rocchetti, Sergio Nesmachnow and Gonzalo Tancredi

**Submission for the Special Issue on High Performance Computing Activities in the Ibero American region (S19-1)**


### Abstract

This article describes the advances on the design, implementation, and evaluation of efficient algorithms for self-gravity simulations in astronomical agglomerates. Three algorithms are presented and evaluated: the occupied cells method, and two variations of the Barnes & Hut method using an octal and a binary tree. Two scenarios are considered in the evaluation: two agglomerates orbiting each other and a collapsing cube. Results show that the proposed octal tree Barnes & Hut method allows improving the performance of the self-gravity calculation up to $100\times$ with respect to the occupied cell method, while having a correct numerical accuracy. The proposed algorithms are efficient and accurate methods for self-gravity simulations in astronomical agglomerates.



### Keywords

Simulation, high-performance computing, self-gravity, astronomical agglomerates


## 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 article 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 allows improving the performance of the self-gravity calculation up to $100\times$ with respect to the occupied cell method. This way, efficient simulations are performed for the largest problem instance including 2,097,152 particles.


Universidad de la República, Uruguay

**Corresponding author:**
Néstor Rocchetti, Universidad de la República, Uruguay.
Email: nrocchetti@fing.edu.uy

This article extends our previous conference articles: "Comparison of Tree Based Strategies for Parallel Simulation of Self-gravity in Agglomerates" (Rocchetti et al. 2018) and "Large-Scale Multithreading Self-Gravity Simulations for Astronomical Agglomerates" (Nesmachnow et al. 2019). The main contributions of this article 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 article is organized as follows. Next section 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 section 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.

## Self-gravity simulations

This section describes the problem of simulating self-gravity interactions in agglomerates and reviews relevant related works on self-gravity simulation applying domain decomposition techniques.

### *The self-gravity calculation problem*

The self-gravity calculation problem consists of computing the self-gravity of assemblies of $N$ particles. In this problem, every particle in the assembly interacts (and is affected by) all 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 Equation 1, where $G$ is the gravitational constant, $\| \overrightarrow{r} \|$ is the norm of vector $\overrightarrow{r}$, $M_i$ is the mass of the particle, and $V_j$ is the gravitational potential of particle $j$.

$$V_j = \sum_{i \neq j} \frac{GM_i}{\| \overrightarrow{r_j} - \overrightarrow{r_i} \|} \qquad (1)$$

A straightforward implementation of the gravitational potential calculation according to Equation 1 results in a computational cost of $O(N^2)$ when calculating the velocity 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.

### *Related work on spatial domain decomposition*

This subsection reviews static and dynamic domain decomposition techniques to speed up the self-gravity calculation when performing astrophysical particle simulations.

*Static hierarchical domain decomposition.* Static techniques for particle interaction are based on a domain decomposition that stays invariable during a simulation. Static techniques are divided in three models: Particle-Particle (PP) methods, Particle-Mesh (PM) methods, and Particle-Particle Particle-Mesh (P3M) methods (Hockney and Eastwood 1988). PP methods compute the forces directly between all particles in the system. Despite being the most simple and accurate, PP methods do not scale in the number of particles, due to its $O(N^2)$ execution time. PM methods use a mesh over the simulated space and compute the potential for each particle that belongs to the mesh. After that, the speed for the particles is calculated via an interpolation. Although PM methods are less accurate than PP methods when computing the forces, they are faster. 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. Short range forces are calculated using a PP method and 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 most relevant are reviewed next.

Couchman (1991) proposed a P3M algorithm that applies a selective refinement of the grid recursively when the particle density of a cell exceeds a threshold. The particle mesh is an spatial division without overlapping zones, so load balancing is achieved by assigning a similar number of particles to each processing unit. 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 had two drawback: the domain of the simulation must be cubic and 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 indicated that the algorithm is up to 20 times faster than a P3M algorithm and also requires less memory.

Kravtsov et al. (1997) presented the Adaptive Refinement Tree N-body solver, based on the PM method over a multilevel grid to perform the force calculations. 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 equally-sized parts, depending on the particle density. The multilevel mesh is created at the beginning of the simulation and 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 (a cell). The implementation was partially parallel (just the update of the forces in the particles). Kravstov et al. stated that the solution is half as fast as the Fourier transform solver with the same number of cells.

Sánchez and Scheeres (2012) implemented a PM simulator using the same static domain decomposition to compute short and long range interactions. The simulation space is divided in cubical cells that are many times bigger than the particles radius. Then, instead of considering individual particles to calculate the gravitational potential, a whole cell is considered as a particle. The method is still $O(N^2)$, but the authors claimed that the required calculations decrease in one order of magnitude. Short range and long range interactions are computed concurrently, but no parallel implementation was proposed. Thus, the method was only able to perform efficient simulations of systems with up to $8,000$ particles. Results show that the algorithm implemented is not suitable to perform large simulations comprising hundreds of thousands particles. In order to do so, a parallel/distributed implementation is needed.

*Dynamic hierarchical domain decomposition.* Dynamic techniques for particle interaction use structures that are adapted or reconstructed from scratch during the simulation. A classic dynamic technique is the Barnes and Hut (1986) method. The simulated domain is divided in a hierarchical octal tree to accelerate the calculation of the gravitational potential in $N$-body simulations. Nodes of the tree represent a portion of the simulation space. If a node (i.e., the space it represents) contains more than one particle, the space is divided into smaller pieces (in a 2D simulation the space is divided in four smaller pieces, in a 3D simulation, the space is divided into eight pieces). Applying this domain decomposition, long-range interactions are calculated in $O(N \log N)$, but the error increases as the simulation runs.

The Fast Multipole Method (Greengard and Rokhlin 1987) performs an expansion of the simulation space, organized as a hierarchy of meshes rather than a tree. Authors claim that clusters of particles can interact with one another as long as they are "well separated" (considering a predefined parameter). The expansion is pure spatial. The spatial center of a father mesh box is an expansion center of the next level. The potential is computed for the center of all meshes. Then, the potential for each particle is extrapolated from the nearest neighbor boxes at the finest mesh level. The algorithm was implemented and compared to a direct computation. Results of the comparison indicate that in an instance consisting of 12,800 particles the execution time of the method presented is 300× lower than the direct method.

*Combined techniques.* Xu (1994) presented Tree Particle Mesh (TPM), a variation of the P3M method that computes short-range interactions using the Barnes & Hut algorithm. However, long-range interactions are calculated using a PM method. TPM applies trees (using different time steps) only on the parts of the field that have a density over a certain threshold. Regions with low particle density are managed only by the PM method. TPM is faster because less cells are created when using the tree. TPM was parallelized using a Multiple Instruction Multiple Data model. To provide load balance, the tree code was parallelized in two levels. After constructing a tree, its structure is broadcasted to processes that will perform the tree walk and force summation. The tree construction was not parallelized because it only demands 2.5% of the time. TPM had a speedup of 26 on 32 processors and was 12× faster than PM.

Bode et al. (2000) improved the TPM method, combining PM to compute long range interactions and a tree to compute short range interactions. PM and the tree algorithm starts with the same time step, but the latter can use a smaller time step if needed. The space is divided into *clusters*, grouping cells based on particle density. A comparison with the P3M algorithm showed that TPM speed up the simulations "by a factor of 3–4" when the trees have individual time steps. The algorithm scales properly from 4 to 16 processors, then efficiency dropped to 70% in 32 nodes.

Bagla (2002) presented the TreePM method for simulating agglomerates, combining PM and Barnes & Hut to compute short range forces. For short range forces, the contribution is computed from particles or from grid cells when a distance threshold is exceeded. TreePM and TPM methods differ in two aspects: i) TPM uses the PM method to compute long range forces, TreePM uses an explicit distance to decide which method is used to compute the forces in a region, and ii) in TPM forces are computed for each individual particle only in high density regions, but in TreePM the forces are computed individually for all particles. Results indicated that TreePM is about 4.5× faster than a tree algorithm and the simulations scaled in O($N \log N$).

Khandai and Bagla (2009) presented a modification of TreePM algorithm using a tree to calculate the short range forces. The force is computed in only one tree walk. In addition, 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 TreePM for a N-body simulation of one trillion particles executed using K Computer (Japan) at full capacity (663552-cores). For short-range forces, the Barnes & Hut algorithm was modified to create trees for groups of particles and a list of shared tree nodes and particles was implemented. Authors claim that TreePM "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. A 3D multi-section decomposition was applied and load balancing adjusts the size of the cells according to the cost of calculating their potential. As a complement, the interactions were calculated using the Single Instruction Multiple Data paradigm, combined with loop unrolling. Regarding the PM method, a distributed Fast Fourier Transform was implemented. 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. The algorithm achieved a performance of 4.45 Pflops and an efficiency of 43% using 82,944 computing nodes.

The analysis of the related works allows identifying several proposals for N-body simulations applying spatial domain decomposition for force calculation in agglomerates. However, no previous work has proposed an efficient method for self-gravity calculation in astronomical agglomerates implemented on a wide purpose public simulation library, as this article contributes.

## A parallel algorithm for self-gravity calculation

This section presents the proposed algorithmic approach for parallel self-gravity simulations.

### DEM-based self-gravity simulations

Our previous work (Frascarelli et al. 2014) presented an algorithm based on DEM to perform self-gravity calculations and contact forces calculations for simulating small solar system bodies. The algorithm applies parallel multithreading techniques to accelerate the calculation of long-range forces and also short-range contact forces. For 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 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 studied 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 GB RAM from Cluster FING (Nesmachnow 2010). To test the numerical accuracy, an scenario containing an agglomerate of 1,022,208 particles with a radius of 20m was used. Results showed an error less than 0.1% of the theoretical center of mass. Computational efficiency results showed a near linear speed up when using approximately 100,000 particles. Nevertheless, the computational efficiency decreased as the number of particles increased.

Later, our work (Nesmachnow et al. 2015) studied 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 that calculate self-gravity: *interlocking linear*: 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; *circular concentric*: 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; *basic isolated linear*: consists of assigning clusters of cells of equal size to each thread of the pool; and *advanced isolated linear*: divides the workload evenly between the threads spawned and after a thread finishes processing its workload, it starts processing the nearest unprocessed cell; until there are no cells left to process. the Advanced isolated linear strategy obtained the best results, being able to scale up linearly with the number of particles in the system, and with an inverse power law (exponent 0.87) with the number of threads. The observed speed up was close to linear for systems containing up to $2{\times}10^5$ particles.

The previous self-gravity algorithms were not integrated in a tool or software library. Next subsection describes how the self-gravity algorithm was included in the ESyS-Particle particle interaction simulator to perform realistic simulations in low gravity environments.

### Implementation of the self-gravity algorithm on ESyS-Particle

ESyS-Particle was developed to simulate geophysical phenomena, so the included particle interacions only comprise contact forces. Weatherley et al. (2010) analyzed the scalability and the accuracy of the calculations of contact forces algorithm in ESyS-Particle. Depending on the number of particles, every time the total forces of the particles are updated, the computation can last from seconds to hours.

When including self-gravity in ESyS-Particle, it must be considered that the number of calculations for $N$ particles and $M$ nodes in the grid is $O(N{\times}M)$. To deal with that efficiency problem, HPC techniques were applied to speed up the calculations. ESyS-Particle implements an static spatial domain decomposition using a master-slave model implemented on MPI. The domain decomposition is described by numerical parameters, one for each axis, indicating how many times each dimension must be divided. E.g., (1,2,2) means that at the beginning of the simulation, the $x$ dimension is not divided, while dimensions $y$ and $z$ are divided into two sections; thus, the space is divided into four subdomains. Each subdomain is assigned to a process, ideally dividing the total calculations by a factor of four.

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. 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 is based on a master-worker model, but including a two-level parallelization scheme: i) a master-worker model is used to calculate and update the forces that affect the particles, 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; and ii) the calculation of self-gravity forces is implemented using shared memory and multithreading programming techniques.

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 $m_j$, radius $r_j$, and positions $(x_j, y_j, z_j)$ is given by Equation 2. Similar equations are formulated for the acceleration along the $y$-axis (Equation 3) and the $z$-axis (Equation 4).

$$a_x = \sum_{j=1,N} G m_j \frac{x_j - x}{r_j^3} \tag{2}$$

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

$$a_z = \sum_{j=1,N} G m_j \frac{z_j - z}{r_j^3} \tag{4}$$

Figure 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.

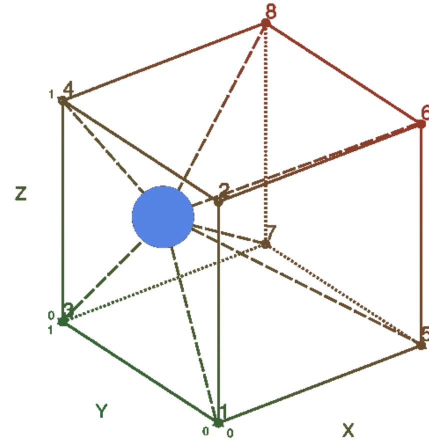



**Figure 1.** Box of the self-gravity grid with eight nodes.

## Adapted Barnes & Hut method for self-gravity calculation

This section describes the Barnes & Hut octal tree used for the self-gravity calculation in ESyS-Particle and a binary implementation applied for results comparison.

### *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, knowing if a node is son of another is a constant time operation $O(1)$. Dividing by $10_8$, the identifier of a node is equivalent to performing a shift operation of three bits to the right. Figure 2 shows a sample two-dimensional tree partition created for an agglomerate of particles and an illustration example of the space partitioning of the adapted Barnes & Hut method. The grid over the agglomerates represents the quadrupole (octapole in three dimensions) tree resulting of the application of the 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.

### *Creation of the octal tree*

The implementation of the Barnes & Hut algorithm 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.

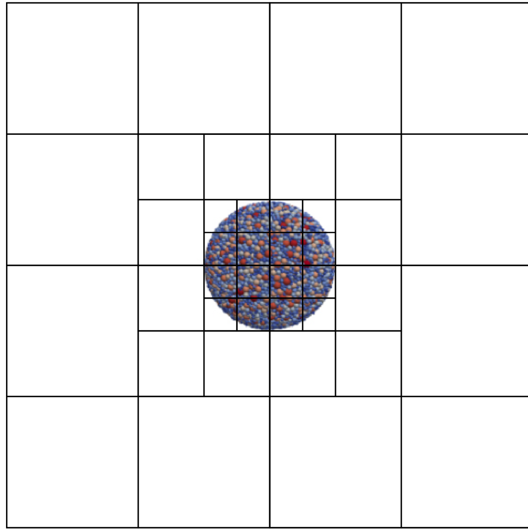


**Figure 2.** Example of tree partition for an agglomerate of particles (a two dimensions projection is shown for better representing the division method).

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 1 describes the process of calculating the self-gravity potential using the proposed Barnes & Hut algorithm. First, the expanded occupied cells list is created (lines 2–3), 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 (lines 4–5). The creation of the tree node list of each of the expanded occupied cells comprises the loop in line 6. Then, the third step calculates the potential of the nodes in the loop in line 9. The output of step 3 is the new potential of the expanded occupied cells list, which is communicated to the worker nodes to calculate the potential of the particles.

**Algorithm 1** General octal tree algorithm

```
1:  procedure RECALCULATE SELF-GRAVITY
2:      OC list ← getOccupiedCells()
3:      expanded OC ← getBoxRecalculate(OC list)
4:      ot ← createOctalTree()
5:      calculateCentersOfMass(ot)
6:      for each oc in expanded OC do
7:          list TN ← createListOfNodes(oc, ot)
8:          add(list of list of TN, list of TN)
9:      for each oc in expanded OC do
10:         update ← updateSelfGravity(oc, list of list of TN)
11:         add(updated cell list, updated cell)
12:     communicateNewValues(updated cell list)
```

*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 & 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 3 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 3 is not calculated because it has no particles.

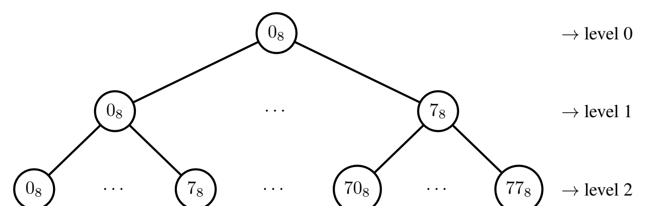



**Figure 3.** Sample of octal tree created using the described algorithm for a self-gravity grid composed of 64 boxes.

Algorithm 2 summarizes the creation of the octal tree. The height of the tree is calculated in line 2 and used in line 3 to create a queue of node levels used as an auxiliary structure. Then, the creation of the tree starts with by instantiating a root node (line 4), which is added to the level queue position that corresponds to the level 0 of the tree (line 5). The created auxiliary structure is used to create the tree nodes and organize them. The creation of the octal tree (lines 6–11) consists of two loops: the outer loop iterates 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. The outer loop performs the change of level. In the inner loop, for each node, the list of sons that correspond to that node is created (line 9) and associated to the lower level queue (line 10). Lines 14–18 correspond to particle counting and center of mass calculation for the nodes. The octal tree is covered by level from the lowest level nodes (i.e., nodes that match with the boxes) up to the root node. The particle count and mass center of lower level nodes are used to calculate the values for the upper levels. Finally, the root node is returned as the representant of the complete octal tree (line 19). 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 2** Creation of the octal tree

```
1:  procedure CREATE OCTAL TREE()
2:      tree height ← calcTreeHeight(dim, box length)
3:      NL queue ← createQL(tree height)
4:      root node ← createNode()
5:      NL queue.at(0).push(root node)
6:      for tree level = 0 to tree height - 1 do
7:          LLqueue ← NL queue.at(tree level + 1)
8:          for each node in upper level queue do
9:              LL node sons ← createSonsList(NQ)
10:             LL queue ← add(LL node sons)
11:         NQ ← LL queue
12:     root node ← getFront(NQ)
13:     # Calculate particle count and center of mass
14:     for tree level = tree height - 2 to 0 do
15:         level NQ ← node level queue.pop()
16:         for each node in level NQ do
17:             calculateParticleCount(node)
18:             mass center ← MassCenterNode(node)
19:     return root node
```

Algorithm 3 summarizes the creation of the eight sons of a not leaf node in the octal tree. First, the list of sons to be returned is created. The id of the first son is calculated, shifting 3 places the father id. The size of each node is half the size of the father node. For each of the eight sons, the position of the node is calculated, then the node is created and added to the sons list.

*Creation of the tree nodes list.* 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*. The concept of *neighborhood* of a node is introduced. A neighborhood of a node is conformed by its surrounding nodes that do not exceed a user defined distance threshold. Each list of tree nodes is composed of the highest level

**Algorithm 3** Create list of sons

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

nodes that are not father of any node that belongs to the neighborhood. For each objective node, the algorithm starts with the root node. The `getProcessableNodes` routine implements the creation of the list of processable nodes for the neighborhood of an objective node (Algorithm 4). First, a list of neighboring nodes from the objective node is created. Then, the octal tree is covered, starting from the root node, with the purpose 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. The list of nodes is used as input for the third step, the update of the potential of a node.

**Algorithm 4** Get processable nodes

```
1:  procedure GET PROC NODES(obj node, octal tree, nr)
2:      NN ← getNeighCells(obj node , nr)
3:      processable nodes ← create queue()
4:      NQ ← create queue()
5:      NQ.push(octal tree)
6:      while !NQ.isEmpty() do
7:          node ← NQ.pop()
8:          if isFatherOfNeighMember(node, NN) then
9:              if !node.is box() then
10:                 NQ.push(node.get sons queue())
11:             else
12:                 processable nodes.push(node)
13:         else
14:             processable nodes.push(node)
15:     return processable nodes
```

*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 5 presents the self-gravity calculation process of an occupied node. The process is called after creating the octal tree and the list of processable nodes for the occupied node neighborhood. In the algorithm, for each processable node, the force vector respect to the occupied node is calculated and added to the *total force vector*. The *total force vector* is returned to be broadcasted to all the worker processes of ESyS-Particle.

**Algorithm 5** Self-gravity calculation of a node

1: **procedure** UPDATE SELF-GRAVITY(*on*, *ot*, *pn*)
2: *total force vector* ← createEmptyForceVector()
3: **for each** *pnode* **in** *pn* **do**
4: *force vector* ← processBHNode(*on, pn*)
5: sumFV(*total force vector*, *force vector*)
6: return *total force vector*

### *Barnes & Hut implementation on ESyS-Particle*

The self-gravity module on ESyS-Particle was extended to implement the Barnes & Hut method. Two new classes were included: `Barnes_And_Hut_Node` and `Barnes_And_Hut_Manager` class. `Barnes_And_Hut_Node` implements the basic functionalities of the node. It is responsible for the creation of a new node and holds the getters and setters of the attributes of the class, and the dispose functions. `Barnes_And_Hut_Manager` hosts the core functions of the Barnes & Hut method. This class is responsible of creating and disposing the octal tree in each update of the self-gravity, implementing the algorithms that creates the list of tree nodes and 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 `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.

### *The binary tree*

This subsection describes the binary tree and the main differences with the octal tree implementation.

*Structure and creation of the binary tree.* Figure 4 shows a sample binary tree for a self-gravity grid composed of 64 boxes. The generated tree has seven levels. Each node has a unique identifier in the corresponding level (an integer in binary code). A node is the father of another if it satisfies $id_f = id_s/(10_2 \times (level_s/\ level_f)$, 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 $O(1)$. Instead of dividing by $10_8$ (as in the octal tree), the division is performed by $10_2$ by a right shift operation.

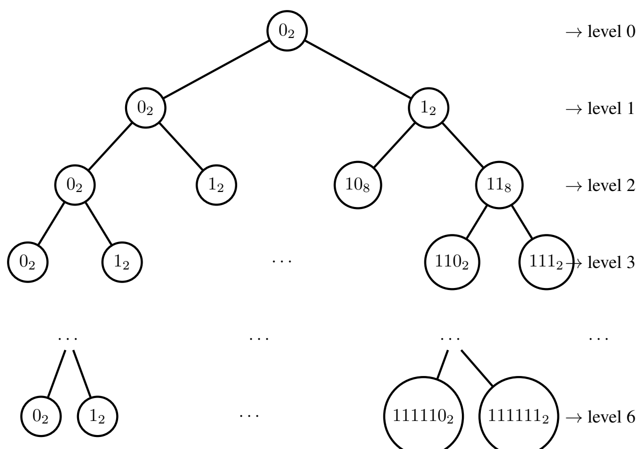



**Figure 4.** Example of enumeration for a binary tree with seven levels for a self-gravity grid composed of 64 boxes.

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

*Comparison of the binary tree and the octal tree.* Node $77_8$ of Figure 3 is taken as an example to perform the trees comparison. This node corresponds to $111111_2$ in the binary tree. Assuming that all boxes are occupied and the neighborhood size is zero, in the binary tree the list of tree nodes is comprised of five elements: 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). 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 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 three bits, a one bit shift is applied to multiply the nodes identifiers by 2. Another difference in the implementation of the binary tree with respect to the octal tree is the tree height calculation process, which involves the three axis instead of only considering the $x$ axis as in the octal tree. The largest 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 those differences, the structure and 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 ESyS-particle.

### *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, computing it 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 computed using the values obtained for the son nodes of the immediate lower level. This means that the center of mass of the higher level is computed based on a previous calculation, thus there is a loss of precision of the calculation of the potential as a consequence.

The octal tree algorithm was modified 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.

## Test scenario and instances

This section describes the test scenarios and instances used in the experimental evaluation of the parallel self-gravity calculation implemented in ESyS particle profiling results of the optimized version.

### 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 are symmetrical with respect to the origin of the coordinates system (point (0,0,0)). Thus, the center of mass is located in the point (0,0,0), 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 1 reports details of instances generated based in the two agglomerates scenario. Three instances were created: i) a small instance, composed of $3,866$ particles, each one having a radius from 50 m to 100m; ii) a medium size instance with $11,100$ particles with a radii from 35 m to 70 m; and iii) a large instance 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 $1.2{\times}10^{12}$ kg to $1.7{\times}10^{12}$ kg. However, the masses of the instances are of the same order of magnitude. Figure 5 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.

**Table 1.** Instances of the two agglomerates scenario.

| *instance name* | *# particles* | *particle radius (m)* |
|---|---|---|
| small | 3,866 | 50-100 |
| medium | 11,100 | 35-70 |
| large | 38,358 | 20-60 |

All instances were simulated for $100,000$ time steps of $0.01$ s 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.

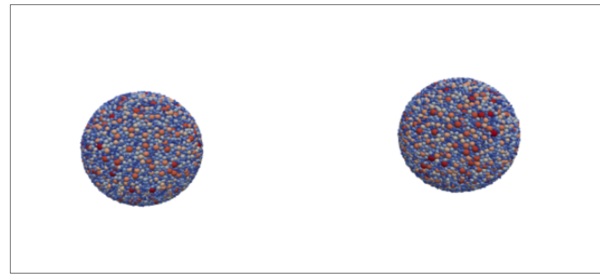


**Figure 5.** The two agglomerate scenario

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 s each (a total time of 100 s). 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.

### Free falling symmetric cube

The second scenario consists of a cube of 1 km on each side, filled with spherical particles with the same radius and density. The particles are located in a cubical box and separated from one another at least a distance that is equivalent to 1/6 of a particle radius. Also, the particle radius is bounded by the dimension of a box. The particle radius is 3/6 of a box edge. 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) 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 m/s and a density of 3000 g/cm$^3$. In this scenario, the cube should collapse in the direction of its center of mass.

The free-fall equation of a sphere 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 (small, medium, and large). The cube for all instances 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. Table 2 presents the details about the generated instances and Figure 6 shows an example representation for the small instance.

**Table 2.** Instances of the free falling symmetric cube scenario.

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

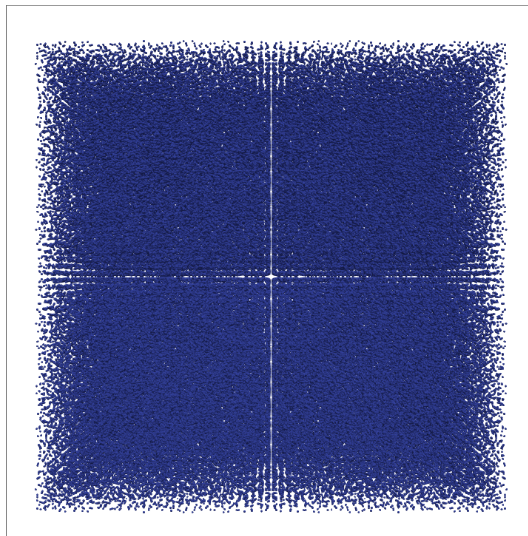


**Figure 6.** Example of tree partition for the cube scenario (initial state of the small instance of 32,768 particles).

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 and 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, the memory used by the binary tree is approximately 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 s each (a total time of 100 s). 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.

### Hardware platform

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, Universidad de la Republica, Uruguay (Nesmachnow 2010).

### Profiling the optimized version of self-gravity calculation

A profiling of the self-gravity module before and after the implementation of the occupied cells method was performed. The profiling was performed using VTune amplifier by Intel.

The main results of the profiling showed that method `BoxCoords::getZ`, which consumed 2269 s in the non-optimized version, had a negligible execution time in the occupied cells implementation. A similar behavior was identified for routines `SharedMemoryManager::getBox`, `BoxCoords::compare`, `Box::getParticlesCount`, and `SharedMemory::getBox`. The modifications in routine `OnlyOccupCellsProcStrategy::getNextOrigin`, that searches for the next node to process, improve the execution time from consuming 704 s to consume a negligible time compared to the other routines. Full details of the profiling analysis are reported in Rocchetti et al. (2017). Finally, improvements on routine `Point::getX` allowed it to execute 8 times faster (from 876 s to 109 s). These results account for notably performance improvements when using the new proposed schema for computing self-gravity. The new implementation allows scaling up to perform larger simulation, i.e., involving millions of particles, in reasonable execution times. Next section reports the full experimental evaluation of the implemented methods.

## Experimental evaluation: occupied cells versus octal tree

This section describes the performance and numerical analysis of the occupied cells and the octal tree self-gravity calculation methods implemented in ESyS-Particle for the two agglomerate scenario and the collapsing cube scenario. First, performance results are reported and discussed. Then, the numerical accuracy of the results is analyzed, by studying the position of the center of mass and the conservation of the angular momentum for the two methods.

### Results for the two agglomerates scenario

This subsection reports the performance evaluation results for the large instance of the two agglomerates scenario. The complete results for the small and medium size instances are reported in our previous work (Nesmachnow et al. 2019).

Table 3 reports the performance results for the large instance using the implemented Barnes & Hut method. 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 configuration (1,1,1) and 16 gravity threads) to 88% (using configuration (1,2,2) and 4 gravity threads), meaning that most of the execution time was spent on self-gravity calculations rather than on contact forces calculation. With regard to the average self-gravity calculation time, the lowest value was 11.01s, using one process and eight

**Table 3.** Performance results of the Barnes & Hut method for the large instance of the two-agglomerate scenario.

| *particle processes* | *gravity threads* | *execution time(s)* | *self-gravity time* | *self-gravity updates* | *avg. self-gravity time(s)* |
|---|---|---|---|---|---|
| 1 (1,1,1) | 1 | $8.58\times10^4$ | 71% | 3813 | 16.06 |
| 1 (1,1,1) | 2 | $7.45\times10^4$ | 76% | 3813 | 12.22 |
| 1 (1,1,1) | 4 | $6.00\times10^4$ | 74% | 3813 | 11.67 |
| 1 (1,1,1) | 8 | $5.48\times10^4$ | 77% | 3813 | 11.01 |
| 1 (1,1,1) | 16 | $8.32\times10^4$ | 68% | 3815 | 14.84 |
| 2 (1,1,2) | 1 | $7.74\times10^4$ | 77% | 3807 | 15.65 |
| 2 (1,1,2) | 2 | $6.34\times10^4$ | 69% | 3807 | 11.48 |
| 2 (1,1,2) | 4 | $5.96\times10^4$ | 71% | 3813 | 11.13 |
| 2 (1,1,2) | 8 | $6.52\times10^4$ | 76% | 3807 | 13.09 |
| 2 (1,1,2) | 16 | $7.36\times10^4$ | 76% | 3807 | 14.69 |
| 4 (1,2,2) | 1 | $7.32\times10^4$ | 83% | 3781 | 16.08 |
| 4 (1,2,2) | 2 | $6.31\times10^4$ | 75% | 3781 | 12.43 |
| 4 (1,2,2) | 4 | $4.96\times10^4$ | 88% | 3781 | 11.50 |
| 4 (1,2,2) | 8 | $6.14\times10^4$ | 84% | 3781 | 13.67 |
| 4 (1,2,2) | 16 | $6.77\times10^4$ | 83% | 3781 | 14.84 |
| 8 (2,2,2) | 1 | $7.43\times10^4$ | 83% | 3781 | 16.26 |
| 8 (2,2,2) | 2 | $7.14\times10^4$ | 76% | 3781 | 14.35 |
| 8 (2,2,2) | 4 | $6.35\times10^4$ | 84% | 3781 | 14.03 |
| 8 (2,2,2) | 8 | $6.77\times10^4$ | 80% | 3781 | 14.37 |
| 8 (2,2,2) | 16 | $6.35\times10^4$ | 82% | 3781 | 13.79 |

threads. Despite that the large instance has almost four times more particles than the medium instance, the average self-gravity calculation time was only $1.26\times$ 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.

On the large instance, the self-gravity 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 was caused by the up to $10\times$ 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 $10\times$ 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 was not implemented in parallel, so most of the time updating the self-gravity was spent performing sequential operations.

### *Collapsing cube scenario*

This subsection 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.

*Small instance.* Table 4 reports the experimental results 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 $1.50\times10^4$ 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 of time computing self-gravity (71%) was obtained using one process and four threads.

**Table 4.** Performance results of the occupied cells method for the small instance of the collapsing cube scenario.

| *particle processes* | *gravity threads* | *execution time(s)* | *self-gravity time* | *self-gravity updates* | *avg. self-gravity time(s)* |
|---|---|---|---|---|---|
| 1 (1,1,1) | 1 | $3.26\times10^4$ | 82.79% | 11 | 2451.75 |
| 1 (1,1,1) | 2 | $2.90\times10^4$ | 77.60% | 11 | 2046.41 |
| 1 (1,1,1) | 4 | $2.45\times10^4$ | 71.51% | 11 | 1591.73 |
| 1 (1,1,1) | 8 | $2.10\times10^4$ | 65.10% | 11 | 1241.98 |
| 2 (1,1,2) | 2 | $2.24\times10^4$ | 84.86% | 11 | 1725.69 |
| 2 (1,1,2) | 4 | $2.22\times10^4$ | 84.77% | 11 | 1711.92 |
| 2 (1,1,2) | 8 | $1.76\times10^4$ | 80.02% | 11 | 1279.81 |
| 4 (1,2,2) | 1 | $3.34\times10^4$ | 98.65% | 11 | 2998.07 |
| 4 (1,2,2) | 2 | $2.41\times10^4$ | 91.42% | 11 | 2001.9 |
| 4 (1,2,2) | 4 | $2.06\times10^4$ | 81.48% | 11 | 1524.34 |
| 4 (1,2,2) | 8 | $1.50\times10^4$ | 91.10% | 11 | 1238.24 |

In turn, Table 5 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 was about 50%. The lowest value reported was 26.14 s using one process and four threads, and the highest value was 39.28 s using four processes and one thread. However, the lowest execution time was $2.25\times10^3$ 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 indicate that using 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%. In addition, the average self-gravity calculation time of both methods was compared. The lowest value of the average self-gravity calculation time for Barnes & Hut was 26.14 s, two orders of magnitude lower than the lowest value of the occupied cells method (1238.24 s). The self-gravity was 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.

**Table 5.** Performance results of the Barnes & Hut method for the small instance of the collapsing cube scenario.

| *particle processes* | *gravity threads* | *execution time(s)* | *self-gravity time* | *self-gravity updates* | *avg. self-gravity time(s)* |
|---|---|---|---|---|---|
| 1 (1,1,1) | 1 | $8.89\times10^3$ | 4.78% | 13 | 32.65 |
| 1 (1,1,1) | 2 | $8.79\times10^3$ | 4.51% | 13 | 30.51 |
| 1 (1,1,1) | 4 | $7.52\times10^3$ | 4.52% | 13 | 26.14 |
| 1 (1,1,1) | 8 | $7.64\times10^3$ | 4.59% | 13 | 26.99 |
| 2 (1,1,2) | 1 | $4.24\times10^3$ | 8.49% | 13 | 27.68 |
| 2 (1,1,2) | 2 | $7.41\times10^3$ | 5.60% | 13 | 31.90 |
| 2 (1,1,2) | 4 | $7.83\times10^3$ | 5.35% | 13 | 32.21 |
| 2 (1,1,2) | 8 | $6.41\times10^3$ | 6.34% | 13 | 31.24 |
| 4 (1,2,2) | 1 | $5.10\times10^3$ | 10.01% | 13 | 39.28 |
| 4 (1,2,2) | 2 | $3.62\times10^3$ | 11.47% | 13 | 31.90 |
| 4 (1,2,2) | 4 | $3.30\times10^3$ | 12.99% | 13 | 32.93 |
| 4 (1,2,2) | 8 | $3.85\times10^3$ | 11.21% | 13 | 33.18 |
| 8 (2,2,2) | 1 | $2.35\times10^3$ | 19.11% | 13 | 34.54 |
| 8 (2,2,2) | 2 | $2.37\times10^3$ | 18.52% | 13 | 33.80 |
| 8 (2,2,2) | 4 | $2.39\times10^3$ | 18.31% | 13 | 33.69 |
| 8 (2,2,2) | 8 | $2.25\times10^3$ | 16.58% | 13 | 28.65 |

*Medium instance.* Table 6 reports the experimental results of the Barnes & Hut method for the medium instance of the collapsing cube scenario. 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 $1.84\times10^4$ s using 27 processes and 16 threads. Results for the occupied cells method showed that the of a single self-gravity update was $2.08\times10^5$ s. So, the self-gravity calculation time in average for the Barnes & Hut method is three orders of magnitude smaller than the result reported for the occupied cells method. For this reason, the complete performance study for the occupied cells method was not performed in the context of this research.

**Table 6.** Performance results of the Barnes & Hut method for the large instance of the collapsing cube scenario.

| *particle processes* | *gravity threads* | *execution time(s)* | *self-gravity time* | *self-gravity updates* | *avg. self-gravity time(s)* |
|---|---|---|---|---|---|
| 1 (1,1,1) | 1 | $8.01\times10^4$ | 8.24% | 21 | 314.69 |
| 1 (1,1,1) | 2 | $7.38\times10^4$ | 7.69% | 21 | 270.38 |
| 1 (1,1,1) | 4 | $1.04\times10^5$ | 11.66% | 21 | 575.64 |
| 1 (1,1,1) | 8 | $7.89\times10^4$ | 9.71% | 21 | 364.81 |
| 1 (1,1,1) | 16 | $6.79\times10^4$ | 10.41% | 21 | 336.64 |
| 8 (2,2,2) | 2 | $3.56\times10^4$ | 20.78% | 21 | 352.40 |
| 8 (2,2,2) | 4 | $1.85\times10^4$ | 28.50% | 21 | 251.06 |
| 8 (2,2,2) | 8 | $2.80\times10^4$ | 35.42% | 21 | 472.47 |
| 8 (2,2,2) | 16 | $2.50\times10^4$ | 27.53% | 21 | 327.87 |
| 27 (3,3,3) | 16 | $1.84\times10^4$ | 36.34% | 21 | 320.09 |

*Large instance.* Table 7 reports the results of the Barnes & Hut method for the large instance of the collapsing cube scenario. 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 was $1.5\times10^6$ s, while the estimated execution time using the occupied cells method was $9.0\times10^7$ 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 $1.25\times10^5$ s using 36 processes and 16 threads. The lowest average self-gravity calculation time was 313.14 s.

**Table 7.** Performance results for the Barnes & Hut method on the large instance of the collapsing cube scenario (2,097,152 particles).

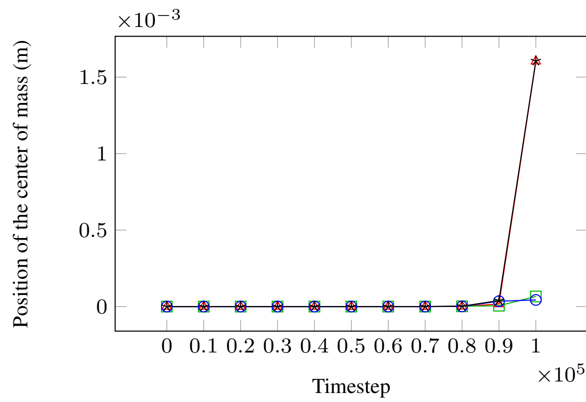
| *particle processes* | *gravity threads* | *execution time(s)* | *self-gravity time* | *self-gravity updates* | *avg. self-gravity time(s)* |
|---|---|---|---|---|---|
| 8 (2,2,2) | 8 | $1.43\times10^5$ | 20.21% | 42 | 689.01 |
| 8 (2,2,2) | 16 | $1.95\times10^5$ | 6.60% | 41 | 313.14 |
| 27 (3,3,3) | 8 | $1.57\times10^5$ | 8.65% | 41 | 331.41 |
| 36 (3,3,4) | 16 | $1.25\times10^5$ | 11.09% | 41 | 336.72 |

*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 execution 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 was 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**$\times$ for the small instance, and was more than **100**$\times$ 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**$\times$ 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.

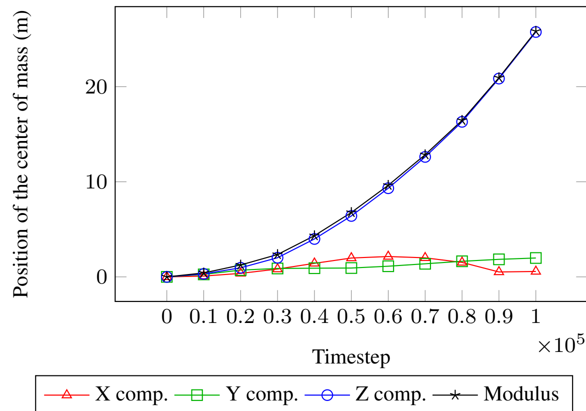
## Numerical accuracy: analysis of the position of the center of mass

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

*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

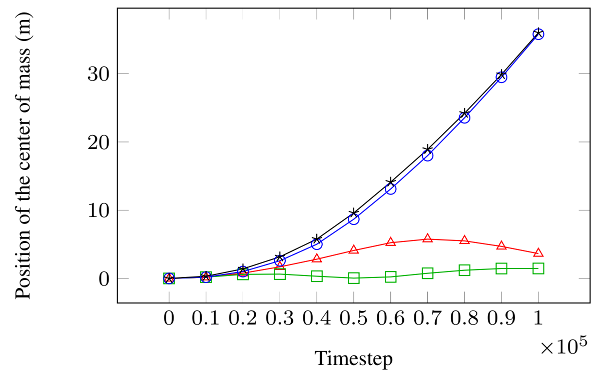



**(a)** Occupied cells method.

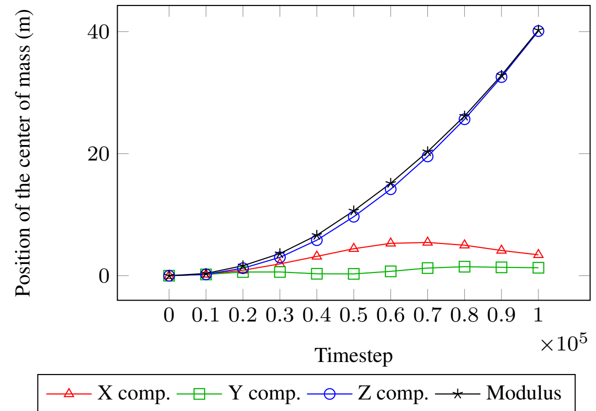



**(b)** Barnes & Hut method.

**Figure 7.** Position of the center of mass over time for the small instance of the two agglomerates scenario.




**(a)** Medium scenario.




**(b)** Large scenario.

**Figure 8.** Position of the center of mass over time for the medium and large instances of the two agglomerates scenario using the Barnes & Hut method.

simulation. The less the center of mass moves during the simulation, the more accurate to the reality represented. In an ideal scenario, 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.

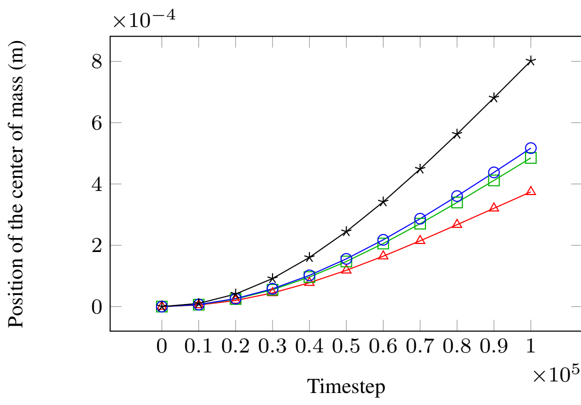
Regarding the figures presented in this subsection, four sets of data are reported in each figure: red, green, and blue lines represent the evolution of the X, Y, and Z components of the position of the center of mass for a simulation, respectively. The variation of the modulus of the position of the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

*Results for the two agglomerate scenario* Figure 7 shows the variation of the position of the center of mass for the small instance of the two agglomerates scenario using the occupied cells method (Figure 7a) the Barnes & Hut method (Figure 7b).

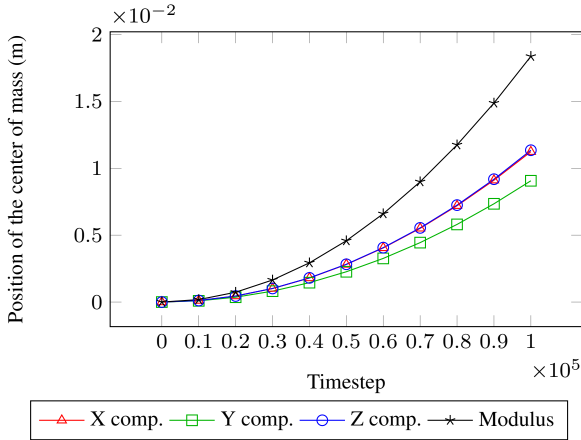
Results show that the position of the center of mass varies in the order of $1 \times 10^{-3}$ m when using the occupied cells method and in the order of $1 \times 10^{1}$ m when using the Barnes & Hut method. So, a loss of precision of the calculation of the self-gravity is perceived when the Barnes & 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. Rounding in 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 8 shows the position of the center of mass for the medium (Figure 8a) and large (Figure 8b) 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.

*Results for the collapsing cube scenario* Figure 9 shows the variation of the position over time of the center of mass for the small cube scenario for the occupied cells method (Figure 9a) and for the Barnes & Hut method (Figure 9b). Results indicate that using either algorithm, the modulus of the center of mass increases over time.

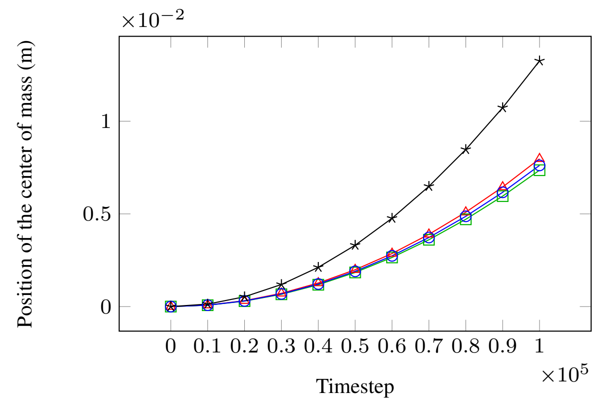



**(a)** Occupied cells method.

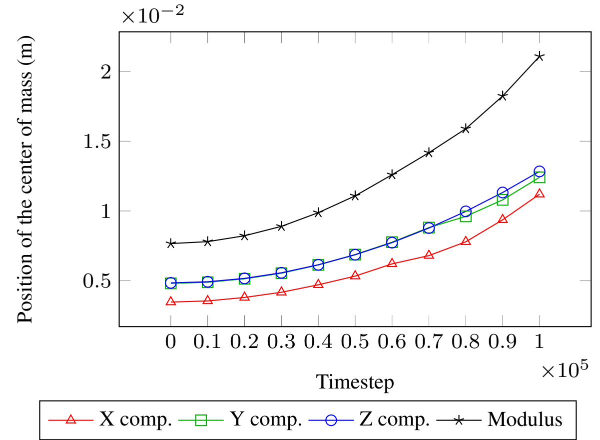



**(b)** Barnes & Hut method.

**Figure 9.** Position of the center of mass over time for the small instance of the cube scenario.




**(a)** Medium cube scenario.




**(b)** Large cube scenario.

**Figure 10.** Position of the center of mass over time for the cube scenario using the Barnes & Hut method.

For the small instance, the position of the center of mass varied in the order of $10^{-4}$ m for the occupied cells method, and in the order of $10^{-2}$ m for Barnes & Hut, confirming that the occupied cells method is more precise than Barnes & Hut. However, the precision is higher for the collapsing cube scenario rather than for the two agglomerates scenario, suggesting that Barnes & Hut performs better in scenarios with slow particles movements.

Figure 10 shows the position of the center of mass for the medium and large scenarios using Barnes & Hut. The maximum value for the position of the center of mass was 36.0 m (medium scenario) and 40.3 m (large scenario), both registered at timestep $1\times10^5$. 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 than 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. The reason of this behavior is that the loss in precision is introduced mainly by the calculation of the self-gravity, which is based on the nodes and boxes rather than on the particles.

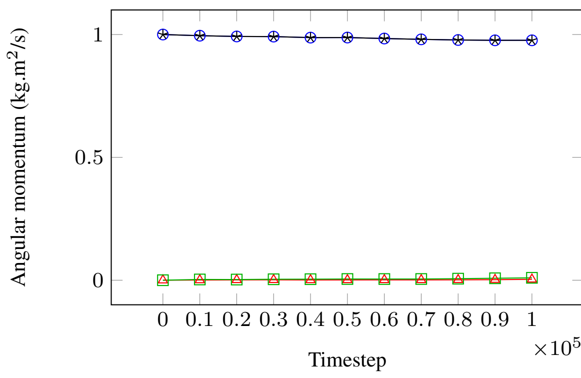
## *Numerical accuracy: analysis of the angular momentum*

This section studies the conservation of the angular momentum for both studied scenarios using a neighborhood of size three. Results obtained are reported, commented, and discussed next.

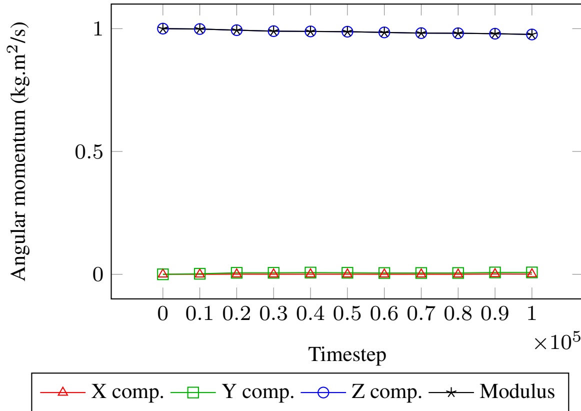
*Angular momentum and its relevance.* The angular momentum is used to calculate and describe the rotational momentum of group of particles. The angular momentum is important because it has the property that it is conserved with respect to a point and it is not affected by external forces. The angular momentum of a point particle with respect to the origin (measured in $kg\frac{m^2}{s}$) is $L = I\omega$, where $I$ represents the moment of inertia for the particle, and $\omega$ is the angular velocity of the particle with respect to the origin. In the analysis, the modulus of the angular momentum vector is calculated, and the variation is studied and commented.

*Results for the two agglomerates scenario.* Figure 11 shows the shows the variation of the angular momentum over time for the small instance of the two agglomerates scenario using occupied cells and Barnes & Hut algorithms. The value for the angular momentum is $6.13\times10^{18}$ kg.m$^2$/s in the last timestep, using either method ($1\times10^5$). 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 12 shows the variation of the angular momentum over time for the medium and large instances of the two agglomerates scenario using the Barnes & Hut method. The initial value for the modulus of the angular momentum is $6.20\times10^{18}$ kg.m$^2$/s and the value at the end of the simulation is $6.03\times10^{18}$ kg.m$^2$/s. The difference between the initial and the final value is similar to the difference for the small scenario using both methods ($0.14\times10^{18}$ kg.m$^2$/s).

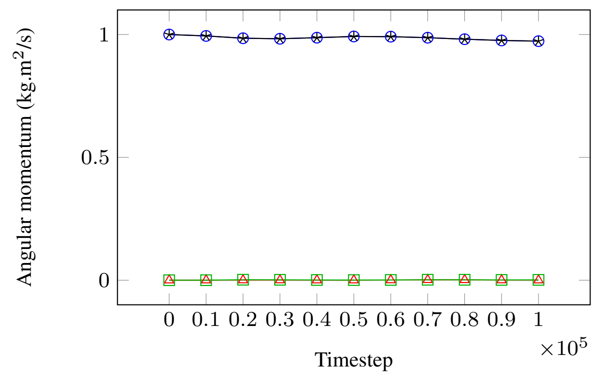



**(a)** Occupied cells method.

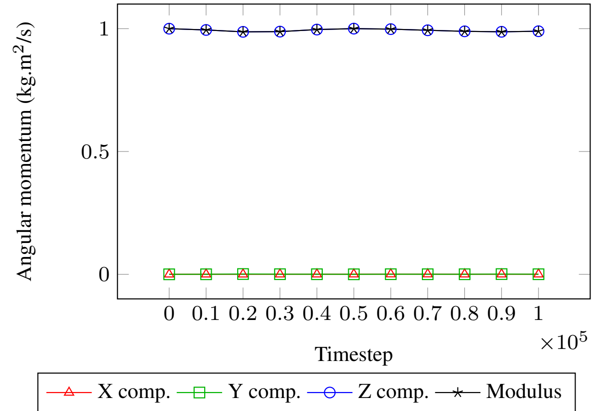



**(b)** Barnes & Hut method.

**Figure 11.** Angular momentum over time for the small instance of the two agglomerates scenario.




**(a)** Medium scenario.




**(b)** Large scenario.

**Figure 12.** Angular momentum over time for the two agglomerates scenario using the Barnes & Hut method.

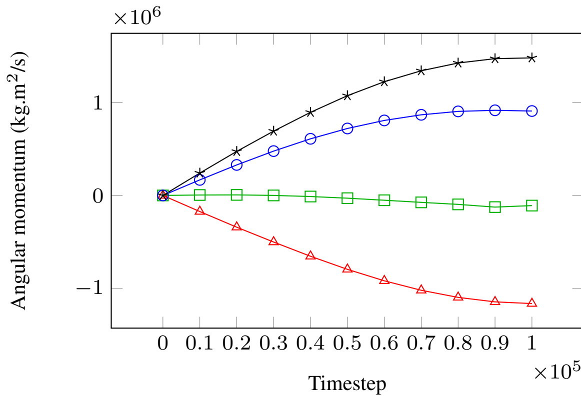
Results obtained for the large instance using the Barnes & Hut method were similar to the ones obtained for the medium instance. Values of the angular momentum modulus oscillated between $8.42\times10^{18}$ kg.m$^2$/s and $8.33\times10^{18}$ kg.m$^2$/s, (the difference of the angular momentum modulus was $0.09\times10^{18}$ kg.m$^2$/s). 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 Barnes & Hut and the occupied cells method.

*Results for the collapsing cube scenario.* Figure 13 shows the variation of the angular momentum for the small instance of the collapsing cube scenario using the occupied cells and the Barnes & Hut methods. 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 increased over timesteps. The maximum value obtained for the small instance using the occupied cells method was $1.48\times10^{6}$ kg.m$^2$/s, while using Barnes & Hut was $1.97\times10^{8}$ kg.m$^2$/s, two orders of magnitude larger. In addition, the increase of the modulus is different for both methods: for the occupied cells method the speed of the increase slowed down as the time steps increase, while the Barnes & Hut method showed an steady increase of the modulus throughout the simulation. In spite of the difference in precision, the acceleration using the Barnes & Hut method was calculated $47.37\times$ faster than using the occupied cells method.

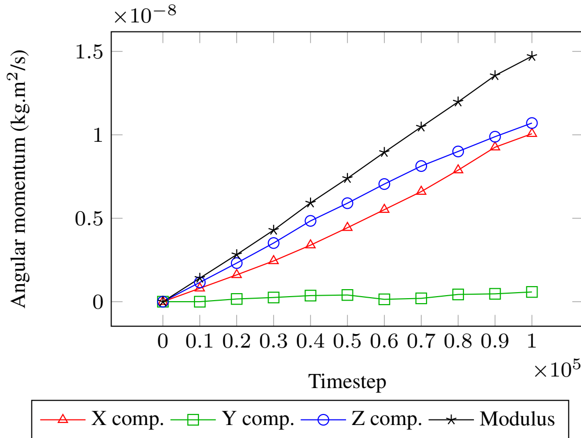
Figure 14 graphically shows the results obtained for the study of the evolution of the angular momentum over time for the medium and large instances of the collapsing cube using the Barnes & Hut method. The results presented in Figure 14a show that the modulus of the angular momentum also increases for 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 $8.52\times10^{6}$ kg.m$^2$/s.

Figure 14b graphically 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 $6.40\times10^{6}$ 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.

Overall, the results obtained for the study of the angular momentum on the collapsing cube scenario suggest that as the number of particles 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, the angular momentum increased over time for all the studied instances. 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.

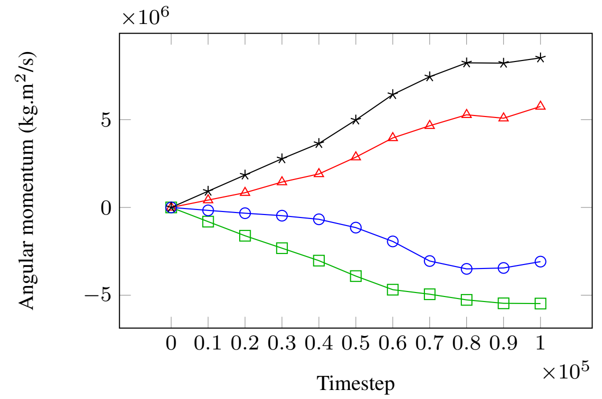



**(a)** Occupied cells method.

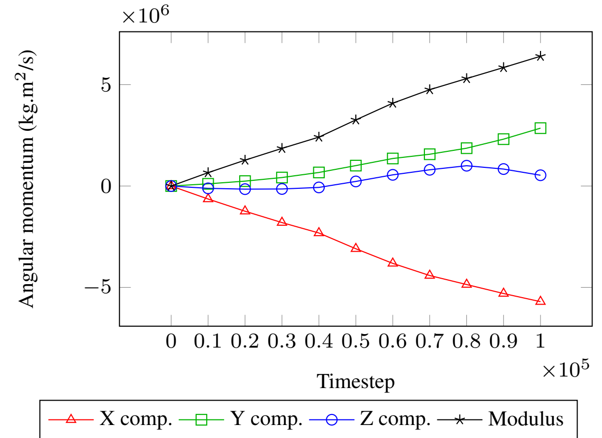



**(b)** Barnes & Hut method.

**Figure 13.** Angular momentum over time for the small instance of the collapsing cube scenario.




**(a)** Medium scenario.




**(b)** Large scenario.

**Figure 14.** Angular momentum over time for the collapsing scenario using the Barnes & Hut method.

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

This section reports results of the binary tree algorithm for the two agglomerates scenario and a performance comparison against the Barnes & Hut octal tree. In addition, a study of the performance of a variation of the octal tree is presented, computing the center of mass using particles of the system at upper tree levels rather than in the lower levels.

### *Performance results of the binary tree algorithm*

The performance of the binary tree algorithm was studied over the two agglomerates scenario. All results correspond to the average of five executions for each configuration of processes and threads.

Table 8 reports the total execution time and the average time of a self-gravity update for the octal tree and the binary tree algorithms for the small instance of the two agglomerates scenario. 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 $2\times$ 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 9 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 8.** Performance results for the small instance of the two agglomerate scenario.

| | | *octal tree* | | *binary tree* | |
|---|---|---|---|---|---|
| *particle processes* | *#gravity threads* | *execution time(s)* | *self-gravity time(s)* | *execution time(s)* | *self-gravity time(s)* |
| 1 (1,1,1) | 1 | $9.15\times10^2$ | 10.11 | $1.30\times10^3$ | 14.55 |
| 1 (1,1,1) | 2 | $6.64\times10^2$ | 6.99 | $1.03\times10^3$ | 11.65 |
| 2 (1,1,2) | 1 | $9.59\times10^2$ | 10.58 | $1.76\times10^3$ | 18.78 |
| 2 (1,1,2) | 2 | $7.08\times10^2$ | 7.40 | $1.47\times10^3$ | 15.18 |

**Table 9.** Performance results for the medium instances of the two agglomerate scenario using Barnes & Hut.

| | | octal tree | | binary tree | |
|---|---|---|---|---|---|
| *particle processes* | *#gravity threads* | *execution time(s)* | *self-gravity time(s)* | *execution time(s)* | *self-gravity time(s)* |
| 1 (1,1,1) | 1 | $6.87\times10^3$ | 51.37 | $8.02\times10^3$ | 59.55 |
| 1 (1,1,1) | 2 | $4.75\times10^3$ | 31.22 | $6.32\times10^3$ | 46.41 |
| 1 (1,1,1) | 4 | $4.30\times10^3$ | 31.09 | $5.27\times10^3$ | 38.19 |
| 2 (1,1,2) | 1 | $7.14\times10^3$ | 54.23 | $9.76\times10^3$ | 72.70 |
| 2 (1,1,2) | 2 | $4.57\times10^3$ | 33.85 | $7.31\times10^3$ | 53.70 |
| 2 (1,1,2) | 4 | $4.10\times10^3$ | 30.35 | $5.70\times10^3$ | 41.26 |

The large instance was studied using 20 different configurations of processes and threads. The gravitational potential was updated 264 times for both algorithms. Table 10 reports the results obtained for each studied configuration.

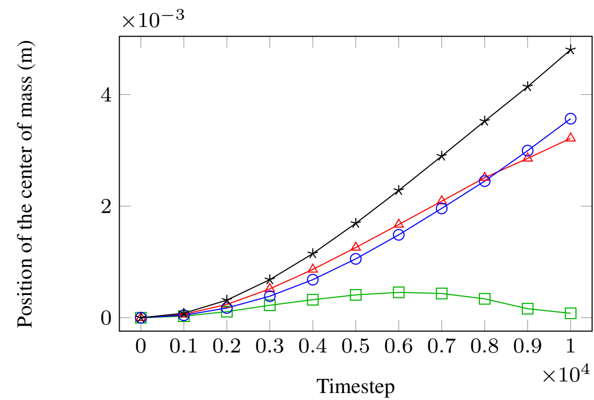
**Table 10.** Performance results for the two agglomerate scenario with 38,358 particles (large instance).

| | | octal tree | | binary tree | |
|---|---|---|---|---|---|
| *particle processes* | *#gravity threads* | *execution time(s)* | *self-gravity time(s)* | *execution time(s)* | *self-gravity time(s)* |
| 1 (1,1,1) | 1 | $1.49\times10^4$ | 49.79 | $1.89\times10^4$ | 64.40 |
| 1 (1,1,1) | 2 | $1.04\times10^4$ | 32.86 | $1.34\times10^4$ | 42.94 |
| 1 (1,1,1) | 4 | $9.21\times10^3$ | 28.08 | $1.10\times10^4$ | 35.38 |
| 1 (1,1,1) | 8 | $9.59\times10^3$ | 29.60 | $1.10\times10^4$ | 35.37 |
| 1 (1,1,1) | 16 | $1.09\times10^4$ | 34.81 | $1.16\times10^4$ | 36.75 |
| 2 (1,1,2) | 1 | $1.43\times10^4$ | 49.58 | $1.90\times10^4$ | 65.97 |
| 2 (1,1,2) | 2 | $1.07\times10^4$ | 35.54 | $1.27\times10^4$ | 42.79 |
| 2 (1,1,2) | 4 | $1.01\times10^4$ | 32.62 | $1.10\times10^4$ | 35.81 |
| 2 (1,1,2) | 8 | $1.09\times10^4$ | 35.79 | $1.02\times10^4$ | 33.92 |
| 2 (1,1,2) | 16 | $1.06\times10^4$ | 34.95 | $1.12\times10^4$ | 36.32 |
| 4 (1,2,2) | 1 | $1.62\times10^4$ | 57.63 | $1.88\times10^4$ | 65.91 |
| 4 (1,2,2) | 2 | $1.07\times10^4$ | 36.56 | $1.49\times10^4$ | 52.46 |
| 4 (1,2,2) | 4 | $9.56\times10^3$ | 32.27 | $9.72\times10^3$ | 32.64 |
| 4 (1,2,2) | 8 | $1.04\times10^4$ | 35.07 | $9.82\times10^3$ | 33.09 |
| 4 (1,2,2) | 16 | $1.07\times10^4$ | 36.20 | $1.09\times10^4$ | 37.28 |
| 8 (2,2,2) | 1 | $1.65\times10^4$ | 60.27 | $1.74\times10^4$ | 62.80 |
| 8 (2,2,2) | 2 | $1.12\times10^4$ | 39.78 | $1.11\times10^4$ | 39.23 |
| 8 (2,2,2) | 4 | $1.03\times10^4$ | 36.72 | $8.83\times10^3$ | 30.94 |
| 8 (2,2,2) | 8 | $9.69\times10^3$ | 34.29 | $9.58\times10^3$ | 33.86 |
| 8 (2,2,2) | 16 | $1.02\times10^4$ | 36.38 | $1.06\times10^4$ | 37.10 |

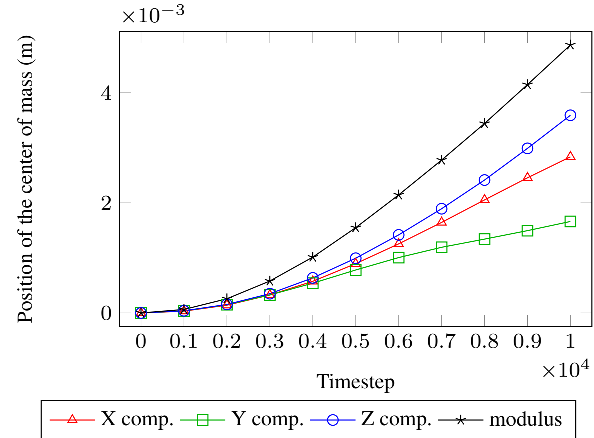
For the large instance, the binary tree with eight processes and four threads had the best execution time. This result supports the rule of thumb. Also, for the same number of processes, the configurations using 8 or 16 threads performed slower than the configurations using 4 threads. Results obtained suggest that the binary tree algorithm performs faster than the octal tree for large instances.

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

Figure 15 shows the position of the center of mass ($x$, $y$, $z$ components, and its module) and its variation over time for the small instance of the two agglomerates scenario for the octal tree, and the binary tree methods. 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 Figures 16 and 17.




**(a)** Octal tree algorithm.




**(b)** Binary tree algorithm.

**Figure 15.** 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.

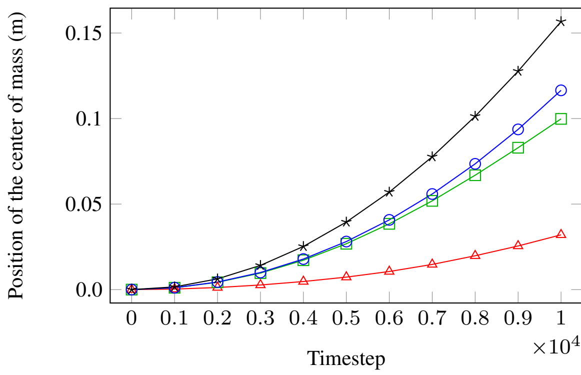
Results support the commented trends for the small instance. In addition to the differences in accuracy, differences in the position of the components of the center of mass were detected when using the different tree structures. An example is shown in Figure 15: the position of the center of mass when 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 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.

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

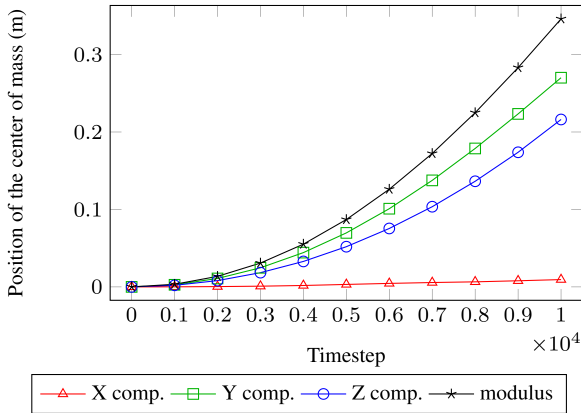
Figure 18 reports the results of the study of the angular momentum for the two agglomerates scenario using the octal tree and the binary tree for the small instance. Then, Figure 19 reports the results for the medium instance. Finally, Figure 20 report the results for the large instance. Results indicate that the angular momentum is of the same order of magnitude either using the octal tree or the binary tree.

## Upper level direct calculation of the mass center

This section reports the experimental results of the upper level direct calculation of the mass center. The results presented include the study of the execution time and the movement of the center of mass of the system.

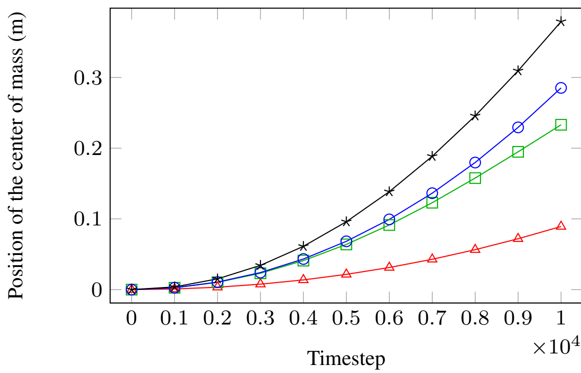



**(a)** Octal tree algorithm.

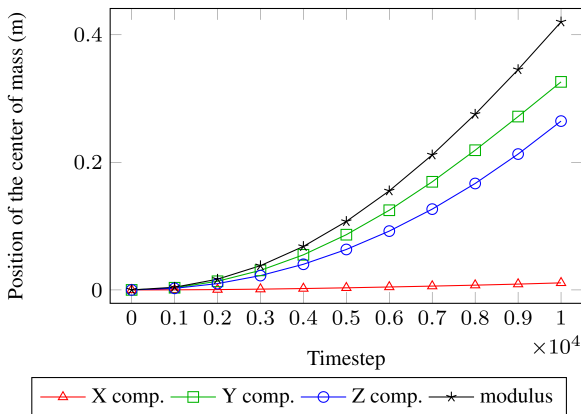



**(b)** Binary tree algorithm.

**Figure 16.** 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.

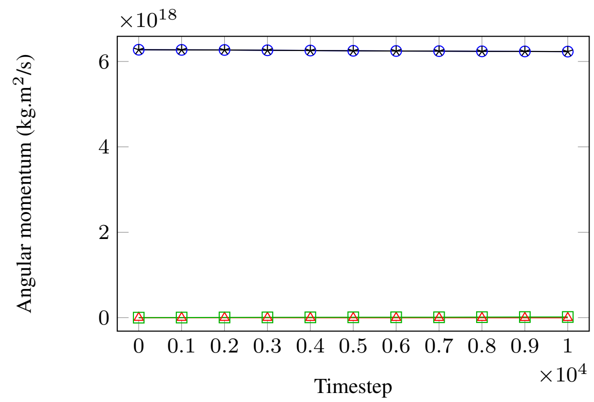



**(a)** Octal tree algorithm.

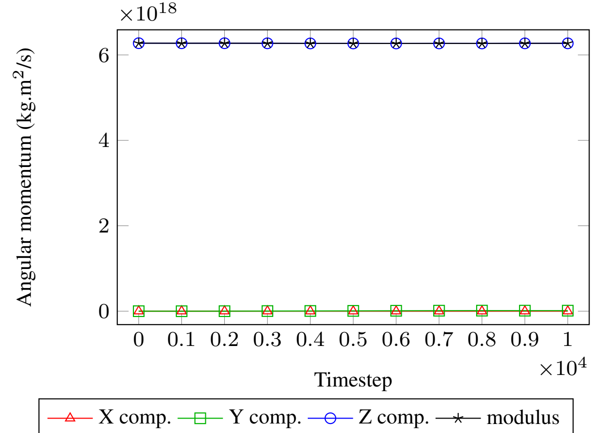



**(b)** Binary tree algorithm.

**Figure 17.** 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.




**(a)** Octal tree algorithm.



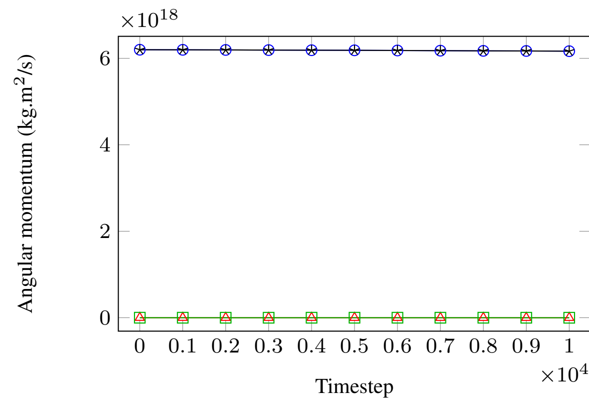

**(b)** Binary tree algorithm.

**Figure 18.** Position of the center of mass over time for the small instance of the two agglomerates scenario using the Barnes & Hut method.

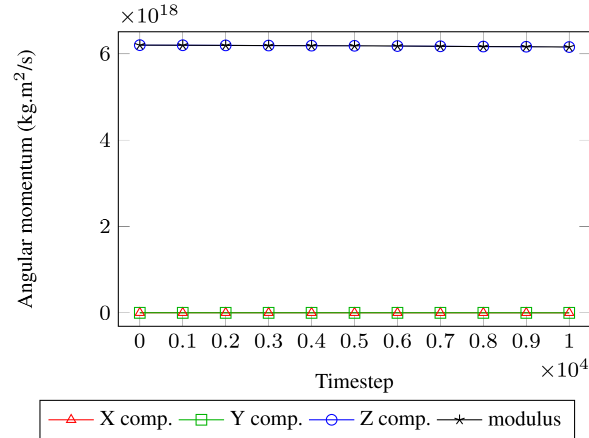
The aim of the study is to prove that the precision of the calculation of the potential increases by calculating the center of mass in upper levels using particles. So, a study of the position of the center of mass was performed. The large instance of the two agglomerates scenario was considered. The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

Table 11 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 21 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.

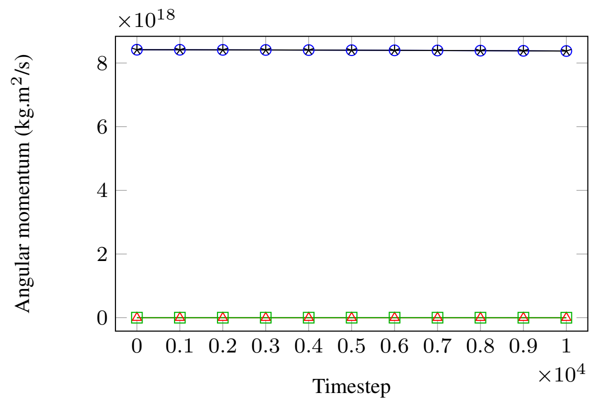


(a) Octal tree algorithm.

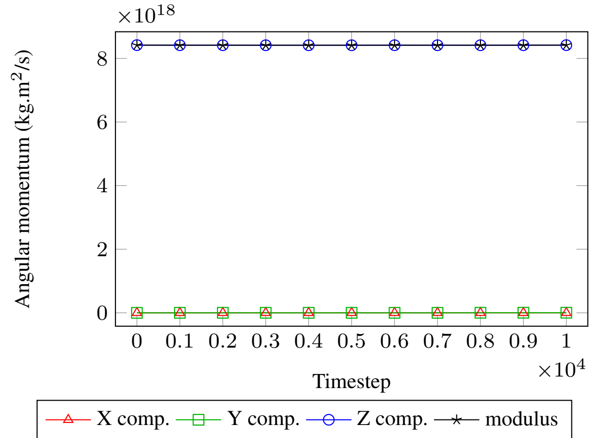


(b) Binary tree algorithm.

**Figure 19.** 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.



(a) Octal tree algorithm.



(b) Binary tree algorithm.

**Figure 20.** Position of the center of mass over time for the large instance of the two agglomerates scenario using the Barnes & Hut method with the binary tree algorithm.

Results of the study of the center of mass for the octal tree on Figure 17a show that the center of mass of the system, for the scenario studied, moved $3.79\times10^{-1}$ m on time step $10,000$. On the other hand, using the upper level direct calculation, the center of mass moved $7.94\times10^{-4}$ m on time step $10,000$. This way, results show that by applying the strategy presented in this section, the precision of the calculation improves up to three levels of magnitude.

**Table 11.** Performance results for the upper level direct calculation of the mass center on the large instance of the two agglomerate scenario.

| *neighb. size* | *execution time(s)* | *self-gravity time* | *self-gravity updates* | *avg. self-gravity time(s)* |
|---|---|---|---|---|
| 0 | $1.80\times10^3$ | $1.28\times10^3$ | 264 | 4.86 |
| 1 | $1.73\times10^3$ | $1.16\times10^3$ | 264 | 4.39 |
| 2 | $1.96\times10^3$ | $1.44\times10^3$ | 263 | 5.49 |
| 3 | $2.99\times10^3$ | $2.16\times10^3$ | 263 | 8.21 |
| 4 | $3.00\times10^3$ | $2.50\times10^3$ | 263 | 9.49 |

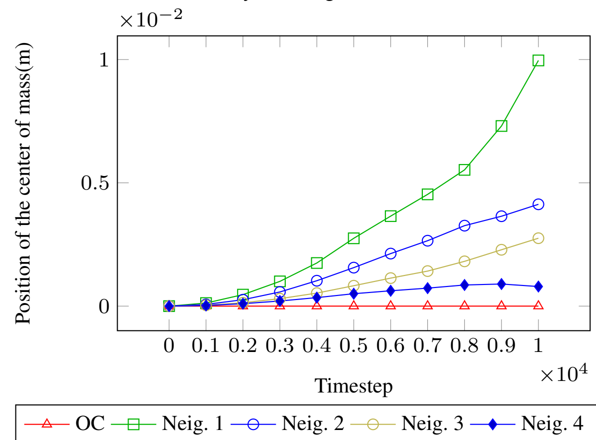


**Figure 21.** Position of the center of mass over time for the large instance of the two agglomerates scenario using the upper level direct calculation of the mass center.

## Conclusions

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

The occupied cells method consists of updating the acceleration of the occupied nodes of an overlying grid. A profiling analysis was performed to identify bottlenecks in the implementation and specific modifications were included to reduce the number of invocations of time consuming routines. Two variants of the Barnes & Hut method were proposed, using octal and binary trees for partitioning the domain. 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. A specific variation was also proposed, 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 $10\times$ faster than the occupied cells method in simulations to compute self-gravity for the two agglomerates scenario and up to $100\times$ 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 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 problem instances and scenarios, and proposing more strategies to improve the precision of the calculations over a simulation.

## References


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.

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

Couchman HMP (1991) Mesh-refined p3m-a fast adaptive n-body algorithm. *The Astrophysical Journal* 368: L23–L26.

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 et al DS (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 RW and Eastwood JW (1988) *Computer simulation using particles*. crc Press.

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*. IEEE Computer Society Press, p. 5.

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

Kravtsov AV, Klypin AA, 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*. Springer, pp. 311–325.

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

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*. Springer, pp. 291–306.

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*. Springer, pp. 141–156.

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 Nesmachnow S (2012) Granular physics in low-gravity environments using discrete element method. *Mnras* 420: 3368–3380. DOI: 10.1111/j.1365-2966.2011.20259.x.

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

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