Proceedings of the 2019 Winter Simulation Conference
N. Mustafee, K.-H.G. Bae, S. Lazarova-Molnar, M. Rabe, C. Szabo, P. Haas, and Y.-J. Son, eds.

LARGE-SCALE MULTITHREADING SELF-GRAVITY SIMULATIONS FOR ASTRONOMICAL AGGLOMERATES

Sergio Nesmachnow
Néstor Rocchetti

Facultad de Ingeniería
Universidad de la República
Herrera y Reissig 565
11300 Montevideo, Uruguay

Gonzalo Tancredi

Facultad de Ciencias
Universidad de la República
Iguá 4225
11400 Montevideo, Uruguay

ABSTRACT

This article presents parallel multithreading self-gravity simulations for astronomical agglomerates, applying High Performance Computing techniques to allow the efficient simulation of systems with a large number of particles. Considering the time scales needed to properly simulate the processes involved in the problem, two parallel mesh-based algorithms to speed up the self-gravity calculation are proposed: a method that updates the occupied cells of the mesh, and a method to divide the domain based on the Barnes-Hut tree. Results of the experimental evaluation performed over a scenario with two agglomerates orbiting each other indicate that the Barnes-Hut allows accelerating the execution times over $10\times$ compared to the occupied cells method. These performance improvements allow scaling up to perform realistic simulations with a large number of particles (i.e., tens of millions) in reasonable execution times.

1 INTRODUCTION

Some astronomical objects, like asteroids and comets, are agglomerates of smaller particles called grains, which are kept together by their mutual gravitational force. Grains are affected by short range interactions, e.g., contact forces, and long range interactions. Long range interactions are a combination of the effect of the influence of the gravity of other objects and the effect of the influence of the other grains that conform the agglomerate itself. The latter is called *self-gravity* of the agglomerate (Harris et al. 2009). 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 the interactions between particles, agglomerates are studied using computational simulations. Using a straightforward approach, interactions between every pair of particles in the agglomerate must be calculated. Thus, the computational cost of performing one step of the simulation is $O(N^2)$, being N the number of particles. This way, performing simulations of millions of particles, as it is usual to model medium-size astronomical objects, is computationally expensive.

High Performance Computing (Foster 1995) is a computational paradigm that helps researchers to solve complex problems and perform large-scale simulations on big domains. This paradigm allows dealing with complicated problems that demand high computer power, in significantly shorter execution times than using a traditional (sequential) computing approach. Instead of using a single computing resource, High Performance Computing proposes using multiple resources simultaneously, applying a coordinated approach. This way, a cooperation strategy is implemented to solve subproblems, allowing the workload to be divided between the computational units available to solve a complex/big problem in reasonable execution times (Hager and Wellein 2010).

Discrete Element Method (DEM) (Cundall and Strack 1979) is a numerical method used for simulating physical phenomena. DEM is used for computing the motion and effect of a large number of small particles, i.e., it was originally created to calculate short range interactions between particles during a simulation. The performance of DEM can be improved by the application of High Performance Computing techniques. The DEM method is implemented in many software package, among them in ESyS-Particle (Abe, Altinay, Boros, Hancock, Latham, Mora, Place, Petterson, Wang, and Weatherley 2009). ESyS-Particle was originally developed to simulate geological phenomena, but it can also be applied in many other research areas.

The first applications of ESyS-Particle in astronomy and planetary sciences was presented by our research group (Tancredi, Maciel, Heredia, Richeri, and Nesmachnow 2012). That work included simulations in low-gravity environments (e.g., asteroids and comets) and proposed new models to simulate contact forces. We identified a specific shortcoming of ESyS-Particle, as well as other well-known DEM software packages: the lack of models to simulate long-range forces. Self-gravity is one of those long range forces, which is very relevant in the study of astronomical objects, as it can cause attraction between particles and deformation (Walsh, Richardson, and Michel 2012). Our previous work (Frascarelli, Nesmachnow, and Tancredi 2014) proposed a specific module for self-gravity calculation on agglomerates, applying High Performance Computing 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. Later (Nesmachnow, Frascarelli, and Tancredi 2015), several strategies were described for designing an efficient and accurate parallel algorithm using multiple threads (i.e., multithreading parallelism) for self-gravity computation in the simulation of astronomical agglomerates.

In this line of work, this article presents large-scale self-gravity simulations of astronomical bodies using parallel multithreading algorithms. The simulations include 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 an octal tree to speed up long range forces calculation. Both methods are evaluated and compared over a scenario that consists in two agglomerates orbiting each other. The experimental evaluation comprises the performance analysis of the self-gravity simulations using the two methods, including a comparison of the results obtained when scaling the number of computational resources to simulate instances with different number of particles. Results show that the proposed Barnes-Hut method allows improving the performance of the self-gravity calculation up to $10\times$ with respect to the occupied cell method. This way, efficient simulations are performed for large problem instances.

The article is organized as follows. Section 2 describes works applying domain decomposition techniques and introduces the parallel self-gravity calculation algorithm for astronomical simulations. Section 3 describes the general approach for self-gravity calculation in ESyS-Particle and Section 4 introduces the occupied cells method to improve the efficiency of self-gravity simulations. The proposed Barnes-Hut method for self-gravity simulations is described in Section 5. Section 6 reports the experimental analysis focusing on the computational efficiency of the proposed implementation. Finally, the conclusions are presented in Section 7.

2 SIMULATIONS APPLYING SPATIAL DOMAIN DECOMPOSITION AND PARALLEL SELF-GRAVITY COMPUTATION

This section introduces spatial domain decomposition techniques, which is the paradigm applied in the simulations performed in the study, and the proposed parallel self-gravity implementation. Domain decomposition techniques are used to speed up the calculation of interactions between particles in order to reduce the $O(N^2)$ complexity order of a straightforward particle-by-particle algorithm.

2.1 Simulations Using Static Hierarchical Domain Decomposition

Static techniques for particle interaction apply a domain decomposition that stays invariable during a simulation. Static techniques are divided in Particle-Particle (PP) methods, Particle-Mesh (PM) methods, and Particle-Particle Particle-Mesh (P3M) methods. PP are the simplest methods and consist of computing the forces directly between all the particles in the system. PP methods are the most accurate, but lack the ability to scale in the number of particles due to its $O(N^2)$ execution time. PM methods use a mesh over the simulated space and calculate the potential for the point particle that belongs to the mesh. After that, the speed for the particles is calculated via interpolation. PM methods are less accurate than PP methods, but they are faster. However, in practical applications PM may need a mesh resolution that makes the method not efficient for computing contact forces. Finally, P3M methods divide the forces into short range and long range. Short range forces are calculated using a PP method, and the long range forces are calculated using a PM method. The combination results in a fast and accurate method of simulating particle interaction.

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

Kravtsov, Klypin, and Khokhlov (1997) proposed an N-body solver based on the PM method over a multilevel grid to perform the force calculations on astronomical systems. The solver creates a mesh over a cubic space that is divided by regular cells with cubic shape and a predefined size. Depending on the particle density, the elements of the mesh can spawn eight sons that have the same size. Couchman (1991) presented a P3M algorithm that implements a selective refinement of the grid depending on the particle density of a cell. The refinement is performed recursively while a density threshold is exceeded. The particle mesh is an spatial division; thus, no computations are needed for overlapping zones.

2.2 Simulations Using Dynamic Hierarchical Domain Decomposition

Dynamic techniques for particle interaction use structures adapted or reconstructed from scratch during the simulation. A classic dynamic technique is the method by Barnes and Hut (1986), which is the method with the best time complexity. The simulated domain is divided in a hierarchical octal tree to accelerate the calculation of the gravitational potential in a N-body simulation. The tree is composed of nodes that represent a portion of the simulation space and the root node represents the complete space of the simulation. If the space represented by a node contains more than one particle, the space is divided into smaller pieces. In a two-dimension simulation, the space is divided in four smaller pieces. However, in a 3D simulation, the space is divided into eight pieces. Applying this domain decomposition, the Barnes and Hut allows computing the long-range interactions in $O(N \log N)$, being N the number of particles in the domain.

The Fast Multipole Method by Greengard and Rokhlin (1987) applies a multipole expansion of the simulation space, organized as a hierarchy of meshes rather than a tree. The reported efficiency results showed that the proposed method significantly improved the computational efficiency of a direct self-gravity calculation: it was up to $300\times$ faster in a problem considering an agglomerate with 12,800 particles.

2.3 Astronomical Simulations Using a Parallel Algorithm for Self-Gravity Calculation

In astronomical simulations, the gravitational potential of a particle must be computed on each timestep. The gravitational interaction between particles i and j is computed using Equation (1), where G is the gravitational constant, $\overrightarrow{r}$ is the position vector, M_i is the mass of particle i, and V_j is the gravitational potential of particle j (Frascarelli, Nesmachnow, and Tancredi 2014).

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

An implementation of the gravitational potential calculation that applies Equation (1) to every pair of particles has a computational cost of $O(N^2)$ in each time step. This approach does not scale efficiently when performing simulations over scenarios with a large number (e.g., hundreds of thousands) of particles.

Our previous work (Frascarelli, Nesmachnow, and Tancredi 2014) proposed a multi threading algorithm for self-gravitating computation on agglomerates to improve the scalability and the computational efficiency for large-scale simulations. A hierarchical grouping approximation method (*Mass Approximation Distance Algorithm*, MADA) was introduced. The main goal of MADA is to improve the performance of the gravitational potential calculation of a particle in a given timestep, by considering a group of distant particles as a single big particle located in the center of mass of the group. The proposed parallel algorithm was able to efficiently calculate the self-gravity of objects with more than one million particles using MADA and a pool of worker threads that execute the most compute-intensive tasks in parallel sections, following a P3M approach. Several computation and data-assignment patterns were proposed and studied to determine the best efficiency and scalability properties of the proposed self-gravity computation method (Nesmachnow, Frascarelli, and Tancredi 2015). Experimental results demonstrated that the best implementation of the algorithm among the presented in the paper is the advanced isolated linear strategy. The strategy consists of two stages: the first stage is assigning sets of cells for each thread. After a thread finishes processing its workload, it starts looking for unprocessed cells and performing the calculations. The advanced isolated linear strategy showed that it is able to scale up linearly with the number of particles in the system, and it scales with an inverse power law (exponent 0.87) with the number of threads used in the computation. The observed speedup was close to linear for systems containing up to 2×10^5 particles.

3 SELF-GRAVITY ON ESYS-PARTICLE

DEM simulations in ESyS-Particle consider two types of forces: long range forces, e.g., gravitational forces that every particle exerts to each other, and short range forces, which are the result of the contact between particles. 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 more than a certain threshold. When particles move faster, self-gravity needs to be updated more frequently than when particles move slower. On the other hand, contact forces are updated in 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 computation scheme in the self-gravity module implemented into ESyS-Particle applies four phases:

1. Compute the gravity acceleration field in a grid of nodes that covers 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 to the next timestep;
4. If particles suffer a large displacement, update the gravity field; if not, the previous gravity field is used for the next timestep and phase 2) is executed again.

In ESyS-Particle, the number of calculations to update the self-gravity for N particles and M nodes in the grid is $O(N \times M)$. Every time the total forces of the particles are updated, the self-gravity computation process can last from seconds to hours, depending on the number of particles simulated. To cope with that efficiency problem, High Performance Computing techniques are applied to speed up the calculations.

ESyS-Particle implements spatial domain decomposition using a master-worker model implemented using the Message Passing Interface (MPI) library. A static domain decomposition is used. The decompo-

sition is defined as a parameter before the simulation starts by stating for each axis, how many times the corresponding dimension must be divided.

A two-level parallelization scheme is applied for the proposed self-gravity algorithm, using multithreading programming techniques. On the higher level, the master-worker model already included in ESyS-Particle is applied to calculate and update the forces that affect the particles and it is implemented using a distributed memory scheme, dividing the space of the simulation in different processes. The master process also is in charge of calling the self-gravity module, which calculates and updates the gravity forces that affects the particles. On the lower level, the calculation of the self-gravity forces is implemented using shared memory and multithreading programming techniques.

Before starting a simulation, the self-gravity module builds an overlaying grid to divide the spatial domain of the simulation. The overlaying grid is composed of boxes, whose vertexes are nodes. The number of nodes (M) is related to the number of particles (N) by applying a rule-of-thumb: the distance between nodes is $2.5\times$ larger than the diameter of the largest particle in the simulation, which provides a reasonable trade-off between numerical accuracy and computational efficiency. All computations are performed using the overlaying grid. For example, 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 computed applying Equation (2). Similar equations are formulated for the acceleration along y-axis and z-axis.

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

4 IMPROVING THE SELF-GRAVITY CALCULATION: THE OCCUPIED CELLS METHOD

This section describes performance improvements of the self-gravity calculation algorithm. The improvements lead to the occupied cells method, a baseline method for the self-gravity algorithm to be compared with. Subsection 4.1 explains the occupied cells method. Then, subsection 4.2 reports a profiling analysis of the self-gravity calculation to implement an improved version of the occupied cells method.

4.1 Reducing the Execution Time of the Self-Gravity Computation

In the implementation presented in our previous article (Frascarelli, Nesmachnow, and Tancredi 2014), the self-gravity potential was updated on every node of the overlay grid used to calculate self-gravity. However, recalculating only the values of the acceleration of the occupied nodes is enough to update the acceleration of the particles. A new scheme was proposed and implemented to accelerate the self-gravity computation. In the new scheme, only the occupied cells are updated. This new scheme improved the utilization of the computational resources available by sparing the update of the acceleration of the unoccupied cells.

In the occupied cells method, self-gravity is updated when the variation of the position of any particle between timesteps is larger than a predefined distance threshold. A particle that belongs to a box at the start of a simulation can migrate to different boxes during the simulation. However, that migration may not trigger the self-gravity update if the particle just moves a distance that is not larger than the predefined distance threshold. Given that the acceleration is only updated on the occupied nodes, some of the eight surrounding nodes needed for the interpolation may be outdated. The use of old self-gravity values may introduce error in the calculations during a simulation. To prevent this situation, the self-gravity is updated for an expanded box that comprises the 64 nodes that surround a particle.

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

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

4.2 Profiling the Self-Gravity Calculation

A profiling of the occupied cells algorithm was performed using the VTune Amplifier tool by Intel. (2006) in order to identify bottlenecks on the implementation, to be mitigated before the performance study (Rocchetti, Frascarelli, Nesmachnow, and Tancredi 2017).

According to the profiling results, the most time consuming routine is the one that retrieves the coordinates of the position for each particle. These routines are time consuming because they are called in every update of the acceleration for each node. In addition, they are also affected by an idle CPU utilization of 335 seconds in average, possible associated to memory management issues such as memory transfer operations, which significantly impact the performance when increasing the number of particles involved in a simulation. When dealing with large-scale simulations, the memory needed to store the data structures is usually larger than the cache memory available in a processor. Thus, the time required to move the data through all the memory levels produce the idle CPU time (i.e., the time required to transfer data from the main memory increases the total time spent by each routine).

Performance optimizations were applied, focusing on improving the routine that performs a search over all the nodes of the grid to find a node to update. The improvement consists on iterating only over the list of occupied nodes that is created every time self-gravity needs to be updated. It is usual that in the scenarios used to perform simulation a significant part of the space is empty. So, the list of occupied cells is significantly shorter than the list of nodes. The described improvement also affects the overall performance of the routines that retrieves the coordinates of the position for each particle.

5 BARNES-HUT METHOD FOR SELF-GRAVITY SIMULATIONS ON ESYS-PARTICLE

The Barnes-Hut method uses octal trees to divide the space of a scenario of a simulation. The proposed implementation of the Barnes-Hut algorithm instantiates an octal tree for the complete space of a scenario of a simulation. The octal tree is created and disposed every time the gravitational potential is updated.

The process of updating the self-gravity consists of three steps:

1. *Creation of the self-gravity tree.* The process of creating a Barnes-Hut tree starts by instantiating a root node that represents the complete space defined for the simulation. Due to the nature of the octal tree structure proposed by Barnes and Hut, the space to perform the simulation has cubic shape. 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 on 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.
   Figure 1 presents an example of the partitioning of the space created with the proposed Barnes-Hut method for an agglomerate of particles. The division is applied on three dimensions, but a two-dimensions (2D) projection is presented for proper visualization). The grid over the agglomerate represents the octapole (quadrupole in the 2D projection) tree created using the information of the self-gravity tree. Figure 1 explains the process of tree expansion, which is performed only in those nodes containing particles, via a recursive refinement.

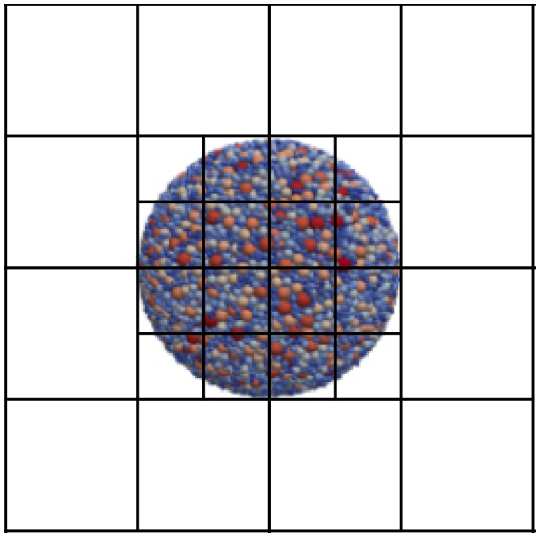

Figure 1: Example of tree partition for an agglomerate of particles (2D projection).

A node of the Barnes-Hut tree represents a cubical part of the space represented by its father node. In the proposed implementation, a node of the tree is represented as a structure that stores relevant attributes: a node identifier, the coordinates of the node, the edge size of the node, the number of particles in the node, the total mass, and the position of the center of mass of all the particles contained in the node. The node identifier is assigned in the process of node creation and it is composed by two numbers: the level of the tree where the node belongs, and the number of the node, that 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. This way, the identifier is initialized to 0 in every level of the tree. With this identification system, the operation that determines if a node is son of another node is executed in $O(1)$. This property allows reducing the time of computing self-gravity on nodes. The center of mass and the total mass of the node are both calculated when the node is instantiated.

2. *Creation of the list of tree nodes.* After creating the expansion tree, the second step in the simulation is to create a list of tree nodes for each of the occupied nodes of the grid. These nodes are called *objective nodes*. For this purpose, the concept of *neighborhood* of a node is introduced. A neighborhood of a specific node is conformed by its surrounding nodes whose distance to the objective node is shorter than a predefined threshold, which is a user-defined parameter for the simulations. Each list of tree nodes is composed of the highest level nodes that are not father of any node that belongs to the neighborhood. For each objective node, the procedure to create the corresponding tree node list starts with the root node. A node is added to the tree node list if it is not father of any member of the neighborhood. Otherwise, if the node is father of at least one member of the neighborhood, then the sons of the node are added to a queue to be evaluated later as candidates to be included in the tree node list.

3. *Self-gravity calculation.* The third step is calculating 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 of tree nodes built in step two, instead of using the occupied cells as proposed in the occupied cells algorithm. After updating the potential for each objective node, the new vector values are communicated to the main force calculation module, to be integrated with the contact forces as computed by ESyS-Particle.

The self-gravity module on ESyS-Particle was extended to implement the proposed methods. For the occupied cell method, specific code was included for retrieving from ESyS-Particle the list of occupied nodes for the simulation step, building the list of nodes to update self-gravity, and constructing the 64-node expansion to update the acceleration (Rocchetti, Frascarelli, Nesmachnow, and Tancredi 2017). For the Barnes-Hut method, two new classes were created to include the basic functionalities of nodes, and to create and dispose the octal tree in each update of the self-gravity using the list of neighbors of a node, retrieved from ESyS-Particle. The self-gravity calculation code in ESyS-Particle was adapted to use the list of nodes retrieved from the corresponding manager class. The main difference between the proposed method and the standard Barnes-Hut lies in the object of the partition. In Barnes-Hut, the space is partitioned again if the created partition is occupied by up to one particle. In the case of the proposed algorithm, the space is partitioned with the same idea but the partitioning stops when the space is empty or the size of a box.

6 EXPERIMENTAL EVALUATION

This section describes the problem scenarios, the computational platform, and the performance analysis of the self-gravity calculation methods implemented in ESyS-Particle. Comparing with methods from the literature requires their adaptation, because they do not compute self-gravity. This is beyond the scope of this article; it is proposed as one of the main lines for future work.

6.1 Test Scenario and Instances

Experiments were performed for a scenario composed of two agglomerates of particles, 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 5m/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 3,000 g/cm^3.

Three instances of the two agglomerates scenario were created to perform the study of the computational efficiency of the self-gravity implementations presented in this article. The first instance defined is a small instance, composed of $3,866$ particles, each one having a radius from 50m to 100m. The second instance is a medium size instance with $11,100$ particles of radii from 35m to 70m. The last instance is a large instance of $38,358$ particles with a radii from 20m to 60m. The mass of the instances is not the same for all the instances and oscillates from 1.2×10^{12} kg to 1.7×10^{12} kg. However, the masses of the instances are of the same order of magnitude. 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.

All instances were simulated for $100,000$ time steps of 0.01 seconds long. The simulations were executed using a varying number of computational resources to evaluate the speedup and scalability of the proposed implementation. In all the instances defined, the initial velocity of all the particles is $5m/s$. 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.

6.2 Hardware Platform

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

6.3 Results and Discussion

Table 1 reports the performance results for the small instance in simulations using both studied methods. The table reports the configuration of processes and threads used for execution, the execution time in seconds, the percentage of the overall execution time spent on self-gravity calculation, the number of self-gravity updates performed, and the average time spent in self-gravity calculations.

Table 1: Performance results for the two agglomerate scenario with 3,866 particles (small instance).

#particle processes	*#gravity threads*	*execution time(s)*	*time computing self-gravity*	*# self-gravity updates*	*avg. self-gravity time(s)*
occupied cells method					
1 (1,1,1)	1	1.06×10^4	86%	1,131	8.15
1 (1,1,1)	2	7.17×10^3	81%	1,131	5.14
2 (1,1,2)	1	1.18×10^4	87%	1,131	9.06
2 (1,1,2)	2	7.79×10^3	78%	1,131	5.39
Barnes-Hut method					
1 (1,1,1)	1	4.09×10^3	74%	1,167	2.60
1 (1,1,1)	2	3.79×10^3	72%	1,167	2.32
2 (1,1,2)	1	4.55×10^3	73%	1,167	2.84
2 (1,1,2)	2	3.95×10^3	68%	1,167	2.30

Using the occupied cells method, the lowest execution time for the small instance was 7.17×10^3 s, using one process for ESyS-Particle and two threads for self-gravity calculation. Performance did not improved when using more computing resources: the execution using two processes and two threads took a longer time to finish. These efficiency results are explained considering a rule of thumb of ESyS-Particle that recommends assigning at least $5,000$ particles to each process. In the small scenario, only one process is recommended according to the rule of thumb. Then, using two processes is a less efficient configuration. The lowest percentage of time computing self-gravity was 78%, when using a configuration with two processes and two threads. The obtained values suggest that the best option to improve the performance is focusing on the self-gravity module, and not in the core modules of ESyS-Particle.

The lowest execution time using the Barnes-Hut method was 3.79×10^3 s, using a configuration of one process and two threads. Similar to the occupied cells method, the performance decreased when using more processes. The aforementioned argument about the rule of thumb to define the number of processes also holds in this case. Regarding the average self-gravity calculation time, the lowest value of the Barnes-Hut method was 2.30 seconds, which is $2.24\times$ faster than the occupied cells method. In addition, using Barnes-Hut, the lower percentage of time computing the self-gravity was 68%, 13% lower than the occupied cells method. Comparing the overall execution time of the small instance, the best result using the Barnes-Hut method was $1.89\times$ faster than the best time using the occupied cells method.

Table 2 reports the performance results for the medium instance using both compared methods. Using the occupied cells method, the configuration with one process and four threads had the lowest execution time (1.31×10^5 s). The lower percentage of time spent on self-gravity calculation was 96%. This result is explained because the number of gravitational force interactions grows in a super-linear manner compared to the linear growth of the contact forces, which causes the increase in the self-gravity computation time. The best average self-gravity calculation time was 73.25 s using the configuration with one process and four threads.

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

Table 2: Performance results for the two agglomerate scenario with 11,100 particles (medium instance).

#particle processes	*#gravity threads*	*execution time(s)*	*time computing self-gravity*	*# self-gravity updates*	*avg. self-gravity time(s)*
occupied cells method					
1 (1,1,1)	1	3.13×10^5	98%	1,733	177.90
1 (1,1,1)	2	1.76×10^5	97%	1,733	99.18
1 (1,1,1)	4	1.31×10^5	96%	1,733	73.25
2 (1,1,2)	1	3.12×10^5	98%	1,755	175.49
2 (1,1,2)	2	2.12×10^5	98%	1,755	118.47
2 (1,1,2)	4	1.61×10^5	97%	1,755	88.99
Barnes-Hut method					
1 (1,1,1)	1	2.80×10^4	79%	1,828	12.18
1 (1,1,1)	2	2.53×10^4	74%	1,828	10.27
1 (1,1,1)	4	2.14×10^4	79%	1,828	9.27
2 (1,1,2)	1	2.43×10^4	81%	1,866	10.50
2 (1,1,2)	2	2.33×10^4	81%	1,866	10.06
2 (1,1,2)	4	2.14×10^4	77%	1,866	8.80

Comparing the results of the medium and small instances using the Barnes-Hut method, the average self-gravity calculation time grown slower than when using the occupied cells method. In addition, the execution time of the medium instance using the occupied cells method is one order of magnitude higher than when using Barnes-Hut.

Table 3 reports the overall execution time in seconds of the algorithm using the Barnes-Hut method for the large instance of the two-agglomerate scenario. The results shown correspond to the different thread configurations and processes with which the algorithm performance tests were performed. Baseline executions using the occupied cells method were not performed due to the large execution times required. The lowest execution time was 4.96×10^4 s, using the configuration with four processes and four threads. Also, results show that increasing the computational resources not always decreases the overall execution time of the algorithm. The execution time decreases when increasing the computational resources up to the scenario using four processes and four threads. For the scenarios that uses either more processes or more threads, the overall execution time increases.

Using the Barnes-Hut method, in the small instance the self-gravity was updated 1,167 times, whereas on the medium instance it was updated from 1,828 to 1,866 times and on the large instance it was updated between 3,781 and 3,813 times. So, the number of gravity updates increases when the size of the particles decreases. Using the Barnes-Hut method to simulate the two agglomerates scenario lowered the percentage of time calculating the self-gravity in up to 23%. This effect is caused by the up to $10\times$ 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.

Results show that the performance of the proposed implementation of the Barnes-Hut method increase when using more computational resources. However, the performance increase is sub linear with respect of the number of computational resources. The reason for this behavior is twofold: i) the average self-gravity calculation time is usually too short to fully exploit the benefits of a parallel environment (i.e., following a theoretical linear increase), and ii) the process in charge of creation and deletion of the Barnes-Hut tree are not implemented in parallel, so a significant part of the time updating the self-gravity is spent performing sequential operations. In further experiments, the efficiency of the Barnes-Hut method allowed to perform realistic simulations in scenarios with 2,100,000 particles.

Table 3: Performance results of the Barnes-Hut method for the two-agglomerate scenario with 38,538 particles (large instance).

#particle processes	*#gravity threads*	*execution time(s)*	*time computing self-gravity*	*# self-gravity updates*	*avg. self-gravity time(s)*
1 (1,1,1)	1	8.58×10^4	71%	3,813	16.06
1 (1,1,1)	2	7.45×10^4	76%	3,813	12.22
1 (1,1,1)	4	6.00×10^4	74%	3,813	11.67
1 (1,1,1)	8	5.48×10^4	77%	3,813	11.01
1 (1,1,1)	16	8.32×10^4	68%	3,815	14.84
2 (1,1,2)	1	7.74×10^4	77%	3,807	15.65
2 (1,1,2)	2	6.34×10^4	69%	3,807	11.48
2 (1,1,2)	4	5.96×10^4	71%	3,813	11.13
2 (1,1,2)	8	6.52×10^4	76%	3,807	13.09
2 (1,1,2)	16	7.36×10^4	76%	3,807	14.69
4 (1,2,2)	1	7.32×10^4	83%	3,781	16.08
4 (1,2,2)	2	6.31×10^4	75%	3,781	12.43
4 (1,2,2)	4	4.96×10^4	88%	3,781	11.50
4 (1,2,2)	8	6.14×10^4	84%	3,781	13.67
4 (1,2,2)	16	6.77×10^4	83%	3,781	14.84
8 (2,2,2)	1	7.43×10^4	83%	3,781	16.26
8 (2,2,2)	2	7.14×10^4	76%	3,781	14.35
8 (2,2,2)	4	6.35×10^4	84%	3,781	14.03
8 (2,2,2)	8	6.77×10^4	80%	3,781	14.37
8 (2,2,2)	16	6.35×10^4	82%	3,781	13.79

6.4 Numerical Accuracy

The numerical accuracy of the proposed method was studied by analyzing the position of the center of mass of the studied system. Results show that the absolute error in the position of the center of mass is of the order of 10^{-1} m for the large instance of the two agglomerates scenario (the reference is the ideal value 0, which corresponds to a still center of mass). The other instances showed less movement of the position of the center of mass. These results suggest that strategies to decrease the movement should be implemented and analyzed when increasing the number of particles.

7 CONCLUSIONS AND FUTURE WORK

This article presented parallel multithreading algorithms for self-gravity computation in ESyS-Particle, to improve the computational efficiency of simulations involving large number of particles. Two methods were presented and compared: the occupied cells method, and the Barnes-Hut method.

The occupied cells method updates the acceleration of the occupied nodes on an overlying grid, and uses a buffering structure where nodes can move to in future time steps. A profiling analysis was performed to identify bottlenecks and reduce the number of invocations of time consuming routines. The Barnes-Hut method applies a domain partitioning based on an octal tree. The root of the tree is the whole simulation domain and the tree nodes are created by recursively refining the space into eight regular cubical cells.

The experimental evaluation of the two proposed methods for self-gravity calculation was performed over a realistic astronomical scenario where two agglomerates orbit each other. Three problem instances were defined considering different number of particles: $3,866$ (small), $11,100$ (medium), and $38,538$ (large). The two-level parallel model was studied by assigning different computing resources to ESyS calculation processes and self-gravity calculation threads.

Experimental results for the two agglomerate scenario show that the average self-gravity calculation time is up to $10\times$ faster for the Barnes-Hut method compared to the occupied cells method. As a consequence, the execution time reported for the simulations using the Barnes-Hut method was lowered in one order of

magnitude compared to using the occupied cells method. The efficiency of the Barnes-Hut method allowed to perform realistic simulations in scenarios with more than one million particles.

The main lines for future work include extending the performance evaluation of the implemented self-gravity simulations to consider larger problem instances and different scenarios and comparing with other methods from the literature. In addition, we are currently working on developing a comprehensive benchmark for performance comparison with other self-gravity simulators. The proposed method is also appropriate for GPU, and an implementation over this platform is proposed as future work.

REFERENCES

Abe, S., C. Altinay, V. Boros, W. Hancock, S. Latham, P. Mora, D. Place, W. Petterson, Y. Wang, and D. Weatherley. 2009. "ESyS-Particle: HPC Discrete Element Modeling Software". *Open Software License version* 3.

Barnes, J., and P. Hut. 1986. "A Hierarchical O(N log N) Force-Calculation Algorithm". *Nature* 324(6096):446–449.

Couchman, H. M. P. 1991. "Mesh-Refined P3M-A Fast Adaptive N-body Algorithm". *The Astrophysical Journal* 368:L23–L26.

Cundall, P., and O. Strack. 1979. "A Discrete Numerical Model for Granular Assemblies". *Geotechnique* 29(1):47–65.

Foster, I. 1995. *Designing and Building Parallel Programs: Concepts and Tools for Parallel Software Engineering*. Boston, Massachusetts: Addison-Wesley.

Frascarelli, D., S. Nesmachnow, and G. Tancredi. 2014. "High-Performance Computing of Self-gravity for Small Solar System Bodies". *Computer* 47(9):34–39.

Greengard, L., and V. Rokhlin. 1987. "A Fast Algorithm for Particle Simulations". *Journal of Computational Physics* 73(2):325–348.

Hager, G., and G. Wellein. 2010. *Introduction to High Performance Computing for Scientists and Engineers*. Boca Raton, Florida: CRC Press.

Harris, A., E. Fahnestock, and P. Pravec. 2009. "On the Shapes and Spins of "Rubble Pile" Asteroids". *Icarus* 199(2):310–318.

Intel. 2006. "Intel VTune Amplifier 2017". https://software.intel.com/en-us/intel-vtune-amplifier-xe. accessed 1st April 2019.

Kravtsov, A., A. Klypin, and A. Khokhlov. 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., D. Frascarelli, and G. Tancredi. 2015. "A Parallel Multithreading Algorithm for Self-gravity Calculation on Agglomerates". In *International Supercomputing Conference in México*, 311–325. Springer.

Rocchetti, N., D. Frascarelli, S. Nesmachnow, and G. Tancredi. 2017. "Performance Improvements of a Parallel Multithreading Self-gravity Algorithm". In *Communications in Computer and Information Science*, 291–306. Springer.

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

Sánchez, P., and D. Scheeres. 2012. "DEM Simulation of Rotation-Induced Reshaping and Disruption of Rubble-Pile Asteroids". *Icarus* 218(2):876–894.

Tancredi, G., A. Maciel, L. Heredia, P. Richeri, and S. Nesmachnow. 2012. "Granular Physics in Low-Gravity Environments Using Discrete Element Method". *Monthly Notices of the Royal Astronomical Society* 420:3368–3380.

Walsh, K., D. Richardson, and P. Michel. 2012. "Spin-up of Rubble-Pile Asteroids: Disruption, Satellite Formation, and Equilibrium Shapes". *Icarus* 220(2):514–529.

AUTHOR BIOGRAPHIES

SERGIO NESMACHNOW is Full Professor at Universidad de la República, Uruguay. He holds a PhD in Computer Science from Universidad de la República. His research interests include high performance computing and simulation, metaheuristics, and smart cities. Email address: sergion@fing.edu.uy.

NESTOR ROCCHETTI is Assissant Professor at Universidad de la República, Uruguay. He holds a Engineering Degree from Universidad de la República. His research interests include high performance computing and simulation. Email address: nrocchetti@fing.edu.uy.

GONZALO TANCREDI is Full Professor at Universidad de la República, Uruguay. He holds a PhD in Astronomy from Uppsala University, Sweden. His research interests include planetary sciences, small solar system bodies and impact process, by observations, laboratory experiments and numerical simulations. Email address: gonzalo@fisica.edu.uy.