Parallel Implementations of Self-gravity Calculation for Small Astronomical Bodies on Xeon Phi

Sebastián Caballero, Andrés Baranzano, and Sergio Nesmachnow(✉)

Facultad de Ingeniería, Universidad de la República,
Herrera y Reissig 565, Montevideo, Uruguay
{sebastian.caballero,andres.baranzano,sergion}@fing.edu.uy

Abstract. This article presents parallel implementations of the Mass Approximation Distance Algorithm for self-gravity calculation on Xeon Phi. The proposed method is relevant for performing simulations on realistic systems modeling small astronomical bodies, which are agglomerates of thousand/million of particles. Specific strategies and optimizations are described for execution on the Xeon Phi architecture. The experimental analysis evaluates the computational efficiency of the proposed implementations on realistic scenarios, reporting the best options for the implementation. Specific performance improvements of up to **146.4**× are reported for scenarios with more than one million particles.

Keywords: Multithreading · Self-gravity · Xeon Phi

1 Introduction

Self-gravity is a long range interaction caused by the mutual influence of particles that conform an agglomerate. This interaction is important to model the dynamic of small astronomical objects like asteroids and comets, which are agglomerates of smaller particles kept together by the gravitational force [1].

Due to the intrinsic complexity of modeling the interactions between particles, agglomerates are studied using computational simulations. Molecular Dynamics (MD) is a simulation method to study physical systems, including granular materials. Trajectories of atoms and molecules in the system are determined by numerically solving Newton's equations of motion of interacting particles over a fixed period of time. Forces between particles and their potential energies are calculated using potentials or force fields, allowing to get a vision of the dynamic evolution of the system [2]. While MD is used to model systems in atomic scale (atoms or molecules), a more general approach for simulation is the Discrete Element Method (DEM) [3].

DEM is a numerical method used for simulating systems involving a large number of small particles. DEM is closely related to MD but allows simulating in larger scale, such as discontinuous materials (powders, rocks, granular), including rotational degrees-of-freedom and contact forces between particles.

E. Meneses et al. (Eds.): CARLA 2018, CCIS 979, pp. 157–173, 2019.
https://doi.org/10.1007/978-3-030-16205-4_12

When applying numerical techniques for simulation, such as DEM, execution times for self-gravity calculation demand minutes, or even hours. High Performance Computing (HPC) techniques are applied to speed up the computation when simulating real scenarios involving a large number of particles [4].

In this line of work, this article presents a parallel implementation of Mass Approximation Distance Algorithm (MADA) to compute self-gravity on systems of particles and evaluates optimizations for execution on the Intel Xeon Phi architecture. The experimental analysis allows concluding that the proposed implementations are able to significantly accelerate the execution time of realistic simulations. Performance results show accelerations of up to 146× are obtained by the best parallel implementation when compared to a sequential version.

The article is organized as follows. The problem of computing self-gravity on small astronomical bodies and a review of related work is presented in Sect. 2. Section 3 describes multithreading libraries for Intel Xeon Phi. The proposed implementations of MADA are presented in Sect. 4 and the experimental evaluation is reported in Sect. 5. Finally, Sect. 6 presents the conclusions and formulates the main lines for future work.

2 Self-gravity Computation on Small Astronomical Bodies

This section introduces the problem of computing self-gravity on small astronomical bodies and the approximation using MADA. In addition, a review of related works about parallel algorithms for self-gravity and other particle interactions in agglomerates is presented.

2.1 Self-gravity Calculation on Agglomerates

Asteroids and comets are agglomerates of small particles that are held together by the action of different forces. One of the most important of these forces is self-gravity [4].

The problem of computing self-gravity considers an agglomerate composed by N particles and M_i the mass of the i-th particle, whose center is located in position $\boldsymbol{r_i}$. The gravitational potential V_i generated in particle i due to the action of the rest of the particles is determined by Eq. 1, where G is the gravitational constant and $||\boldsymbol{r_x}||$ is the norm of the vector r_x.

$$V_i = \sum_{j \neq i} \frac{GM_j}{||\boldsymbol{r_i} - \boldsymbol{r_j}||} \tag{1}$$

When the number of particles in an agglomerate is in the order of millions, executing an algorithm that iterates over all particles becomes unpractical since the execution time grows quadratically ($O(N^2)$) with respect to the size of the input data. In order to model the dynamics of an astronomical system, a large

number of simulations are required. Thus, using a straightforward $O(N^2)$ algorithm for self-gravity calculation demands a significantly large execution time. For this reason, approximation algorithms are applied to compute accurate estimations of the gravitational potential in shorter execution times.

2.2 Mass Approximation Algorithm

MADA is an approximation algorithm for calculating the self-gravity of a system of particles. The main idea of MADA is substituting groups of distant particles for a single particle located at the center of mass of the group. The considered groups involve larger sets of particles when they are located far from the particle in which self-gravity is computed.

Initially, MADA divides the calculation domain in a certain number of partitions on each axis. This partitioning forms cubes that are called sectors. For particles belonging to the same sector, the self-gravity force between them is calculated exactly, applying Eq. 1 without using an approximation. Particles that do not belong to the sector of the particle where self-gravity is computed (target particle) are grouped in subsectors of variable size, depending on the distance to the sector of the target particle. For each of these subsectors, the self-gravity between the target particle and the center of mass of the sub-sector is computed, avoiding to perform a large number of calculations.

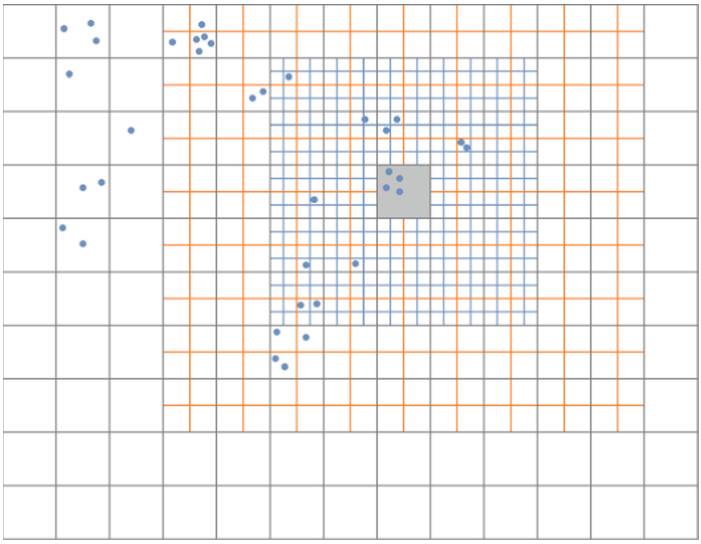

Fig. 1. Division of a domain into subdomains to process the gray sector. The closest sectors to the grey sector are divided into larger subdivisions. Within the grey sector, MADA is not applied, particle to particle calculation is used instead.

Figure 1 shows a two-dimension representation of the domain decomposition applied by MADA to compute self-gravity on particles of the grey sector

(target sector). The farthest sectors are processed as a single particle by using the approximation proposed by MADA. Sectors that are closer to the target (grey) sector have more levels of division, as its contributions are more significant than the one from farthest sectors.

MADA allows reducing the calculations need for computing self-gravity by grouping distant particles and treating them as a single particle. The calculated center of mass can be stored and reused when computing self-gravity of other particles, e.g. (potentially millions of times for agglomerates of millions of particles) since the MADA sectors are fixed for all particles of the agglomerate.

The division into sectors proposed by the MADA algorithm allows to process subsets of sectors by different threads. Also the center of gravity calculations can be shared among them. The application of parallelism allows to reduce the execution times and consequently reduce the execution times of the simulations.

2.3 Related Work: Parallel Algorithms for Self-gravity and Other Particle Interactions Calculation

Our research group at Universidad de la República has published previous articles on parallel algorithms for self-gravity calculation in particle systems.

MADA was introduced by Frascarelli et al. [5], including a parallel implementation for simulating large systems in a cluster. The experimental analysis studied the relative error when using MADA (less than 0.1% for all problem instances) and the speedup (up to 15× on AMD Opteron 6172 processors).

Four parallel strategies for domain decomposition and workload assignment for threads for the previous MADA implementation were studied [6]. The Advanced Isolated Linear strategy obtained the best performance and scaled up for simulating an astronomical agglomerate of 1218024 particles using 12 execution threads. The best strategy also allowed to reduce significantly the execution times: speedup values up to 13.4 (computational efficiency 0.85) were obtained.

Later, Rocchetti et al. [7] studied a MADA implementation included in the ESyS-Particle library for DEM simulations. A profiling analysis was performed using Intel VTune Amplifier to detect bottlenecks and the most time-consuming subroutines were reimplemented to improve execution time. In the improved version, particle acceleration is computed for a surrounding box for each particle and empty cells are omitted. The experimental evaluation studied a two-agglomerates scenario with up to 38538 particles and the performance results reported that the execution time of self-gravity calculation was reduced up to 50× when compared with a baseline non-optimized implementation.

The MADA algorithm was not ported/adapted for execution on Xeon Phi in any of the aforementioned previous works.

Regarding parallel implementations on Xeon Phi for simulating other phenomena in granular systems, Rönnbäck [8] studied optimizations for Parallel Projected Gauss-Seidel method. The study was focused on bottleneck and scalability analysis for non-trivial parallel programs. However, no specific recommendations about how to port this kind of applications to Xeon Phi was presented.

Surmin et al. [9] presented a parallel implementation of the Particle-in-Cell method for plasma simulation on Intel Xeon Phi. The parallel method improved the performance of laser acceleration simulations up to 1.6× when compared with an implementation on Xeon processors. The analysis also showed that vectorization significantly contributed to performance improvements.

Pennycook et al. [10] described a bottleneck analysis of accumulation and dispersion processes in particle dynamics. A Single Instruction Multiple Data (SIMD) approach was proposed to improve execution time using specific SIMD operations provided by Intel Xeon/Xeon Phi architectures. The bottleneck analysis was performed for the miniMD algorithm [11], using different combinations of 128, 256, and 512 bits SIMD operations on Intel Xeon/Xeon Phi. The best results were obtained using 512 bits SIMD operations on Xeon Phi and manual vectorization. This work proved that the use of SIMD operations reduces the execution times (up to 5× faster) for miniMD algorithm.

The related works showed no previous proposals of Xeon Phi implementations of self-gravity calculation or similar particle interaction methods.

3 Multithreading Libraries for Intel Xeon Phi

This section describes different multithreading libraries that are compatible with Intel Xeon Phi. Two of them are applied in this article in the proposed implementations for self-gravity calculation for agglomerates.

3.1 Pthreads

Pthreads is a standard model to divide programs into subtasks that can run in parallel. Pthreads defines a set of types, functions, and constants of the C programming language to create, destroy, and synchronize execution of threads and provides functions for managing concurrency in shared memory.

The main advantage of using pthreads is the low cost of creation and destruction of threads, which is 100 to 150 times faster than for processes [12]. The cost of access to shared memory between threads is lower (threads of a process share all the memory) and the context switch between threads requires less execution time since contexts share information between threads of the same process (but not between processes). Multithreading libraries such as Cilk Plus, Thread Building Blocks, and OpenMP use pthreads internally for thread management.

3.2 Intel Cilk Plus

Cilk Plus is an extension of C/C++ to support data and task parallelism. It provides basic functions, array notation, and compiler directives to execute SIMD instructions. The main advantages of Cilk Plus are: ease of use, maintainability, and the few changes required to transform a sequential code into a parallel one.

Cilk Plus provides support to automatically execute parallel loops via `cilk_for`. It dynamically creates execution threads and assigns work to them,

following a divide-and-conquer pattern. By default, Cilk Plus determines the optimal level of parallelism by considering the workload and the cost of creating new threads with *cilk_spawn* function but the programmer can manually specify a fixed number. This strategy is effective to calculate the self-gravity of an agglomerate using MADA, because there is no dependency between MADA sectors.

3.3 Intel Threading Building Blocks

Thread Building Blocks (TBB) is a C++ library for developing multithread applications. It uses C++ templates that provides automatic thread management and scheduling to implement loops that run in parallel, allowing developers to generate parallel code without need to handle the creation, destruction, and synchronization of execution threads. Furthermore, TBB handles load balancing between threads and provides mechanisms for concurrent reading and writing.

TBB is a data level parallelism library. Each thread works on a portion of input data, so it benefits directly from having a greater number of processing units. MADA algorithm can take advantage of this type of load division, since input data is divided into sectors that are processed independently of each other.

3.4 Comparative Analysis

Comparative analysis of the parallel multithreading libraries described in this section were performed by Ajkunic [13] and Leist and Gilman [14]. Results confirmed that all of them are able to improve the execution time of simulations but there is not much difference regarding performance between them.

Each library has its own advantages and disadvantages. Choosing one or the other depends on the type of application and the host architecture. Pthreads provides a low level model for shared-memory parallel programing, without including a task scheduler. It requires the developer to directly manage the execution threads, implying a large effort in comparison to developing the same program using a multithreading library Pthreads also requires the implementation of a specific task scheduler for the developed application. For these reasons, pthreads is not considered for developing parallel implementations for self-gravity in agglomerates in the research reported in this article.

4 Parallel Implementations for Self-gravity Calculation

This section presents the proposed implementations of MADA for Xeon Phi: a sequential version used as baseline to compare results/efficiency of the proposed parallel methods, and the parallel implementations using Cilk Plus and TBB.

4.1 Baseline Method: Sequential Implementation

The sequential implementation of MADA (Algorithm 1) does not use parallelism. The method in line 1 initializes the data structures used. After that, load input (line 2) reads input data for the current execution and arranges them in the data structures. Centers of mass for each sector defined by MADA are computed before processing. This pre-computation avoids performing concurrency checks during execution, thus decreasing waiting time between threads.

Algorithm 1. MADA: sequential implementation

```
1: Initialize sectors and centers of mass
2: Load input
3: Pre-compute centers of mass
4: for each sector s in domain do
5:     Process sector(s)
6: end for
```

Algorithm 2 describes the process sector subroutine. To reduce workload, self-gravity is computed for points that define sectors with the smallest subdivision. This data can be interpolated to obtain the self-gravity potential for the system. For each point p_i that defines the greatest subdivision of sector s_h the algorithm has two parts: (i) particle to particle interactions are computed between point p_i and all particles in sector s_h; (ii) for each other sector s_k, self-gravity is computed between p_i and the centers of mass of its subdivisions. MADA dynamic grid is used to determine the subdivisions the centers of mass.

Algorithm 2. Process sector(s_i)

```
1: for each smallest subdivision in sector s_h do
2:     Determine particle p_i that identifies sector s_i
3:     for each particle p_j in s_h do
4:         Calculate self-gravity between p_i and p_j
5:     end for
6:     for each sector s_k, s_k ≠ s_h do
7:         Find distance to s_k
8:         Determine sector subdivision according to distance
9:         for each subdivision d_l of s_k do
10:            Calculate self-gravity between p_i and center of mass of d_l
11:        end for
12:    end for
13: end for
```

4.2 Parallel Implementation Using Cilk Plus

Two parallel implementations of MADA using Cilk Plus were developed to study explicit vs. automatic vectorization: (i) using Array of Structures (AoS) and (ii) using Struct of Arrays (SoA). Preliminary tests showed that AoS outperformed SoA, therefore only the AoS approach is presented in this article.

Intel Cilk Plus: AoS Approach. Figure 2 shows a diagram of the domain division performed by `cilk_for`. The array of sectors is divided in two, and each half is divided in two again, until there the domain division guarantee load balancing: each thread processes the same number of sectors. However, sectors usually have different number of particles, which translates into more work in some threads. The Cilk Plus scheduler does not know beforehand which sectors of the domain should be processed, resulting in a large amount of lost time allocating empty sectors, or in threads with a higher workload. To avoid this problem, only those sectors that have particles in the data loading process are stored.

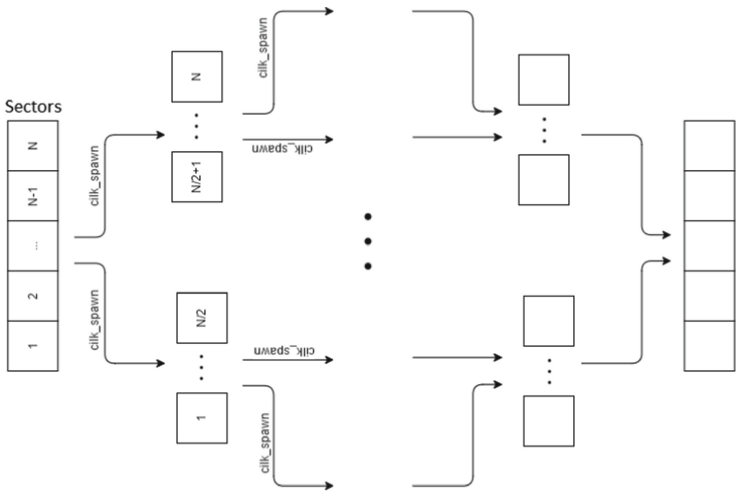

Fig. 2. Domain division of the sector array performed by `cilk_for`

Intel Cilk Plus: AoS and Array Notation. Cilk Plus Array Notation is a variant for explicitly specifying vectorized operations using an own syntax. The data structures remain unchanged with respect to those used in the AoS implementation. Specific changes are made to the vectorized loops, as shown in Algorithm 3, where the loops are replaced by operations with array notation.

Algorithm 3. Particle-to-particle self-gravity calculation using array notation

```
1: int size = sectors[part_sector].size
2: double self_grav = 0
3: Part * cPart = sectors[part_sector].particles
4: __assume_aligned(cPart, 64)
5: self_grav += __sec_reduce_add((G * cPart[0:size].mass) / sqrt(pow(cPart[0:size].x -
   p→ x, 2) + pow(cPart[0:size].y - p→ y, 2) + pow(cPart[0:size].z - p→ z, 2)))
6: return self_grav
```

Algorithm 3 makes use of two extensions of the language introduced by Cilk Plus. The syntax `currentParticles[i:length]` indicates the compiler that the instruction must be performed for each value of the array `currentParticles` between the index value i and $i+length$. Given an instruction in array notation, operation `_sec_reduce_add` sums the result of each part of the array in a numeric variable. Two examples of equivalent code are shown on Listings 1.1 and 1.2.

```
for (int i = 0; i < size; i++) {
    dx[i] = currentParticles[i].x − p−>x;
}
```

Listing 1.1. Standard loop

```
dx[0:size] = currentParticles[0:size] − p−>x;
```

Listing 1.2. Equivalent loop to Listing 1.1 using array notation

4.3 Parallel Implementation Using Thread Building Blocks

The parallel implementation of MADA using TBB was developed by performing a set of modifications over the AoS Cilk Plus code, to evaluate the performance of the TBB scheduler vs. the Cilk Plus scheduler.

Intel TBB provides its own parallel for. The differences with *cilk_for* are presented in Listings 1.3 and 1.4. Modifications are needed to adapt the Cilk Plus implementation to use Intel TBB to manage threads and workloads.

```
cilk_for(int i = 0; i < sectors_to_process_length; i++){
    process_sector(sectors_to_process[i], __cilkrts_get_worker_number());
}
```

Listing 1.3. Cilk For

```
parallel_for<int>(0, sectors_to_process_length, 1, process_sector );
```

Listing 1.4. TBB Parallel for

Unlike Cilk Plus, TBB does not provide a method to identify the thread number in execution at a given time. Thus, an index must be manually assigned to access the reserved memory for the thread and to do so, a `concurrent_hash_map`

from TBB is implemented to allow the execution of concurrent reads. The concurrent hash maps the ID of the thread with the current number of threads in execution. The number of executing threads is calculated adding to a mutually excluded counter every time a new thread is created. The new data structures needed for the TBB implementation of MADA are shown in Listing 1.5.

```
typedef concurrent_hash_map<tbb::internal::tbb_thread_v3::id,int>
WorkerTable;
int thread_count = 0;
typedef spin_mutex ThreadCountMutexType;
ThreadCountMutexType threadCountMutex;
WorkerTable workerTable;
```

Listing 1.5. TBB structures

A specific method was created to obtain the thread number and access the reserved memory of that thread.

5 Experimental Evaluation

This section reports the experimental evaluation of the proposed MADA implementations to compute self-gravity on particle systems.

5.1 Methodology

The analysis compares the performance of the parallel implementations of MADA using Cilk Plus and TBB with the baseline sequential implementation.

Efficiency Metrics. Three independent executions were performed using 1, 60, 120, 180, and 240 threads, to minimize variations due to non-determinism in the execution. Average and standard deviation of the execution times are reported. Standard metrics to evaluate the performance of parallel algorithms are studied: *speedup*, the ratio of the execution time of the sequential and the parallel version. and *computational efficiency*, the normalized value of the speedup.

Execution Platform. Experiments were performed on a Xeon Phi 31S1P from Cluster FING, Universidad de la República, Uruguay [15]. It was used in dedicated mode to prevent external processes from affecting the execution times.

Self-gravity Problem Instances. The experimental analysis was performed over six problem instances that model small astronomical bodies with different characteristics. The main details of the problem instances are described in Table 1.

All executions used a 0-0-0-1 configuration for MADA: the three closest sectors to the one processed are computed particle by particle, the fourth using a single subdivision level and from the fifth, the center of mass calculation is used. Equation 2 is used to determine the number of partitions for a domain, to

Table 1. Self-gravity problem instances used in the experimental evaluation

#I	Size	Domain radius (R)	Particle radius (r)	Partitions (P)
1	21.084	10	0.168–0.672	125
2	17.621	10	0.336	1.331
3	167.495	20	0.168–0.672	1.331
4	148.435	20	0.336	12.167
5	1.304.606	40	0.168–0.672	12.167
6	1.218.024	40	0.336	103.823

assure that the smallest subdivision is at least 2.5 times the maximum radius of particles [7].

$$P = \left\lfloor \frac{R}{2.5 \times r \times 2} \right\rfloor^3 \tag{2}$$

5.2 Sequential Implementation

Table 2 reports the execution times for the sequential MADA implementation.

Table 2. Execution time results for the sequential MADA implementation

#I	#sectors	Execution time (s)	Std. deviation (s)	Std. deviation (%)
1	125	0.678	0.010	1.48
2	1331	4.097	0.027	0.66
3	1331	15.054	0.48	3.19
4	12167	14.487	0.18	1.24
5	12617	413.250	3.184	0.77
6	103823	18957.014	117.702	0.62

Results show that execution times depend on the number of sectors processed and not on the number of particles. Instance #5 has more particles than instance #6 but but it demands shorter execution time since the number of MADA sectors MADA is larger (as it depends on the maximum radius of particles).

5.3 Intel Cilk Plus

Cilk Plus with AoS. Table 3 reports execution times, speedup, and efficiency (*eff*) of the parallel implementation using Cilk Plus, AoS, and optimizations for automatic vectorization. for different number of threads ($\#t$). The best execution time of each instance is marked in bold. Figure 3 summarizes the results.

Table 3. Performance results: MADA Cilk Plus implementation using AoS

#I	#t	Time (s)	Speedup	Eff.	#I	#t	Time (s)	Speedup	Eff.
1	1	0.962	–	–	4	1	16.321	–	–
	60	**0.339 ± 0.010**	**2.84**	**0.05**		**60**	**1.807 ± 0.077**	**9.03**	**0.15**
	120	0.444 ± 0.011	2.17	0.02		120	2.724 ± 0.455	5.99	0.05
	180	0.544 ± 0.015	1.77	0.01		180	2.028 ± 0.010	8.05	0.04
	240	0.6 ± 0.020	1.60	0.01		240	2.147 ± 0.023	7.60	0.03
2	1	6.298	–	–	5	1	580.659	–	–
	60	**0.583 ± 0.021**	**10.80**	**0.18**		60	21.762 ± 0.058	26.68	0.44
	120	0.661 ± 0.003	9.53	0.08		120	17.845 ± 0.195	32.54	0.27
	180	0.833 ± 0.025	7.56	0.04		180	16.247 ± 0.016	35.74	0.20
	240	0.895 ± 0.019	7.04	0.03		**240**	**15.855 ± 0.486**	**36.62**	**0.15**
3	1	0.895	–	–	6	1	25913.53	–	–
	60	**2.017 ± 0.017**	**10.81**	**0.18**		60	487.294 ± 2.918	53.18	0.89
	120	2.117 ± 0.030	10.30	0.09		120	276.941 ± 1.483	93.57	0.78
	180	2.211 ± 0.031	9.86	0.05		180	206.556 ± 0.598	125.46	0.70
	240	2.411 ± 0.017	9.04	0.04		**240**	**183.875 ± 0.369**	**140.93**	**0.59**

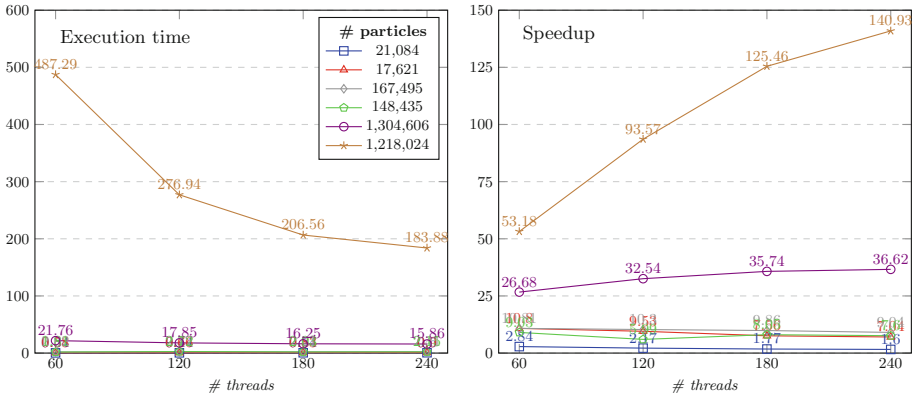

Fig. 3. Performance results: MADA Cilk Plus implementation using AoS

The Cilk Plus implementation significantly reduced the execution time to less than a hundredth of the sequential version to execute instance #6. To evaluate the impact of using vectorization, the Cilk Plus implementation was executed using the number of threads that obtained the best time for each instance and compared with a non-vectorial execution. Table 4 summarizes the results. The impact of the vectorization is very high, obtaining accelerations of 7.1 and 5.8 in the largest instances (5 and 6). This imply shorter execution times, e.g., from 1060 s to 183 s for instance 6. This results imply a significantly higher scalability of self-gravity calculation when vectorized operations are used.

Table 4. Execution time and acceleration with/without vectorization in Cilk Plus

Instance	#threads	Time vectorial (s)	Time non vectorial (s)	Acceleration
1	60	0,339 ± 0,010	1,313 ± 0,025	3,873
2	60	0,583 ± 0,021	2,363 ± 0,670	4,053
3	60	2,017 ± 0,017	14,879 ± 0,477	7,377
4	60	1,807 ± 0,077	5,238 ± 0,095	2,899
5	240	15,855 ± 0,486	116,682 ± 16,334	7,061
6	240	183,875 ± 0,369	1060,763 ± 14,619	5,769

Cilk Plus with Array Notation. Table 5 reports the execution time, speedup, and efficiency of the implementation using Cilk Plus with array notation, varying the number of threads. Results are graphically compared in Fig. 4.

Table 5. Performance results: MADA Cilk Plus implementation using AoS and array notation

#I	#t	Time (s)	Speedup	Eff.	#I	#t	Time (s)	Speedup	Eff.
1	1	0.716	–	–	4	1	13.594	–	–
	60	**0.348 ± 0.020**	**2.06**	**0.03**		**60**	**1.822 ± 0.034**	**7.46**	**0.12**
	120	0.436 ± 0.020	1.64	0.01		120	2.384 ± 0.059	5.70	0.05
	180	0.541 ± 0.007	1.32	0.01		180	3.182 ± 0.149	4.27	0.02
	240	0.616 ± 0.036	1.16	0.00		240	4.208 ± 0.243	3.23	0.01
2	1	4.714	–	–	5	1	492.541	–	–
	60	**0.516 ± 0.007**	**9.14**	**0.15**		60	19.739 ± 0.040	24.95	0.42
	120	0.613 ± 0.022	7.69	0.06		**120**	**16.911 ± 0.025**	**29.13**	**0.24**
	180	0.718 ± 0.008	6.57	0.04		180	17.131 ± 0.165	28.75	0.16
	240	0.842 ± 0.013	5.60	0.02		240	18.677 ± 0.264	26.37	0.11
3	1	15.779	–	–	6	1	24151.367	–	–
	60	**1.945 ± 0.007**	**8.11**	**0.14**		60	452.197 ± 1.092	53.41	0.89
	120	2.081 ± 0.015	7.58	0.06		120	268.803 ± 1.186	89.85	0.75
	180	2.169 ± 0.016	7.27	0.04		180	214.701 ± 1.334	112.49	0.62
	240	2.324 ± 0.019	6.79	0.0		**240**	**206.636 ± 0.788**	**116.88**	**0.49**

The implementation using array notation obtained similar results to the implementation using automatic vectorization in executions with 60 threads. However, when using more computing resources, the speedup decreased and the execution times worsen with respect to the implementations analyzed in the previous sections. Graphics in Fig. 4 show that the improvements when using 120, 180, and 240 threads are lower than those obtained when using automatic vectorization. The speedup obtained for instance #6 with 240 threads was 116.88, significantly lower than the one obtained with automatic vectorization.

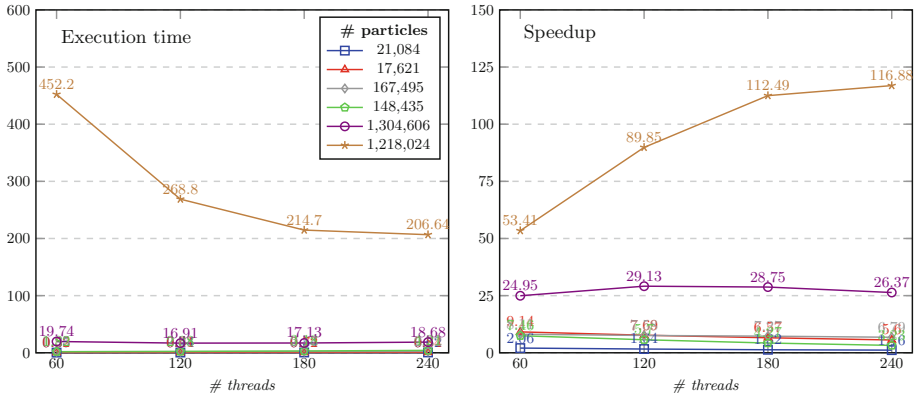

Fig. 4. Performance results: MADA Cilk Plus implementation using AoS and array notation.

In any case, implementing vectorization using array notation is simpler for the programmer. It is done explicitly and data dependency rules do not have to be checked. Array notation is a viable alternative for non-expert programmers to obtain performance improvements with little implementation effort.

5.4 Intel Thread Building Blocks

Table 6 reports the execution time, speedup, and efficiency of the implementation using Intel TBB for thread management and optimizations for automatic vectorization, varying the number of threads. Figure 5 summarizes the results.

Table 6. Performance results: MADA TBB implementation

#I	#t	Time (s)	Speedup	Eff.	#I	#t	Time (s)	Speedup	Eff.
1	1	0,991	–	–	4	1	15,005	–	–
	60	**0,436 ± 0,005**	**2,27**	**0,04**		**60**	**1,912 ± 0,021**	**7,85**	**0,13**
	120	0,612 ± 0,035	1,62	0,01		120	2,181 ± 0,039	6,88	0,06
	180	0,955 ± 0,070	1,04	0,01		180	2,466 ± 0,035	6,08	0,03
	240	1,348 ± 0,281	0,74	0,00		240	3,011 ± 0,120	4,98	0,02
2	1	6,221	–	–	5				
	1	583,309	–	–					
	60	**0,675 ± 0,029**	**9,22**	**0,15**		60	21,376 ± 0,239	27,29	0,45
	120	0,847 ± 0,017	7,34	0,06		120	17,336 ± 0,214	33,65	0,28
	180	1,202 ± 0,036	5,18	0,03		**180**	**16,537 ± 0,034**	**35,27**	**0,20**
	240	1,792 ± 0,119	3,47	0,01		240	16,999 ± 0,140	34,31	0,14
3	1	21,854	–	–	6	1	25543,485	–	–
	60	**2,235 ± 0,068**	**9,78**	**0,16**		60	449,619 ± 0,134	56,81	0,95
	120	2,375 ± 0,021	9,20	0,08		120	258,625 ± 0,371	98,77	0,82
	180	2,718 ± 0,032	8,04	0,04		180	198,065 ± 0,760	128,97	0,72
	240	3,261 ± 0,062	6,70	0,03		**240**	**174,85 ± 0,171**	**146,09**	**0,61**

The MADA implementation using TBB obtained the best values of computational efficiency of all the variants analyzed in this article. This is reflected in the execution time when the number of thread increase: when using 240 threads to process instance #6, MADA TBB demanded 174.85 s to execute, the lowest execution time for all compared algorithms.

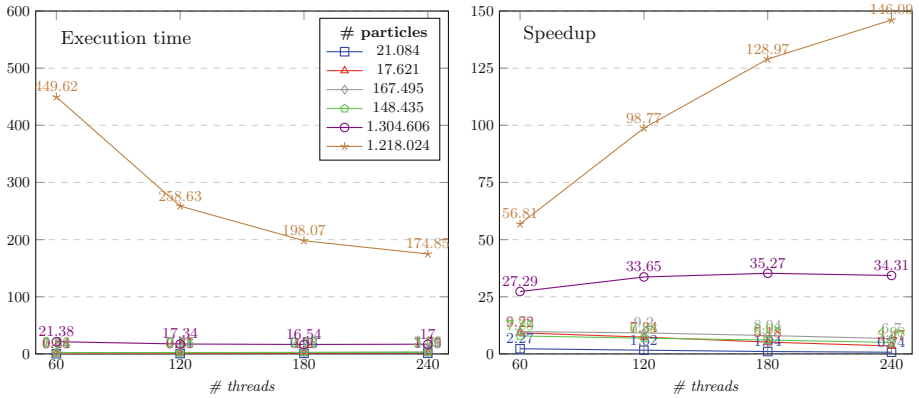

Fig. 5. Performance results: MADA TBB implementation

The TBB implementation improved 5.16% the execution time in the largest instance, mainly due to the fact that the cost of creating threads in TBB is higher than in Cilk Plus. Execution times using TBB are better when the size of the problem is large enough.

5.5 Results Discussion

Experimental results showed that using the Intel Xeon Phi architecture significantly reduces the execution times of self-gravity calculation. Performance results indicated that the using automatic vectorization allowed obtaining better results than those obtained by explicit vectorization with array notation (improvements of up to 12.37% were obtained). On the other hand, using TBB as a thread manager/scheduler instead of Intel Cilk Plus improved the execution time up to 5.16% for largest instance using 240 threads and automatic vectorization. MADA TBB demanded 174.85 s to execute, corresponding to a computational efficiency of 0.61.

6 Conclusions and Future Work

This article analyzed several parallel implementations of MADA to calculate self-gravity on astronomical systems composed of millions of particles using the Intel Xeon Phi architecture. Specific optimizations were studied regarding thread

management (Cilk Plus and TBB), data structures (SoA, AoS, and array notation), and vectorization options (automatic, explicit).

The experimental analysis was performed on six problem instances that model different small astronomical bodies with diverse features: number of particles (up to 1.2 million), particles radius, and subdivisions for MADA calculation. Instances with were considered. The execution time of each studied implementation were evaluated and compared using configurations of 1, 60, 120, 180, and 240 threads for each instance.

Performance results showed that using Xeon Phi significantly reduces the execution times of self-gravity computation. For the most complex instance, the best execution time of 174.85 s was obtained using 240 threads, automatic vectorization, and TBB as thread manager/scheduler. Using automatic vectorization yielded better results than those obtained by explicit vectorization with array notation. Using TBB as thread manager/scheduler improved over the implementation using Cilk Plus in 5.16% for the most complex instance.

The results obtained in the analysis clearly show the potential of parallel computing using the Intel Xeon Phi architecture for efficiently solving complex scientific computing problems, such as simulations of small astronomical bodies.

The main lines for future work are related to study the efficiency of the proposed implementations on modern Xeon Phi versions, studying bottlenecks of I/O operations, and analyze the impact of using offload mode for execution.

References

1. Harris, A., Fahnestock, E., Pravec, P.: On the shapes and spins of "rubble pile" asteroids. Icarus **199**(2), 310–318 (2009)
2. Haile, J.: Molecular Dynamics Simulation: Elementary Methods. John Wiley & Sons Inc., New York (1992)
3. Cundall, P., Strack, O.: A discrete numerical model for granular assemblies. Géotechnique **29**(1), 47–65 (1979)
4. Hager, G., Wellein, G.: Introduction to High Performance Computing for Scientists and Engineers. CRC Press, Boca Raton (2010)
5. Frascarelli, D., Nesmachnow, S., Tancredi, G.: High-performance computing of self-gravity for small solar system bodies. Computer **47**(9), 34–39 (2014)
6. Nesmachnow, S., Frascarelli, D., Tancredi, G.: A parallel multithreading algorithm for self-gravity calculation on agglomerates. In: Gitler, I., Klapp, J. (eds.) ISUM 2015. CCIS, vol. 595, pp. 311–325. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-32243-8_22
7. Rocchetti, N., Frascarelli, D., Nesmachnow, S., Tancredi, G.: Performance improvements of a parallel multithreading self-gravity algorithm. In: Mocskos, E., Nesmachnow, S. (eds.) CARLA 2017. CCIS, vol. 796, pp. 291–306. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-73353-1_21
8. Rönnbäck, E.: Parallel implementation of the projected Gauss-Seidel method on the Intel Xeon Phi processor-application to granular matter simulation. Master Thesis ID: diva2:747201. Umeå University, Sweden (2014)
9. Surmin, I., Bastrakov, S., Gonoskov, A., Efimenko, E.S., Meyerov, I.: Particle-in-cell plasma simulation using Intel Xeon Phi coprocessors. Vychislitel'nye Metody i Programmirovanie **15**(3), 530–536 (2014)

10. Pennycook, S., Hughes, C., Smelyanskiy, M., Jarvis, S.: Exploring SIMD for molecular dynamics, using Intel® Xeon® processors and Intel® Xeon Phi coprocessors. In: 27th International Symposium on Parallel & Distributed Processing, pp. 1085–1097 (2013)
11. Sandia National Laboratories. Mantevo Project (2017). https://mantevo.org/. Accessed March 2018
12. Kothari, B., Claypool, M.: Pthreads performance. Technical Report WPI-CS-TR-99-11. Worcester Polytechnic (1999)
13. Ajkunic, E., Fatkic, H., Omerovic, E., Talic, K., Nosovic, N.: A comparison of five parallel programming models for C++. In: 35th International Convention on Information and Communication Technology, Electronics and Microelectronics, pp. 1780–1784 (2012)
14. Leist, A., Gilman, A.: A comparative analysis of parallel programming models for C++. In: 9th International Multi-conference on Computing in the Global Information Technology, pp. 121–127 (2014)
15. Nesmachnow, S.: 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 (2010)