An approach to the n-body problem solution using the particle-particle scenario in Multi and Many-cores

Cristián D. Maureira Fredes
cmaureir@csrg.cl

***Abstract*—In the present work, is described a parallel programming comparison, between Many and Multi core scenario, focused on the widely know *n-body* problem. We present an state of art of the main approaches to the problem force calculation and finally we show the obtained results of the three different parallel programming approaches with the pros and cons of each implementation.**

Key words: Parallel Algorithms, OpenMP, Pthreads, CUDA, n-body

I. Introduction

The $n-body$ problem is widely used in different investigation related to the formation and evolution of several other problems, like the planetary clusters, star clusters, galaxy clustering, back holes, universe large structure formation, etc.

The main idea of the $n-body$ problem is to simulate the motion of a certain particle number, interacting gravitationally between each other by a force caused by other bodies. So, looking the motion of stars and planets, the gravity is the main character of this phenomena. This movement is defined by some differential equations proposed in first instance by Newton.

The *n-body* problem is the problem of predict the movement of a particles set of celestial objects, which interact with each other due to the gravitational force.

Each celestial object will have a determinate *mass* (m) and a specific location in 3D space (x, y, z), in addition to an initial velocity determined in each direction (v_x, v_y, v_z) and finally will have some initial zero acceleration (a_x, a_y, a_z). It should be noted that the *position*, the *velocity* and *acceleration* will be updated depending on the interaction with other bodies which follow the gravitational force formula, which is defined by:

$$f_{ij} = G \cdot \frac{m_i \cdot m_j}{||r_{ij}||^2} \cdot \frac{r_{ij}}{||r_i j||} \tag{1}$$

where the initial position are x_i, the velocities are v_i, having a i, between values, $1 \leq i \leq N$, the i and j bodies mass is determined by m_i and m_j where $r_{ij} = (x_j - x_i)$ is the distance vector between the i and j bodies and then G, gravitational constant. $(6.67428 \times 10^{-11} m^3 kg^{-1} s^{-2})$

A classic example of what is the *n-body* problem is the planets movement in our solar system, which is affected by the Sun properties and as all the other bodies that are between their orbits.

The *n-body* problem interest factor is that numerical algorithms are developed to solve the dynamics of this problem can be applied not only in the astrophysics area, with fields such as *Celestial mechanics*, *Dense stellar systems*, *Sphere of Influence of a massive BH*, and *Galaxy dynamics and cosmology*, but it also is widely used in the fluid dynamics field, which for example may be reflected in the work of Gingold et al. [3] which takes as its main objective a method of resolution of physical models, more than the same particle interaction.

Another important area in fluid dynamics are the *Vortex Methods*, which are a technique for various turbulent flows simulation of a particular fluid. However, applications are not only in theory since, for example, have been made simulations of smoke in real time for use in the video games development [5].

Solving the *n-body* problem consists of three key issues that will determine the calculation difficulty.

First, for this problem is required an initial scenario so there are alternatives, which can be randomly generated positions, which itself carries a problem, or take existing models to generate a baseline, as *Plummer model* although unrealistic, attempts to deliver a distributed bodies scenario based on the density of a certain system.

Another important aspect is that as we work with the bodies interaction, we need to use a method to integrate the motion equations, in order to update *position*, *speed* and *acceleration*.

This is where various methods widely known have been used, such as *Forward and Backward Euler* which are not recommended due to low precision. From that point that other methods have improved the operation of the motion equations, such as *Leapfrog integration*. Finally, the recommended methods are *Runge Kutta* and *Two-stepAdams Bashworth* which have a much better accuracy than previous methods, but are highest level of calculation.

Finally, the most important point to this problem is the force calculation between the particles, being the main focus of the present work, because if we consider the exerted force on a particle is determined by all other particles in a given system, the order of the algorithm grows quadratically by increasing the amount of particles, matter by which various researchers have conducted techniques to reduce the order of $O(n^2)$ it holds in its initial version, taking an approach called *Particle-Particle*.

II. N-BODY DETAILS

The Particle-Particle (PP) method is the simplest wat to address the task of calculating the exerted force by all the bodies on a single body.

Broadly, the method consists of:

- Collect forces on a given particle (using the previous indicated formula).
- Integrate the motion equations.
- Update the counter.
- Repeat the process.

Note that the process in which the equations are solved, consists of two first order differential equations, to calculate the acceleration and speed, and also uses an integration method for new positions and velocities of the bodies.

As this is the simplest scenario, we realize that the process is of order $O(n^2)$, so for any algorithm, not a desirable scenario.

Since this initial solution, completely theoretical, without any improvement, any text that refers integration methods will be useful, but is also convenient to use simulation references, as is the work of Gould and Tabochnik [4].

This work will consider only an implementation of the PP method, but is important to clarify that there was other approximations more efficient like the Particle-Method (PM) method, which present a to $O(n + ng \log ng)$ order, (ng: amount of the used vertex); with $O(n \log n)$ algorithm order. Other algorithm, is the model proposed by Barnes & Hut [1] in 1986, called Tree Code, offering a $O(n \log n)$ algorithm for the force calculation. To the point reached by the TC algorithms, which have been studied to obtain a good performance on GPU clusters, making it a typical test case in order to obtain excellent performances, allowing to have been winners of several awards, as the case Hamada et al. [7], which was awarded the Gordon Bell prize [1] several times, obtaining in one of his last work 190 TFlops using TC algorithms to solve the *n-body* problem. Finally, the *Fast Multipole Method* is one of the methods mostly used, for its low complexity and provides high accuracy, being proposed by Leslie Greengard in his doctoral thesis [6], this algorithm is known to propose an order usually of $O(n)$.

III. ALGORITHM APPROACH

The used algorithm is the more simplest way to implement a *Particle-Particle* approach to the *n-body* problem, i.e. we will have a main loop, with a certain time-steps, and inside we will update the particles attributes, like the "position", "acceleration" and "velocities".

The main characteristics, is that is a fine-grained algorithm, it means, perform a lot of small mathematical calculation, several times, instead of a big task.

In pseudo-code, the algorithm idea will be as follows:

$updateAccelerations()$
for $counter = 0 \rightarrow iterations$ **do**
  $updatePositions()$
  $updateAccelerations()$
  $updateVelocities()$
end for

[1] http://awards.acm.org/bell/

After a serial implementation profiling, we can note that almost all the execution time (98%) is in the *updateAccelerations()* function, because inside we have a line which use a `math.h` function called *sqrt()*.

So the main goal will be optimize that function, to optimize the execution time of our *n-body* problem implementation.

A. OpenMP Implementation

The main idea behind the OpenMP implementation is to use the most important characteristic of this approach, the simplicity.

When we use OpenMP, we are looking for a quick optimization to our code, in this example, after the profiling, we note that all the execution time was in the `updateAcceleration()` function, so we need to optimize that code portion.

So, the simplest way to optimize a *for* loop, will be to add a `#pragma omp parallel for`, and works, but in this case, we have a nested *for* loop, inside another loop, so the previous pragma will not work.

One of the OpenMP characteristics to solve this issue, is to privatize variables, that can be done in different ways, for example:

- `private(i)`: A *private* variable means that the value is not dependent on any other thread, so each thread will have a copy of the variable. The internal behaviour is:
  - Create new object of the same type in each thread,
  - The references to the original object are replaces by the new one,
  - We assume that the variable are uninitialized,
- `firstprivate(i)`: A *firstprivate* variable is very similar to the *private* variable, but provides an automatic initialization of the variables, according to the original value, before enter in the parallel section.
- `lastprivate(i)`: A *lastprivate* variable is very similar to the *firstprivate* but the value of the variable is from the last loop iteration. So the last value is copied back into the original variable.

In this case, we use the `private` OpenMP variable to the second counter (j), because we need to each i-thread has a self j counter to calculate the force on a single body.

Finally, the code was programmed as follows:

```
void updateAccelerations(){
    int i,j;
    for (i = 0; i < n; i++) {
        bodies[i].ax = bodies[i].ay = bodies[i].az = 0;
    }
    #pragma omp parallel for private(j)
    for (i = 0; i < n; i++) {
        for(j=0; j<n; j++){
            if(j != i){
                accelerationCalc(i,j);
            }
        }
    }
}
```

You can note, that the only modification is only the pragma line, and internally, OpenMP will distribute and synchronize

all the threads in the CPU's of the computers, which is good is we are looking for quick programming optimization, but not enough good if we like to handle the synchronization by our-self, avoiding all the possible overhead of the threads call.

We do not use critical sections, because we do not have shared variables in each iteration, or loop dependencies.

B. Pthreads Implementation

Using POSIX threads, we have more low level control to the threads itself, instead the OpenMP approach.

In this case, we write the code thinking in obtain the best results, using the CPU in a very efficient way. This can be possible, because we perform an operation between the amount of bodies and the number of cores, so we distribute equally in each core a set of bodies, in other words, we are change the algorithm grain, from a "fine" to "coarse", grouping bodies calculations.

As the calculation of the force on every body depends of all the others bodies, we do not have a calculation dependence, because we modify only the acceleration using the position and mass of the other bodies, so is not necessary to use `mutex` to avoid data corruption or errors.

```
void updateAccelerations(){
    int i;

    for (i = 0; i < n; i++) {
        bodies[i].ax = bodies[i].ay = bodies[i].az = 0;
    }
    int bound = n/cores;
    for (i = 0; i < cores; i++) {
        threads[i].ini = i * bound;
        threads[i].end = threads[i].ini + bound;
        pthread_create(&threads[i].thread, NULL ,
             accelerationCalc, (void *)&threads[i]);
    }

    for(int j=0 ; j < cores ; ++j){
        pthread_join(threads[j].thread, NULL);
    }

}
```

The previous Pthreads implementation follows three simple steps:

- Calculate the bodies bound (per each core),
- Execute the *accelerationsCalc()* function, per each thread (considering an amount of bodies).
- Join all the threads, and wait for his end.

Finally, we obtain a little more complicated approach to the parallelism, having more control to the threads synchronization and execution, but we are taking advantage of OpenMP, because the change from *fine-grained* to *coarse-grained* because is more appropriate to a CPU parallelization approach, to use the most resources that we can obtain for each core.

C. CUDA Implementation

The GPU based implementation, using CUDA, can be more complicated because is in essence C++, but with some extra statements to handle the differentiation between *Device* and *Host* functions, which execute into the GPU and in the CPU respectively.

There are several research arround this direct method implementation using CUDA, like the work of Belleman et al. [2], obtained good results, also, people from Nvidia use this problem as a toy algorithm to try the power of GPU, an example can be found in [8].

The main idea of the implementation is the same, but the only difference is that we have 240 cores (1.3 GHz), instead of the 16 cores in the CPU (2.27 Ghz), which give us a great scenario for a fine-grained algorithm.

The main kernel call is very simple, besides the number of *blocks per grid* and the number of *threads per block*, we need the main bodies array (`bodies`), the iteration variable (`iter`), the time step in each iteration (`dt`) and the total of bodies (`N`).

```
nbody<<< blockNum, threadsPerBlock >>> (bodies,iter,dt,N);
```

The kernel implementation, is based on the idea of separate each function in a different *device function*, so we have the need to synchronize all the used threads after each device function call.

The following code, represent the CUDA implementation based on the main pseudo-code idea.

```
__global__ void nbody(particle *bodies,float iter, float dt
     , int N){
    reset_accelerations(bodies,N);
    __syncthreads();
    update_accelerations(bodies, N);
    __syncthreads();

    for (float t = 0.0; t < iter; t += dt){

        update_positions(bodies,dt,N);
        __syncthreads();

        reset_accelerations(bodies,N);
        __syncthreads();

        update_accelerations(bodies,N);
        __syncthreads();

        update_velocities(bodies,dt,N);
        __syncthreads();
    }
```

We take advantage with this implementation, because the GPU is physically designed to work with several small calculation in all the cores, base idea of the graphical computation.

We use the idea of, *one-thread* means *one-body*, which is different to the previous implementations.

IV. Experimental Results

A. Hardware and Software Configuration

The details of the computer were the test was performed, as the following.

- CPU: 16 x Intel(R) Xeon(R) CPU E5520 @ 2.27GHz
- RAM: 12 GB
- Graphic Card: 2 x nVidia Tesla C1060.
  - Number of GPUs: 1
  - Number of GPU cores: 240
  - Cores frequency: 1.3 GHz
  - Memory: 4GB
- OS: Scientific Linux SL release 5.6 (Boron)
- Kernel: 2.6.18-238.12.1.el5 x86_64 GNU/Linux

B. Execution Set-up

To the OpenMP and Pthreads implementation we consider one node with the previous technical details, and the benchmark was performed using the following combinations:

Number of cores:	1	2	4	6	8	10	12	14	16
Number of bodies:	16	32	64	128	256	512	1024	2048	4096

For each number of cores, we perform test with all the number of bodies combinations. Each test, was execute 20 times, to ensure an approximated real value.

In the other hand, the test on the GPU was performing modifying the number of block per grid (BpG) and the number of threads per block (TpB), with the constraint that the warp launch set of 32 threads, so to do not waste threads, we use threads between 32 and 512, so for each amount of bodies, we divide it, in multiples of 32 (i.e., for 1024 bodies we use, (2 BpG, 512 TpB), (4 BpG, 256 TpB), (8 BpG, 128 TpB), (16 BpG, 64 TpB) and (32 BpG, 32 TpB))

C. Assessment on Parallel Performance

We consider the speed-up to evaluate the performance in each implementation according to the following formula:

$$Speed\ up\ =\ \frac{Serial\ Time}{Parallel\ Time} \tag{2}$$

1) OpenMP assessment: In the figure 1 we can see the most representative speed-up of some amount of bodies.

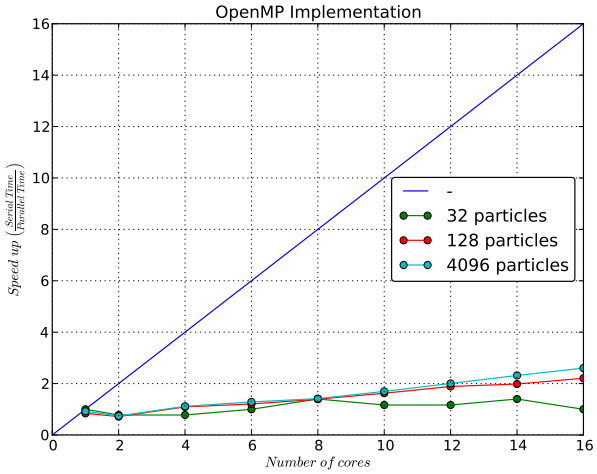


Fig. 1. OpenMP implementation. Speed-up of 32, 128 and 4096 bodies.

The difference between the amount of bodies is very big, and as the figure 1 shows the behaviour in speed-up terms, is very similar, which represents that the OpenMP implementation works in the same way, almost independent of the amount of bodies, which is not a good scenario in this simulation, because it is expected an evolution considering different bodies amount.

We can not forget that the additional programming with OpenMP is almost nothing, only one pragma line, and it is possible to obtain a x3 speed-up.

2) Pthreads assessment: In the figure 2 we can see the most representative speed-up of some amount of bodies.

In this case, we can note that the normal evolution of the speed-up considering an increase in the amount of bodies, because with more bodies we can give more work to each thread, without wasting so much resources. It is important to

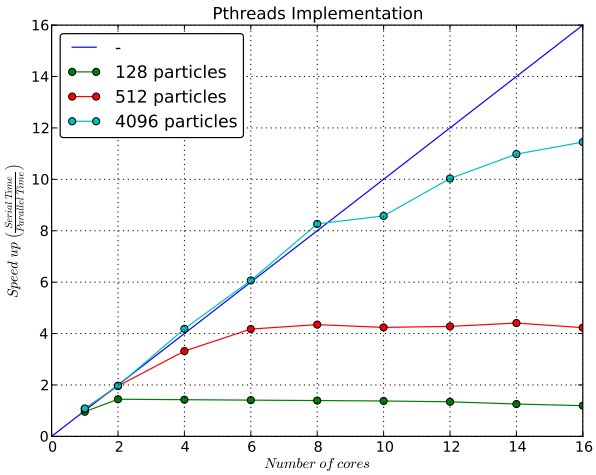


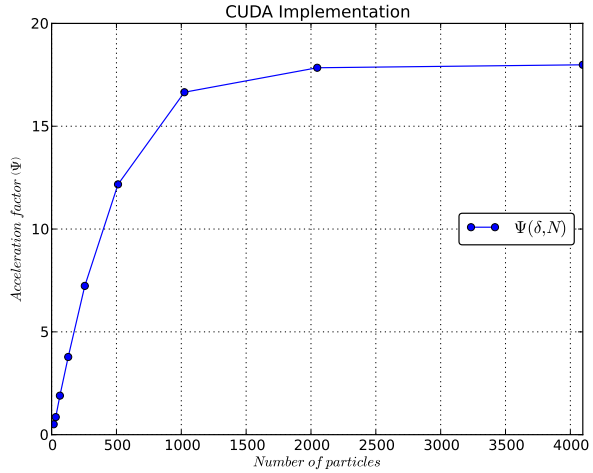
Fig. 2. Pthreads implementation. Speed-up of 128, 512 and 4096 bodies.

remember that in this implementation, there is a little trick gathering $\frac{n}{c}$ bodies per thread (n: total bodies, c: total cores).

Another important issue, is that the synchronization is performed by the program itself, in difference with the pre-compilation process used by OpenMP.

Note that the behavior with a little amount of bodies is very similar, and in both represented cases we can see an asymptotic behavior, with more than 6 cores.

3) CUDA assessment: In the figure IV-C3 we can see the acceleration factor of all the performed test, using different numbers of bodies.



This case can be seen as a different implementation, considering the huge difference between the CPU cores and the GPU cores, following the previous idea, we present an Acceleration Factor to represent the speed-up on GPU,

$$\Psi(\delta, N) = \frac{t_s}{t_{p_{(\delta)}}} \tag{3}$$

the main idea is the same, serial time versus parallel time, but in each GPU iteration we use a configuration (δ) between the *blocks per grid (BpG)* and the *threads per block (TpB)*, aside of the fixed number of GPU cores.

In the figure IV-C3 we can appreciate the behavior of the CUDA implementation increasing the amount of bodies, obtaining a really good acceleration factor, but after the 2000 bodies, it is possible to note a asymptotic behavior, which shows an issue in the implementation, breaking the initial good scalability.

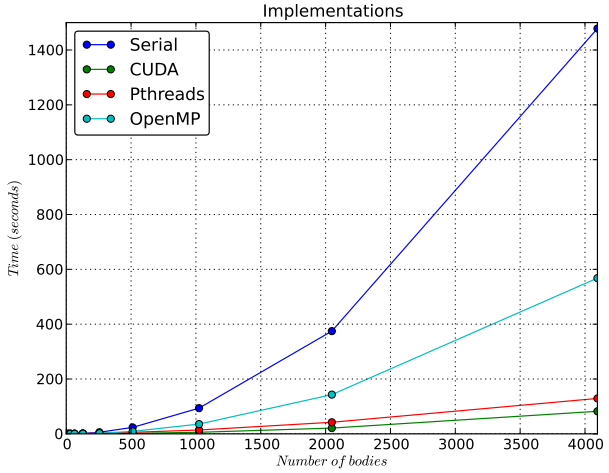


Fig. 3. Best execution time of each implementation.

Finally, as the figure 3 and the table IV-C3 shows, the final review of each technique is very clear.

Understanding the differences between the CPU and GPU programming paradigms, is good to consider both alternatives at the time to solve problems with a large computational work.

Implementation	Speed-up
Serial	1x
OpenMP	3x
Pthreads	11x
CUDA	18x

TABLE I
FINAL BEST RESULTS OF THE PERFORMED TESTS

Please note, that not always the CUDA implementation will be the best, this happens only because this algorithm, is *fine-grained*, and the CPU are a best choice for *coarse-grained* algorithms, which is affirmed by the Pthreads results, because the transformation in the code, gathering bodies to give more work to the CPU, changing from fine to coarse grain.

V. Conclusions and Future Work

In the present work, we describe a benchmark of the particle-particle n-body algorithm, using different many and multi-core scenarios, with different techniques to take some advantage in each case.

The previous work, showed a more programming vision of the worst computational method of the *n-body* problem, considering only a high speed-up as a main issue, without think in change the other important aspects of this problem, like the integration method and the initial population generation.

Most of the done work, take only one or two computational approach, considering CPU or GPU cores to perform the calculation, but it is very important the fact that a good programming in CPU can reach not the entire GPU speed-up, but a good approximation, modifying the behind idea of the CPU implementation, because in some implementation, the programmer forgot the strength of a CPU-thread, which compared to a GPU-thread is more computationally strong.

Finally, is important to note that this three implementations are basic ones, so, it is very possible to start a research to improve the code, specially the CUDA implementation, because with some better manipulation of the GPU memory, we could obtain more than the double of the obtained speed-up.

References

[1] Barnes, J., Hut, P.: A hierarchical o(n log n) force-calculation algorithm. Nature 324, 446 - 449 (1986)

[2] Belleman, R.G., Bédorf, J., Zwart, S.F.P.: High performance direct gravitational n-body simulations on graphics processing units ii: An implementation in cuda. New Astronomy 13(2), 103 – 112 (2008), http://www.sciencedirect.com/science/article/pii/S1384107607000760

[3] Gingold, R.A., Monaghan, J.J.: Smoothed particle hydrodynamics - theory and application to non-spherical stars. Royal Astronomical Society, Monthly Notices, vol. 181, Nov. 1977, p. 375-389.

[4] Gould, H., Toochnik, J.: An Introduction to Computer Simulation Methods (1988)

[5] Gourlay, M.J.: Fluid simulation for video games. Intel Software Network (2009)

[6] Greendard, L.: The rapid evaluation of potential fields in particles systems. Ph.D. thesis, Yale University, New Haven, CT (1987)

[7] Hamada, T., Nitadori, K.: 190 tflops astrophysical n-body simulation on a cluster of gpus. In: Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis. pp. 1–9. SC '10, IEEE Computer Society, Washington, DC, USA (2010), http://dx.doi.org/10.1109/SC.2010.1

[8] L. Nyland, M. Harris, J.P.: Fast n-body simulation with cuda. GPU Gems 3. H. Nguyen, ed. Addison-Wesley (2007)