SEQUENTIAL OPTIMIZATION AND SHARED AND DISTRIBUTED MEMORY PARALLELIZATION IN CLUSTERS: N-BODY/PARTICLE SIMULATION

Fernando G. Tinetti[1,2], Sergio M. Martin[3]
[1]III-LIDI, Fac. de Informática, UNLP
Calle 50 y 120, 1900, La Plata, Argentina
[2]Comisión de Inv. Científicas de la Prov. de Bs. As.
[3]Universidad Nacional de La Matanza
Florencio Varela 1903 - San Justo, Argentina
fernando@info.unlp.edu.ar, smartin@unlam.edu.ar

ABSTRACT
The particle-particle method for N-Body problems is one of the most commonly used methods in computer driven physics simulation. These algorithms are, in general, very simple to design and code, and highly parallelizable. In this article, we present the most important approaches for the application of the three performance improvement areas on these algorithms when executed on high performance computing (HPC) clusters: 1) sequential optimization (a single core in a node of the cluster), 2) shared memory parallelism (in a single node with multiple CPUs available, just like a multiprocessor), and 3) distributed memory parallelism (in the whole cluster). For each one of the improvement areas we present the employed techniques and the obtained performance gain. Also, we will show how some (sequential/classical) code optimizations are almost essential for obtaining at least acceptable parallel performance and scalability.

KEY WORDS
High Performance Computing, Source code optimization, Parallel Computing, Cluster Computing, N-Body/Particle Simulation.

1. Introduction

The particle-particle (PP) method for N-Body problems is simple as well as highly parallelizable. Also, it provides a representative test bed for analyzing performance impacts of optimization and parallelization approaches on applications performance and scalability in large scale simulation contexts. We will explore the three basic performance improvement methods for N-Body problems using cluster computing:

- Sequential Code Optimization
- Intra-Node Parallelism using shared memory
- Multi-Node Parallelism using message passing

The N-bodies problem has been one of the biggest challenges for mathematiciansfor the last centuries[6]. A purely numerical solution that turns to a result in a linear number of operations for any N, any time lapse, and taking the possibility of collisions into account has not yet been found. Therefore, the only way to approximate to a real solution is to use a differential method with tiny time slices along with computational processing

In this article, we use a gravitational calculation problem where all bodies with mass are attracted to each other. Also, each body is considered a small particle unable to collide with each other.The force applied to a particular body in a particular moment depends on the masses and distances of the other bodies relative to him, as shown in Eq. (1).

$$F_i = \sum_{i \neq j}^{n} G \frac{m_i m_j}{\|r_j - r_i\|^3} (r_j - r_i) \qquad (1)$$

Where G is the gravitational constant, m_i and m_j are the masses of bodies i and j, and r_i and r_j are the three-dimensional vectors describing the positions of each one.The PP method for this kind of problems is one of the simplest to design and code. We use itto calculate the total gravitational force applied to each particle and then apply that force as a constant to calculate the changes in acceleration, velocity, and position of the body using a time differential. After all these changes are applied for each body, the simulation time increases with the value of the time differential, and a new step is executed.

These simulations diverge from the real case as the simulation steps are executed because they take each force as a constant during the time differential. Knowing that forces in nature vary constantly, these may be different from the simulated lapse of the time differential, where they are considered as constants. There are other simulation methods, different ways of simulation error control, and numerical optimizations [3][13][5], but we are going to use the basic method for the analysis of different optimization and parallelization approaches.

Clusters of computers have become standard for scientific parallel computing since many years ago. In these platforms, there are several choices for optimization and parallelization. Our objective is the analysis and, more

specifically, the quantification of possible performance gains at three levels: sequential (one core) source code optimization, multiprocessing or shared memory (one node), and multicomputing (the cluster) or distributed memory parallel computing .

Results and quantified improvement for original and optimized codes for each environment will be shown as they are presented, as well as a comparison of methods.The definition of cluster that we are using in this article refers to a set of computers interconnected by a standard local area or high-speed network. This includes a wide range of possibilities: from commercial or scientific supercomputers belonging to the TOP500 Supercomputer listto home-made clusters with PCs and a wireless router. Each individual computer in the cluster will be referred to as a node.

The rest of this article is organized as follows: Section 2 introduces the initial non optimized algorithm. The current hardware used for HPC which should be taken into account for optimizations as well as paralellization is briefly explained in Section 3. The most successful sequential optimizations are shown in Section 4. Section 5 explains the parallelization for (shared memory) multiprocessing environments via OpenMP (i.e. with several threads of execution) and its relationship with memory access optimization. The parallelization for distributed memory environments, usually clusters, via MPI (Message Passing Interface) is introduced in Section 6. Section 7 explains several details related to performance from the point of view of optimization and parallelization. Finally, Section 8 includes conclusions and further work.

2. Initial Algorithm, the Departure Point

The basic PP algorithm can be defined in pseudo-code as shown in Fig. 1, where

- vec is an array of structures, and each structure (an array element) fully represents one of the bodies in the problem.
- "Compute distance between i and j" implies using x, y, and z coordinates of bodies i and j.
- apply_time_step(vec) is devoted to tasks like velocity and position updates as well as to properly advance the simulated time.

The computing time (complexity) of this algorithm grows in quadratic order for every new body added to the simulation. From the point of view of the simulated time, the computing time grows linearly for each further time step of calculation. This means that if one time step of n bodies takes a certain amount of time, two time steps of the same number of bodies will take twice the time. A priori, no parallelism can be applied through different time steps since all of the data depend on the previous results. Therefore, we will focus on performance and parallelization enhancements within one time step, for a *representativenumber* of bodies. Here, a *representative number* is defined from the point of view of performance, i.e. one that fulfills the following:

- Every data is on main memory, with no need to use out-of-core/swap/disk memory for storage.
- The whole data does not fit in cache memory, so performance is not automatically optimized by every access being supplied by cache at any level/s (L1, L2, or, eventually, L3).

```
initialize_bodies_data(vec)
For each time step Do
For each i in vecDo
 For each j in vec Do
     If (i≠ j) Then
       Compute distance between i and j
       Update acceleration of i
     End If
  End For
End For
apply_time_step(vec)
End For
```

Figure 1: Pseudo-code of the Basic N-Body Simulation.

As explained above, Fig. 1 shows in bold face the code to be analyzed and optimized from the point of view of performance. Every floating point operation is related to one of the two tasks identified, respectively, as

- Compute distance between bodies i and j.
- Update acceleration of i (taking into account the influence of body j).

And Fig. 2 shows the pseudo-code of computations required for computing the distance between i and j,

$$d\_x = vec[j].x - vec[i].x$$
$$d\_y = vec[j].y - vec[i].y$$
$$d\_z = vec[j].z - vec[i].z$$
$$ab\_dc = (sqrt(d\_x^2 + d\_y^2 + d\_z^2))^3$$

Figure 2: Computing Distance Between Bodies i and j.

according to Eq. (1) above, where

- d_[x/y/z] are the distances in each of the three corresponding dimensions.
- sqrt() is the function used for computing the square root of a number.
- ab_dc is the cube of the absolute distance between bodies i and j.

At the same level of abstraction of Fig. 2, Fig. 3 shows the the pseudo-code of computations required for updating the acceleration of i (taking into account the influence ofbody j), where

- vec[i].a_[x/y/z] represents the acceleration in each of the three corresponding dimensions of body i.

- vec[i].m represents the mass of body i.
- Δt and G are the time step and gravitational constant respectively.

```
vec[i].a_x+=d_x*Δt*G*vec[j].m / ab_dc
vec[i].a_y+=d_y*Δt*G*vec[j].m / ab_dc
vec[i].a_z+=d_z*Δt*G*vec[j].m / ab_dc
```

Figure 3: Acceleration Update of Body i.

There are some immediate optimizations as well as other more elaborate ones. However, the specific figure of performance effect will depend on the hardware. Thus, the next section will introduce the hardware on which we are going to experiment with and measure performance.

3. Computing Hardware, and Clusters

Clusters are used for parallel processing since many years ago, due to their very good cost/performance ratio [11]. And the definition of clusters for parallel processing has been almost unchangeable since its definition [2][15] as a set of computers (nodes) which are interconnected by a network. There has been, of course, a *natural* evolution of the hardware involved in clusters:

- Current nodes are mostly multi-core multiprocessors, and there are several Top500 computers (and among the top 10 range in the Top500) with attached GPU (Graphical Processing Units) [16].
- Current interconnection hardware ranges from very low cost Ethernet 100 Mb/s or 1Gb/s to Infiniband to custom made interconnection networks in the range of the top 10 or 50 supercomputers in the Top500 list.

We will experiment on the most available and also lowest cost parallel computers. More specifically, we will obtain and analyze performance measurements on an Intel i5 multi-core computer and on a cluster of dual quad-core Xeon processors. We will refer to these platforms as mc (since a computer with an Intel i5 is *just* a multi-core computer) and cluster respectively. Table 1 shows the characteristics of the mc platform, and Table 2 shows the characteristics of the cluster.

Both platforms (mc and cluster) provide multiprocessing facilities, but only the cluster is able to be used for distributed memory parallel computing. As expected, the cluster is more scalable in terms of the number of processors (cores) and memory. As turns out from Table 1 and Table 2, there are many similarities in a single node of most of the current HPC clusters:

- Multiprocessing via multi-core, i.e. several cores with shared access to a single memory.
- Memory hierarchy including several cache levels (at least L1 and L2) in the processor.
- Single cores running at GHz frequency, including pipelining and floating point units and, most of the times, with superscalar capabilities.

And these similarities are also found in clusters made up with AMD processors (e.g. those based on Opteron or Phenom processors).

Table 1: mc Platform Characteristics

Processor	Intel i5-2310
Cores/processor	4
RAM	4 GB
OS, gcc v.	Linux 3.0.0, gcc 4.6.1
# Nodes	1

Table 2: Cluster Platform Characteristics

Processor	2 x Intel Xeon E5405 per node
Cores/processor	4 (8 cores per node)
RAM/node	2 GB
OS	Linux 2.6.31, gcc 4.6.3
# Nodes	4
Network	Ethernet 100 Mb/s

4. Code Optimization

Traditionally, sequential algorithm programmers had only to worry about optimizing their code in order to reduce runtime using well known techniques [4][14][8]. Also, many of the classical optimization techniques have influenced compiler design [1] and current (optimizing) compilers development. However, with the dawn of parallel computing, programmers had (and still have) to adapt to a whole new paradigm or, more specifically, now the code has to take into account the underlying parallel hardware. Furthermore, some optimization techniques may have impact on code parallelization.

Taking into account that some of the optimizations are architecture-dependent doesnot necessarily make them less generally usable. Several node and cluster optimization and parallelization concepts can be applied on most of the algorithms running in production/scientific computers and clusters. We will go through several optimizations applied to the initial algorithm version in order to obtain the best possible sequential version, as shown in subsections below.

4.1 Operator Strength Reduction

The idea behind the operator strength reduction optimization is that some operators are more processor-demanding than others. In particular, the division (/) operator tends to be more expensive than multiply (*) [7]. Fig. 3 above shows that there are three divisions in the initial version of the algorithm, where the divisor is always the same. These divisions can be changed to a less costly operation by multiplying by the corresponding inverse number, as shown in Fig. 4.

```
df = 1/ab_dc
vec[i].a_x+=d_x*Δt*G*vec[j].m * df
vec[i].a_y+=d_y*Δt*G*vec[j].m * df
vec[i].a_z+=d_z*Δt*G*vec[j].m * df
```

Figure 4: / Replaced by *.

According to Fig. 4, there is just one division and three multiplications. Thus, two divisions can be successfully replaced by three multiplications, which enhances performance.

4.2 Constants and Loop Invariants

The computations included in the inner loop of Fig. 1 define the hidden constant of the algorithm quadratic order of operations. Furthermore, the inner loop (as shown in Fig. 2, Fig. 3, and Fig. 4) includes repeated calculations involving constants and, also, loop invariants:

- Constants: the time differential (Δt**)** and, also, the gravitational constant (G).
- Loop invariants (variables whose content won't change, at least during the execution of the step): the body's mass (vec[j].Mass), and the inverse of the cubic distance (Dist_3).

Constant and loop invariants can be processed out of the loop or in an optimized manner. Note that some of these optimizations are can be easily carried out by almost every modern compiler while others may require complex interprocedural optimization (IPO) heuristics/techniques. Fig. 5 shows one of the most immediate source code optimizations for the code in Fig. 4 above, where

- **dtG** should be initialized with **Δt*G**.
- **dtG * vec[j].m** is computed just once and used three times by using **comm**.

Note that the last optimization is a common subexpression elimination (CSE) that is already used from Fig. 2 by computing the values of **d_x**, **d_y**, and **d_z** just once, and referencing those values where needed.

```
df = 1/ab_dc
comm = dtG * vec[j].m
vec[i].a_x+=d_x* comm * df
vec[i].a_y+=d_y*comm* df
vec[i].a_z+=d_z* comm * df
```

Figure 5: Optimization at the Inner Loop.

There are two further optimizations regarding constant values, loop invariants, and subexpression elimination:

- **vec[j].m** is set and constant for each body, so it is possible to precompute (outside the inner loop) **dtGm** as an array as **dtGm[j] = dtG * vec[j].m**
- Inside the inner loop, instead of computing **df** as shown in Fig. 5 and using the precomputed array **dtGM** as explained before, a further optimization is possible, by directly computing

```
comp = dtGm[j] / ab_dc
vec[i].a_x+=d_x* comp
vec[i].a_y+=d_y*comp
vec[i].a_z+=d_z* comp
```

And now it is possible to verify that some of these optimizations are not simple enough for an optimizing compiler.

4.3 Cache Temporal Locality Considerations

The algorithm updates the body acceleration according to its relationship with all of the others (except itself) by carrying out the operations described above. Then, given two bodies {a,b}, it is possible to compute two update operations: a' = upd(a,b), and b' = upd(b,a). From a cache temporal locality point of view, this would avoid many cache misses because all the operations involving the same data pair will be executed. Note that, even though the required iterations would be reduced to a half, the number of instructions per iteration would be duplicated. A priori, there is no reduction in the number of operations but there is an increasing cache hit ratio. However, given that more operations are carried out in every iteration, there would be more opportunities for other performance optimization techniques to be applied on more data (e.g. further possibilities for subexpression eliminations). Thus, the inner loop of the algorithm shown in pseudocode in Fig. 1 can be replaced by the loop given in Fig. 6, where

- The inner loop now starts at i+1, since each iteration implies two updates, shown with comments "update i" and "update j".
- **dtGm[i] = Δt*G * vec[j].m**isprecomputed. i.e. computed before the outer loop (e.g. immediately before the outer loop, where every constant and the body vector have the values needed for computing the values of **dtGm**.
- It is worth mentioning that **d_x**, **d_y**, **d_z**, and **ab_dc** are used for computing both, update i and update j.

```
For each j (starting from i + 1) in vec Do
d_x = vec[j].x – vec[i].x
d_y = vec[j].y – vec[i].y
d_z = vec[j].z – vec[i].z
ab_dc = (sqrt(d_x^2 + d_y^2 + d_z^2))^3

compi = dtGm[j] / ab_dc      /* update i */
   vec[i].a_x+=d_x * compi
vec[i].a_y+=d_y* compi
vec[i].a_z+=d_z * compi

compj = dtGm[i] / ab_dc      /* update j */
vec[j].a_x -=d_x * compj
vec[j].a_y -=d_y*compj
vec[j].a_z -=d_z * compj
End For
```

Figure 6: Inner loop for N-Body Computations.

4.4 Performance Optimization Results

Several experiments have been carried out in order to provide a quantitative view of the optimization techniques described above. We have taken a value for the number of bodies so that

- The data involved for calculations does not fit into any level of cache.
- The runtime is only a few hours long, so that we can run several experiments for testing. The initial runtime is near 3 hours.

Table 3 shows the performance obtained by implementing the optimizations described so far. For every run, the optimization level of the compiler has been set to -O3, and performance is given in MFlop/s.

Table 3: Sequential Experiments Performance (MFlop/s)

	mc	cluster node
-O3	1115	810
-O3 + all	2679	1859

5. Shared Memory Parallelization: OpenMP

OpenMP[10] allows for a very simple implementation of intra-node shared memory parallelism by only adding a few compiler directives. There is, however, an important detail: every thread should have a local copy of the bodies' vector, and the final results consolidation should be done within a critical region. The local copy of the vector prevents data races in the acceleration updates. Fig. 7 shows the #pragma omp added to the code,where schedule static prevents unbalanced workload.

```
#pragma omp parallel private(…) shared(…)
Copy body vector
#pragma omp for schedule(static, 1)
For each i in vecDo
For each j (starting from i + 1) in vec Do
    …
  End For
End For
#pragma omp critical
Consolidateresults
```

Figure 7: OpenMP Parallelization

Fig. 8 shows schematically the threaded execution using the OpenMP terminology. Even when the parallelization is straightforward and there are minimum synchronization points, the performance is not acceptable.Table 4 shows the performance values for 2, 4, and 8 cores in a cluster node (mc values are similar) in terms of Mflop/s and efficiency. Clearly, the scalability is far from optimal.Given the simple algorithm and our parallelization via independent data, there is an almost direct candidate for performance penalization as the number of simultaneous threads increases: memory contention. Thus, instead of changing the parallel approach we enhance memory accesses.

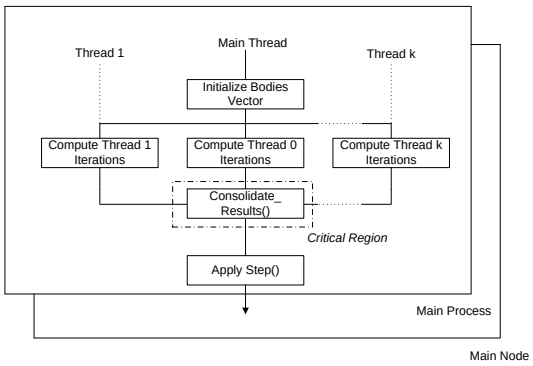

Figure 8: OpenMP Parallel Runtime.

Table 4: First OpenMP Version Performance in a Node

Cores	2	4	8
Mflop/s	2582	4403	7536
Efficiency	0.69	0.59	0.51

More specifically, we implemented a tiled version of the sequential algorithm. Table 5 and Table 6 show the performance values in mc and in a cluster node respectively for the tiled implementation.

Table 5: Tiled Performance in mc

Cores	1	2	4
Mflop/s	2806	5477	10604
Efficiency	1.0	0.98	0.94

Table 6: Tiled Performance in a Cluster Node

Cores	1	2	4	8
Mflop/s	1963	3815	7625	15232
Efficiency	1.0	0.97	0.97	0.97

The tiled implementation of the sequential algorithm has shown very interesting *properties*:

- Provides only a few Mflop/s more than the previous optimized version (compare single core performance in Table 5 and Table 6 with values in Table 3).
- Every new variable needed for the tile optimization (e.g. block size) is local to the thread in the OpenMP version. This reduces parallelization complexity.
- Clearly, scalability is enhanced (seeTable 4).

6. Distributed Memory Parallelization: MPI

MPI (Message Passing Interface) [9] provides many communication functions for distributed memory parallel computing.Also, MPI usually implies to recode at least some portions of the program so that different processes

carry out a fraction of the total workload. A parallel program with MPI can take advantage of multiple computing (CPUs/cores) resources in a multicore computer as well as in a cluster of multicore computers. From this point of view, MPI provides a uniform (referred to as homogenous) view of parallel processing regardless of running on shared or distributed memory parallel hardware. Fig. 9 shows how processes are organized for parallel computing the N-Body simulation. Most of the code is borrowed from the OpenMP version, since there is a very low data dependency in computations. It is worth noting that it is not always possible to maintain the source code almost equal from sequential to an MPI parallel implementation.

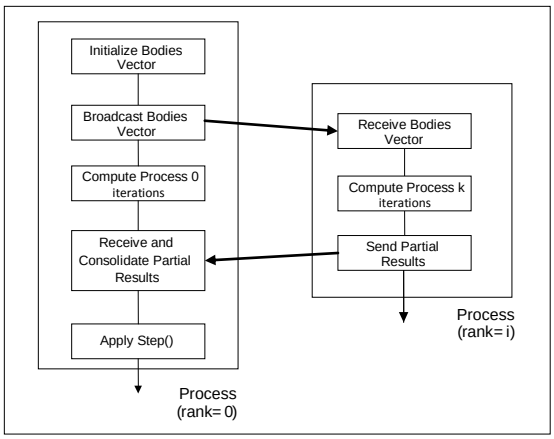

Figure 9: MPI Parallelization

As shown in Fig. 9, all processes are involved in two main communications: one for receiving the data (bodies' information) to be updated and one after the data has been updated. Both communications define a very constrained synchronization, since a simulation step cannot start *in advance*, i.e. before the previous step has been completed. Data distribution from process with rank (or id) 0 is made through a classical *collective* communication primitive: a broadcast message. Processes have been organized as master-workers, which is very usual in the context of distributed memory parallel programs. There are some optimizations specifically related to communications which have not been implemented, such as:

- Reducing communications to the variable data in the bodies array. The whole vector is always transferred among computers (some information such as the body masses are constant).
- Reducing communications to receive at process 0 only the updated data. Again, the whole array of bodies is transferred.

These optimizations can be made if the experiments show unacceptable performance penalties.

The performance (in Mflop/s and efficiency) obtained by the MPI version on mc using 2 and 4 cores is shown in Table 7. Note that performance values are almost the same as those obtained with OpenMP (shown in Table 5). Table 8 shows corresponding values on the cluster. Note that for 2, 4, and 8 cores only one node is used, while for 16 and 32 cores we use 2 and 4 nodes (the whole cluster). We did not approach a hybrid parallel implementation (i.e. one with OpenMP intra-node and MPI inter-node parallelization) because of the very good performance obtained by the MPI parallel program.

Table 7: MPI Parallel Code Performance in mc

Cores	**2**	**4**
Mflop/s	5384	10498
Efficiency	0.96	0.94

Table 8: MPI Parallel Code Performance in Cluster

Cores	**2**	**4**	**8**	**16**	**32**
Mflop/s	3921	7842	15670	31057	59805
Efficiency	1	1	1	0.99	0.95

As expected, as more nodes are used in the cluster, the performance is more penalized. However, the values obtained in the cluster show that the performance loss is very little in terms of scalability, e.g. 0.4% performance loss when doubling the number the number of nodes (from 2 to 4 nodes, i.e. 16 to 32 processes/cores).

7. Further Comments on Performance and Parallelization

Optimization techniques are usually well defined, but not always the compilers are able to identify and implement the best choices. As expected, compilers are conservative and do not always apply every possible optimization. This is confirmed once again with the results shown in this work. More specifically, Table 3 shows that source code optimizations obtain more than 142% and 151% of performance gain in mc and a cluster node respectively over that provided by a compiler.It is possible, however, that carefully selected compiler optimization switches provide better performance than that provided by -O3.

Optimizations specifically focused on reducing the memory wall performance problem are almost a constraint for obtaining *acceptable* parallel performance. As shown in Table 4 performance is not acceptable even for a small number of cores and, specifically, scalability is explicitly affected if the code does not include tile optimizations. It is worth noting that the source code that provided the performance values of Table 4 already included a cache optimization, which was explained as cache temporal locality optimization.

The OpenMP parallelization is made with some simple changes to the sequential code. Interestingly, it shows some issue(s) related to optimization that has a strong influence (penalty) on performance. The so called memory wall, defined as the (very large) difference in

performance among CPU and memory access, is made proportionally worse when more CPUs (cores) share the same memory subsystem. And the problem would be aggravated if more cores share memory (with more cores per processor or more processors sharing memory). Table 5 show that even when tiling does not provide a great improvement on performance related to the previous code version, it produces an essential improvement on performance in the context of parallel computing. And this improvement is obtained regardless of the parallel algorithm/implementation (OpenMP or MPI).

The MPI parallel algorithm obtains almost optimum performance even in the context of a shared memory parallel hardware (for which it was not necessarily defined). Unfortunately, this optimum result is not possible to be expected in every application or source code algorithm implementation. N-Body simulation is particularly well suited for parallel computing and many (most of) the problems are not so easily parallelizable. However, it has been shown in other contexts [12] that an MPI parallelization can be as good as an OpenMP parallel program. Thus, the programmer should check (experiment with) the MPI program before deciding to implement a so called hybrid approach to parallelization.

Finally, Fig. 10 and Fig. 11 show the performance improvements obtained from an original application going through (standard/sequential) optimizations and, also, different approaches to parallel computing. Distributed memory parallelization is still (a priori) the most scalable one, since multiprocessors have strong limitations in the number of CPUs (cores) sharing a single memory subsystem.

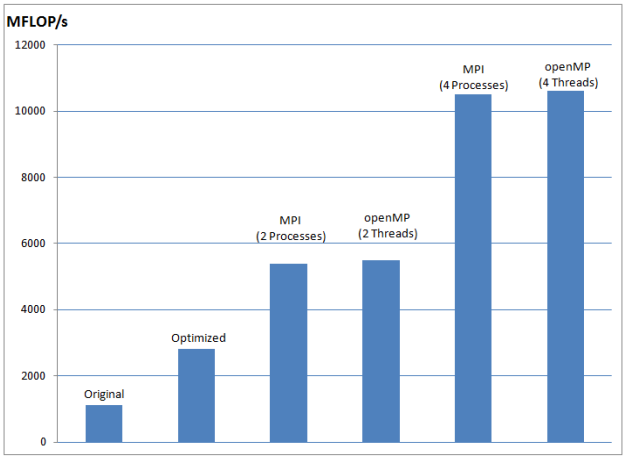

Figure 10: Summary of Performance Improvements (mc)

Performance values shown in Fig. 10 for mc are comparable with values in Fig. 11 for the cluster up to 4 processes (for MPI) or threads (for OpenMP). As expected, mc provides better performance intra-node (greater MHz, enhanced memory access). However, there are not many installed clusters based on i5 or i7 to make a general comparison, and, also, using a large number of processors (tens and hundreds of nodes).

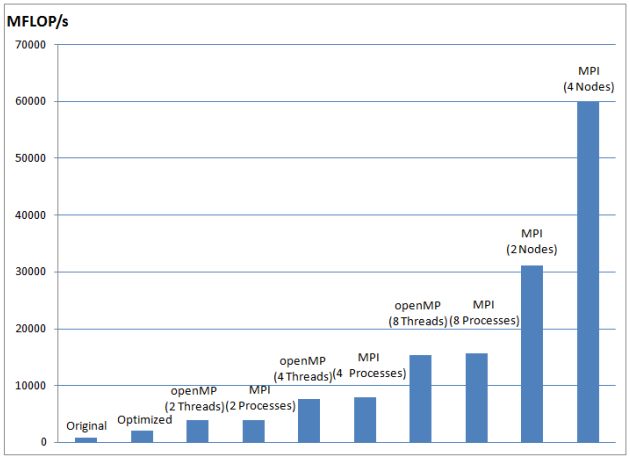

Figure 11: Summary of Performance Improvements (cluster)

8. Conclusions and Further Work

The PP method of N-Body problems holds high potential for sequential optimizations and parallel approaches. Fig. 11 above shows experimental support and data for the previous claim. Even when compilers make an excellent job in implementing optimizations, some optimizations have to be applied directly on the source code. Or, at least, the (scientific) programmer should be able to experiment and measure different source code optimizations on the source code.

Parallelization for shared memory parallel hardware is expected to be both, the most simple and less scalable parallel approach to a given code. However, when reached the limit on the scale of parallel computing, shared memory parallelization provides good (or, at least, a minimum) previous insight to distributed memory parallelization. A parallel approach initially oriented towards a distributed memory parallel platform can be perfectly appropriate for a shared memory platform.

Parallelization for distributed memory hardware is not necessarily always a problem, but it is not immediate. Some recoding seems to be necessary, even for the PP method, which is particularly simple to parallelize. As parallelization started (in this work) in the context of a shared memory environment, we already had some experience at the time of the MPI implementation.

Initially, the parallel approach should be analyzed on clusters with a large number of nodes (tens, hundreds, and thousands). Unfortunately, clusters with a large number of nodes are mainly devoted to production environments, but it would be highly beneficial if some fraction of the time they could be used for research. Further investigations should focus on the optimization of the Runge-Kutta and

Euler integration methods running on HPC clusters since their implementation may reduce the high levels of scalability required to justify network and communication overheads.

References

[1]A. V. Aho, M. S. Lam, R.Sethi, J. D. Ullman, Compilers: Principles, Techniques, and Tools, 2nd Edition, Prentice Hall, 2006.

[2] T. Anderson, D. Culler, D. Patterson, and the NOW Team, "A Case for Networks ofWorkstations: NOW", IEEE Micro, Feb. 1995.

[3]U. Ascher, L. Petzold, Computer methods for ordinary differential equations and differential-algebraic equations (Philadelphia, PA: SIAM, Society for Industrial and Applied Mathematics, 1998).

[4] J. Bilmes, K. Asanovic, C. Chin, J. Demmel, "Optimizing matrix multiply usingphipac: a portable, high-performance, ansi c coding methodology", Proceedings of theInternational Conference on Supercomputing, Vienna, Austria, July 1997, ACM SIGARC.

[5]G. J. Cooper, R. Vignesvaran, On the use of parallel processors for implicit Runge-Kutta methods, Computing, 51(2), 1993, 135-150.

[6]F. Diacu, The solution of the n-body problem, The Mathematical Intelligencer, 18(3), 1996, 66-70.

[7]T. Granlund, P. L. Montgomery, "Division by invariant integers using multiplication", ACM SIGPLAN Notices, Volume 29 Issue 6, June 1994, ACM New York, NY, USA

[8]R. Hyde, Write Great Code, Volume 2: Thinking Low-Level, Writing High-Level, No Starch Press, 2006.

[9]MPI Forum, "MPI: a message-passing interface standard", International Journal ofSupercomputer Applications, 8 (3/4), pp. 165-416, 1994.

[10]OpenMP Architecture Review Board, OpenMP Application Program Interface - Version 3.1, July 2011.Available athttp://openmp.org/wp/

[11] F. G. Tinetti, Parallel Computing in Local AreaNetworks, 2004, PhD Thesis, Universidad Autónoma de Barcelona, Spain, available at http://phdthesis.webs.com/

[12] F. G. Tinetti, G.Wolfmann,"Parallelization Analysis on Clusters of Multicore Nodes using Shared and Distributed Memory Parallel Computing Models", Proc. 2009 World Congress on Computer Science and Information Engineering, IEEE Computer Society, March 31 - April 2, 2009, Los Angeles/Anaheim, USA, ISBN 978-0-7695-3507-4/08, pp. 466-470.

[13]V. Venkatakrishnan, H. D. Simon, T. J. Barth, A MIMD implementation of a parallel Euler solver for unstructured grids, The Journal of Supercomputing, 6(2), 1992, 117-137.

[14] R. Whaley, J. Dongarra, "Automatically Tuned Linear Algebra Software",Proceedings of the SC98 Conference, Orlando, FL, IEEE Publications, November, 1998.

[15]B. Wilkinson, M. Allen, Parallel Programming: Techniques and Applications Using Networked Workstations and Parallel Computers, 2nd Edition, Prentice Hall, 2004.

[16] Top500 Supercomputing Sites, http://www.top500 .org