Performance Evaluation of two Parallel Programming Paradigms Applied to the Symplectic Integrator Running on COTS PC Cluster

Lorena B. C. Passos[1], Gerson H. Pfitscher[1], Tarcísio M. Rocha Filho[2]

[1]University of Brasilia
Department of Computer Science
70.919-970, Brasília – DF, Brazil
{lbrasil, gerson}@unb.br

[2]University of Brasilia
Institute of Physics
70.919-970, Brasília – DF, Brazil
marciano@fis.unb.br

Abstract

There are two popular parallel programming paradigms available to high performance computing users such as engineering and physics professionals: message passing and distributed shared memory. It is interesting to have a comparative evaluation of these paradigms to choose the most adequate one. In this work, we present a performance comparison of these two programming paradigms using a Computational Physics problem as a case study. The self-gravitating ring model (Hamiltonian Mean Field model) for N particles is extensively studied in the literature as a simplified model for long range interacting systems in Physics. We parallelized and evaluated the performance of a simulation that uses the symplectic integrator to model an N particle system. From the obtained results it is possible to observe that message passing implementation of the symplectic integrator presents better results than distributed shared memory implementation.

1. Introduction

In Physics and Engineering, there are several simulation problems that require High Performance Computing to be executed. In Astrophysics and Cosmology, the many-body self-gravitating systems are of great importance. Typical examples of those systems are globular clusters and elliptical galaxies recognized as self-gravitating stellar systems.

1-4244-0910-1/07/$20.00 ©2007 IEEE.

In this work, an experimental model representing an *N*-body self-gravitating system is used. In this model, it is simulated the circular movement of particles of mass m, speed v in a radius r having a spherical mass distribution and constant density. A mathematical tool that can be employed in the simulation is the symplectic integrator.

The symplectic integrator is best suited for mechanical conservative systems described by Hamilton equations, since they preserve the volume in phase space and can be put in an explicitly reversible form. Very large simulations are not possible to run using a sequential implementation of a symplectic integrator algorithm, though it is necessary to parallelize the solution.

There are several ways to parallelize a sequential code and various parallel programming paradigms that could be used to implement the parallel algorithm. Message passing and Distributed Shared Memory – DSM – are two well known parallel programming paradigms designed to fit in distributed architectures like personal computer clusters. During the last decades, cluster computing has shown an enormous growth and is an interesting low cost alternative of parallell computational system.

Our goal in this work is to compare the performance of the message passing and the DSM parallel programming models used to implement the symplectic integrator applied to solve an *N*-Body problem. The algorithm chosen is the fourth order integrator described by Omelyan et al. [1]. Our code is applied to a self-gravitating ring model for *N* particles, extensively studied in the literature as a simplified model for long range interacting systems in Physics [2, 3, 4, 5, 6, 7]. The interaction potential in this system leads to some simplification in the numerical implementation such that the CPU time scales linearly with *N*. The message passing implementation chosen to be used is the MPICH 1.2.6 and for DSM we employed a middleware called JIAJIA.

In this work, we present the results obtained from the execution tests of the message passing and the DSM implementations of the symplectic integrator, and from these results we concluded that the message passing implementation has better performance results than the DSM one for this particular application. We also present the application behavior obtained by tracing points placed into the MPICH source code, using a hardware assisted tool called PM^2P.

2. Parallel programming paradigms

There are two most used parallel programming paradigms for parallel systems based on clusters of computers: message passing and distributed shared memory [8], thus it is interesting to compare them and evaluate their performance in a real case.

In the message passing paradigm data is exchanged among remote tasks basically by send/receive operations and any process interaction is made by the explicit message passing. The message passing is typical for programs running on computational systems each with its own private memory and connected by a communication network. Due to possible communication overheads, the use of this programming paradigm is only profitable when the application granularity is bigger than the communication cost involved. Message passing libraries allow efficient parallel programs to be written for distributed memory systems, providing routines to initiate and configure the messaging environment as well as sending and receiving packets of data.

The two most popular high-level message-passing systems for scientific and engineering application are: Message Passing Interface – MPI – defined by the MPI Forum [9] and the Parallel Virtual Machine – PVM – from Oak Ridge National Laboratory [10]. PVM has several features that MPI lacks to offer [11], leading to conclude that this would be the best choice for the implementation of a parallel application. However, these features offered by PVM have a cost that appears in the application performance as observed in [12], where one same application was implemented using MPI and PVM, and performance of implementation MPI showed superior performance results. Therefore, in the present work, MPI was the message passing model chosen for the implementation of the application.

Distributed Shared Memory – DSM – is the extension of the shared memory programming model on systems without physically shared memory, in this model the shared data space is accessed through normal read and writes operations encapsulated by a middleware that still uses message passing. In contrast to the message passing paradigm, in a DSM system a process that wants to fetch some data does not need to know its location, it is the DSM middleware that will find and fetch the data automatically. In most DSM systems, shared data may be replicated to enhance the parallelism and the efficiency of the applications. There are many implementations of DSM middleware, e.g., Quarks [13], JIAJIA [14, 17], TreadMarks [15], and DSM-PM 2 [16]. In this work we use JIAJIA middleware.

3. Code instrumentation

Instrumentation is done by the insertion of specific code instructions to detect and record the program events. Each event record contains at least the type of the event, the date, and the identification of what process performed the event. Performance tuning, debugging, and testing all require instrumentation to provide information about the execution. Information about the use of architectural features, for example, is required for performance tuning, while detailed information concerning the global state of the computation is essential for testing and debugging [18].

Code instrumentation is one way to collect data about a program and there are many ways to do this instrumentation. One can choose to insert it into the application source code directly or to let the compiler automatically do this job [19]. Another way to instrument the code is to use runtime libraries, or even to modify the linked executable.

In this work, we are interested in collecting finer grained information about our application, and due to this fact, we decided to use direct instrumentation of the application because when it is compared to kernel or library instrumentation it is known that this is the best way to collect information in this granularity [18]. For example, loop level or basic block data can only be collected with the direct instrumentation technique.

To instrument our code, we tested five ways to capture time information: MPI_WTIME, jia_clock, Gettimeofday, RDTSCdll, and rdtsc. Our experiments showed that the assembly instruction rdtsc is the less costly and consequently the less intrusive way to gather time results. This assembly instruction returns the number of clock cycles elapsed since the last reboot of the machine and due to its machine's clock precision and minimal execution time, it gives better execution time information when compared to the functions gettimeofday of the C language or to mpiwtime of the MPI library that gives only microsecond precision [20].

4. Hardware assisted tracing

There are three basic techniques for tracing: Hardware, Hybrid and Software tracing [18, 21]. Each of those has

positives and negatives points. Software tracing is the cheapest and most portable technique, but it is difficult to obtain high quality traces due to the lack of global clocks and to the intrusions caused by the tracing activity itself. Hardware tracing requires the development of specific hardware what can be very costly. On the other hand, it is the less intrusive way of tracing, what means that it tends not to interfere into the code execution to make the trace.

Hybrid tracing combines the software and hardware tracing approaches. This kind of hardware assisted tracer is triggered by application level instructions, the program being monitored initiates event recording by making a request to the hardware collector, and its generated information is written on dedicated hardware ports connected to monitoring hardware by a separate bus or interconnection network, what maintains the monitoring intrusion in a very low level. This approach provides the event visibility of an application direct instrumentation with the low overhead of dedicated hardware collectors.

To obtain precise measures of execution time, it is necessary to synchronize the clocks of the machines where the program is executed. However, in Beowulf class of computer clusters, clock synchronization is one of the problems that constantly appear. The individual clock of each node is sensitive to external conditions such as temperature and suffers the effect of different constant drifts, making it difficult to maintain [22] a unified clock view. Many software, hardware and hybrid [18] approach have been proposed to enhance clock synchronization between the nodes of the clusters, but problems like network delays do not permit a perfect view of an unified reference clock [22, 23]. Due to this fact, that local clocks of the machines are asynchronous, it is necessary to have an external way to obtain the time measures of application events.

One very interesting approach to this choice is the use of the parallel port to send signals to a machine which is responsible for collecting the data of all the machines executing the program. In [24], it is presented PM^2P, a tool for use in clusters of personal computers that provides a graphic visualization of the temporal execution of distributed applications that use the MPI standard for message passing. The tool uses an approach involving the parallel port to read the time of events that occur in all different machines of a cluster. In the present work, we used PM^2P to observe the behavior of the MPI implementation in a cluster environment.

5. Symplectic parallel algorithm

In astrophysics and cosmology, many-body gravitating systems are of great importance. Typical examples of these kinds of systems are globular clusters and elliptical galaxies, which are recognized as self-gravitating stellar systems. In this work, we used a toy model representing an *N*-body self-gravitating systems: the circular motion of particles of mass m in a spherical mass distribution with a constant density. The core values involved in this model are: position, velocity, mass and force. In our tests, the mass of all the particles of the system was considered to be unitary. In the algorithm implementation, the values of position, velocity and force are represented by three distinct vectors: $\boldsymbol{p}$, $\boldsymbol{r}$ and $\boldsymbol{f}$ respectively. Each element of these vectors corresponds to the information about each system particle.

The mathematical tool used to solve the equations modeling the simulation is the symplectic integrator, which is best suited for mechanical conservative systems described by Hamilton equations, since they preserve the volume in phase space and can be put in an explicitly reversible form. The sequential algorithm chosen is the fourth order integrator described by Omelyan et al. [1]. This algorithm works as follows: in the initialization step of the algorithm, random values are given to each element of the vectors $\boldsymbol{r}$ and $\boldsymbol{p}$, and the initial energy of the system is calculated. After the initializing step, there is an iteration loop that represents system time elapsing. Inside this loop, four different steps are followed to calculate the values of force, velocity, and position for each particle of the system. The computation continues until the time previously set by the loop condition finishes. Figure 1 shows the expressions for the single-step propagation of position and velocity from time t to $t+h$ of the symplectic sequential algorithm.

$$\boldsymbol{r}_{\mathrm{I}} = \boldsymbol{r}(t) + \xi h \boldsymbol{f}[\boldsymbol{p}(t)]/m$$
$$\boldsymbol{p}_{\mathrm{I}} = \boldsymbol{p}(t) + (1-2\lambda) h \boldsymbol{r}_{\mathrm{I}}/2$$
$$\boldsymbol{r}_{\mathrm{II}} = \boldsymbol{r}_{\mathrm{I}} + \chi h \boldsymbol{f}[\boldsymbol{p}_{\mathrm{I}}]/m$$
$$\boldsymbol{p}_{\mathrm{II}} = \boldsymbol{p}_{\mathrm{I}} + \lambda h \boldsymbol{r}_{\mathrm{II}}$$
$$\boldsymbol{r}_{\mathrm{III}} = \boldsymbol{r}_{\mathrm{II}} + (1-2(\chi+\xi)) h \boldsymbol{f}[\boldsymbol{p}_{\mathrm{II}}]/m$$
$$\boldsymbol{p}_{\mathrm{III}} = \boldsymbol{p}_{\mathrm{II}} + \lambda h \boldsymbol{r}_{\mathrm{III}}$$
$$\boldsymbol{r}_{\mathrm{IV}} = \boldsymbol{r}_{\mathrm{III}} + \chi h \boldsymbol{f}[\boldsymbol{p}_{\mathrm{III}}]/m$$
$$\boldsymbol{p}(t+h) = \boldsymbol{p}_{\mathrm{III}} + (1-2\lambda) h \boldsymbol{r}_{\mathrm{IV}}/2$$
$$\boldsymbol{r}(t+h) = \boldsymbol{r}_{\mathrm{IV}} + \xi h \boldsymbol{f}[\boldsymbol{p}(t+h)]/m$$

Figure 1. Single-step propagation of position and velocity from time *t* to *t+h* for a single particle

For the algorithm parallelization, we adopted a simple approach which consists in the equally distribution of the system particles among the computational tasks, therefore, each task is responsible for a subset of particles. In the message passing implementation, reduction operations are responsible for the aggregation of values calculated for each particle subset by the responsible task and for the posterior broadcast of the total values to all tasks. In the

shared memory implementation, global values are directly written into shared variables using the appropriate synchronization mechanisms. Figure 2 shows the pseudo-code of the parallel symplectic algorithm implementation for each subset of system particles, it is also shown the communication that occurs among the existing tasks and it is possible to observe the tracing points introduced into the code. Our code is applied to a self- gravitating ring model (Hamiltonian Mean Field model) for N particles, extensively studied in the literature as a simplified model for long range interacting systems in physics. The interaction potential in this system leads to some simplification in the numerical implementation such that the CPU time scales linearly with N. This property is particularly useful for testing the parallel code for a large number of particles (typically 10^8). As mentioned before, a reason for using parallel computing is putting together several processors in parallel to solve a problem should help obtain the solution more quickly. Moreover, distributing a problem onto several processors can help solve a problem that is very large for a single serial machine. In our experiment, we observed that the sequential code maximum number of particles is limited to 50 millions, when we increased this number of particles, the system answers with a lack of memory message. Though, the parallel execution seems to be the solution for a larger number of particles simulations.

```
while (time < finalTime)
{
     // 1° step

     for (i=0;i<vectorSize;i++)
     {
            tracing_point1
            Force_calculation
            Position_calculaton
            Velocity_calculation
            tracing_point2
     }
     tracing_point3
     Communication among tasks
     tracing_point4
     .....
     // 4° step

     for (i=0;i<vectorSize;i++)
     {
            Force_calculation
            Position_calculaton
            Velocity_calculation
     }

     Communication among tasks
     time = time + deltaTime;
}
```

Figure 2. Parallel symplectic algorithm showing communication among tasks and tracing points.

6. Computational environment

The cost of COTS (commodity off the shelf) components for PCs has been constantly decreased over the past decades, what made it possible to build parallel systems, such as PCs clusters, with a relatively little amount of money. Moreover, the growth of public domain software such as MPI library and operational systems like Linux, also contributed a lot to the increment of this kind of clusters. Nowadays, the cluster is an established and spreadly used paradigm of parallel computation [18, 19].

Our computational environment is a homogeneous Beowulf cluster of eight machines. Each machine has two AMD Atholn MP 1900+ processors at 1600 MHz and 256 KB of cache L2 memory. Each node has 1GB of local RAM memory and 40 GB hard disk. They are interconnected by a gigabit network. We run it with a version of Linux operating system.

As shown in Figure 3, when using the parallel port to gather the performance results, we used a front-end machine that has the following characteristics: Intel Pentium II 350 MHz processor with 512 KB of L2 cache, 160 MB of RAM memory and a hard disk of 6.5 GB. This machine is interconnected to the others by a megabit network.

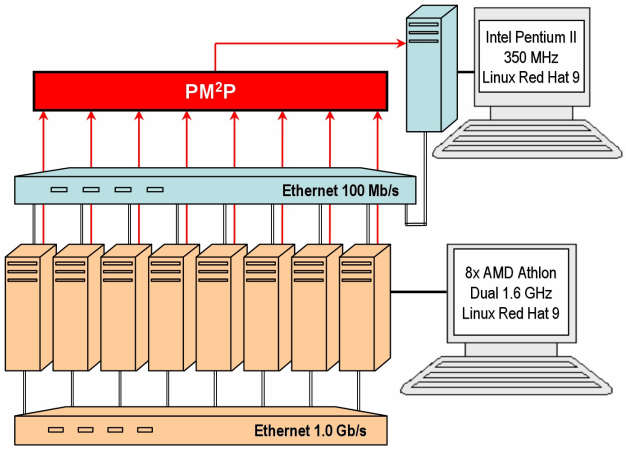

Figure 3. Homogeneous computational environment and PM²P tool.

7. Experimental results

The implementations written in message passing and distributed shared memory were tested for 1, 2, 4, 6 and 8 tasks, each task running on a different machine of the homogeneous cluster [20]. We ran the tests in a way that all the tasks treated the same number of particles. The total number of particles used in the tests was in the range of 24 thousand to 98 millions of particles. Figure 4

graphically exposes the behavior of the execution times, in seconds, for a single iteration of the message passing implementation.

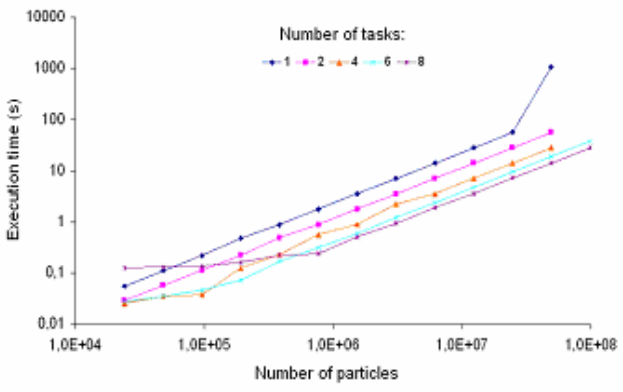

Figure 4. Execution time versus number of particles and number of tasks for the MPI implementation.

Using the results showed before, we calculated the obtained speedup. Due to the fact that the simulation with just a task is limited to 50 million particles, it was not possible to calculate the speedup for a simulation of a system with 98 million of particles. The graphics of Figure 5 visually presents the obtained speedup. Looking at this graphics, we can notice that significant values for the speedup are obtained for a number of tasks greater than 4, simulating a system with more than 49 million particles.

One possible explanation for this fact is that using a little number of tasks (one task per machine) leads to an overload of each of them when processing the vectors involved in the calculus of the simulation. This overload decreases when new tasks (machines) are added to the computational system.

For the JIAJIA implementation the tests were also executed using 1, 2, 4, 6 and 8 tasks, one task per machine, but the number of particles of the simulated system was restricted to a range of 24 thousand to 3 million of particles. During the execution of the tests, we noticed that for 2 tasks (2 machines) it was possible to simulate a system of at most 12 million of particles, but it was not possible to run the tests for this number of particles to another number of tasks. Thus, we decided to analyze the data obtained from the simulations of at most 3 million of particles.

The graphic of Figure 6 shows the behavior of these values for each set of machines, allowing the visualization of the relation between number of machines, number of particles and execution times. It can be noticed that the behavior of the relation between number of machines and execution times is the inverse one of what it happens in the MPI implementation. In the JIAJIA implementation, as the number of machines grows, the time of execution for one same number of particles also grows. However, in the MPI implementation, as more machines are added to the cluster, the execution time decreases. This difference can be explained for the fact that as the number of machines in the cluster increases, JIAJIA has a bigger cost in the distributed shared memory management what leads to a significant performance loss. On Figure 7 are represented the obtained speedups for the JIAJIA implementation.

On the tests previously taken, we also measured the communication times for each implementation which obtained results are next presented on the following graphics. On Figure 8, one can observe how the communication cost behaves as function of the number of particles and the number of tasks for the MPI implementation for an iteration of the algorithm.

The values of communication costs for the JIAJIA implementation are presented on Figure 9 which shows the behavior of this cost as a function of the number of particles and the number of tasks.

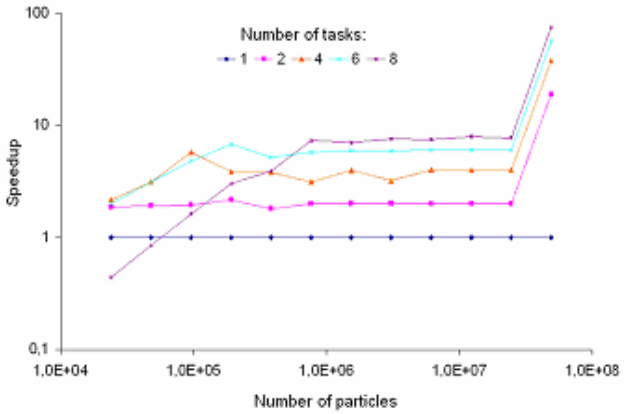

Figure 5. Obtained speedups versus number of particles and number of tasks for the MPI implementation.

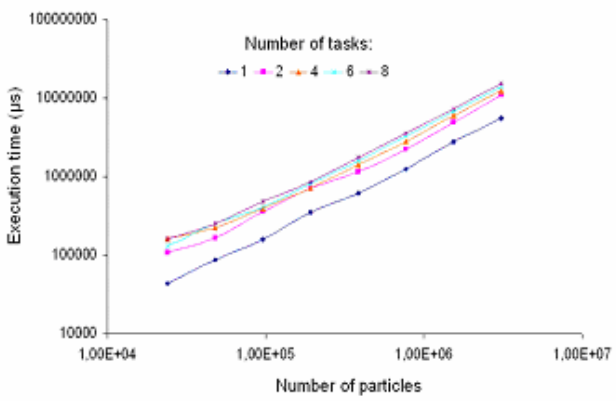

Figure 6. Execution time versus number of particles and number of tasks for the JIAJIA implementation.

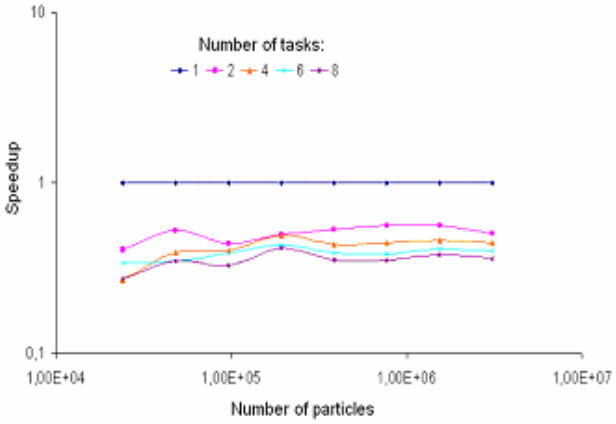

Figure 7. Obtained speedups versus the number of particles and the number of tasks for the JIAJIA implementation.

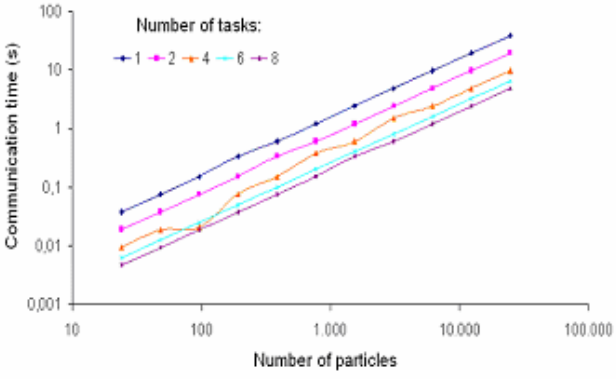

Figure 8. Communication costs versus the number of particles and the number of tasks for the MPI implementation.

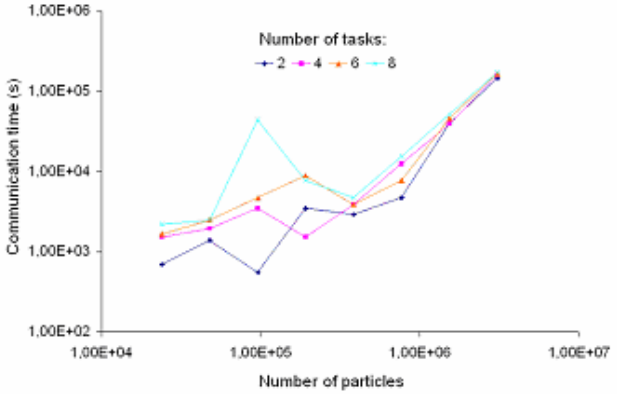

Figure 9. Communication costs versus the number of particles and the number of tasks for the JIAJIA implementation.

The three Gantt charts (Figures 10, 11, and 12) are graphical representations of the durations for the execution of an algorithm iteration of, respectively, 2, 4, and 8 tasks against the progression time. These results were obtained using the PM^2P tool. In these figures, each vertical bar represents a specific tracing point placed into the source code.

The Gantt chart of Figure 12 represents the execution of an iteration of a 98 million particles system distributed among eight tasks. The values for the tracing points recorded time instants are detailed on Table 1. The time interval between instants Time0 and Time1 (tracing points) is spent in parallel tasks initialization. Data structure allocation and initialization occurs on the time interval between time instants Time1 and Time2. These two time intervals just happen on the first program iteration, the following intervals occurs inside the iteration loop. Inside the loop, mathematical computations are done between time instants Time3 and Time4. Time interval between Time4 and Time5 denotes the interval time spent in the global communication carried out by the AllReduce MPI operation. Finally, time instant Time6 represents the iteration execution end.

8. Conclusions

In this work it is presented the performance evaluation of two implementations of the symplectic integrator using two parallel programming paradigms. These paradigms are not restricted to professionals of Computer Science; researchers of areas like Physics, Mathematics, Engineering, and Biology also need High Performance Computing to make their simulations feasible.

To quantify the parallel execution performance of the two distinct implementations, the measures of execution times and the calculation of obtained speedups have been

done varying the number of tasks and the size of the problem, by means of the variation of the number of particles of the simulated system. To measure the execution times, the assembly rdtsc instruction was inserted into specific places of each source code. As the number of particles was increased, we observed a limitation in the JIAJIA implementation in respect to the number of particles that could be simulated at a time. In a general way, for any number of tasks, it was only possible to simulate a system of up to three million of particles. However, for the message passing implementation, systems of up to a hundred million of particles could be simulated.

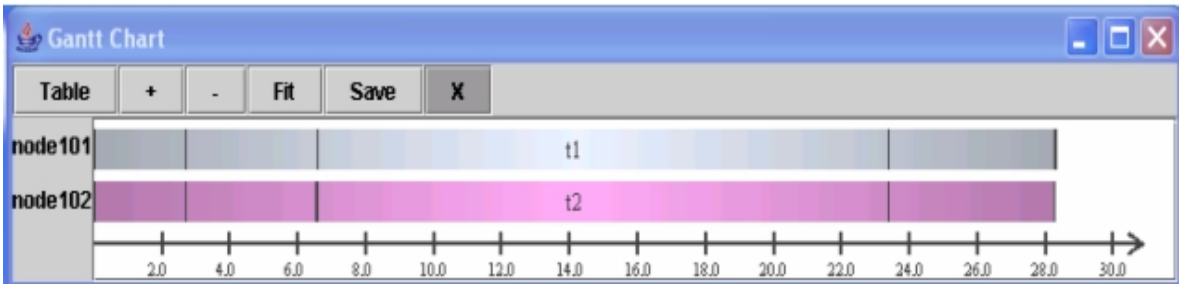

Figure 10. Gantt chart for 2 tasks working on a system of 24 million of particles.

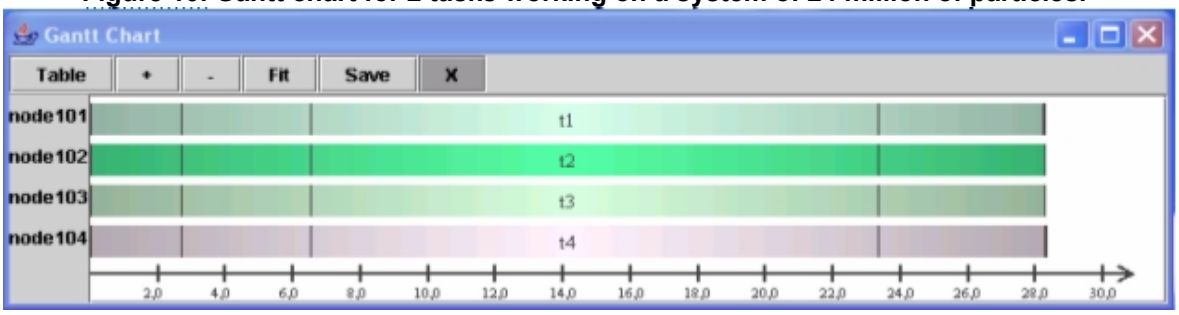

Figure 11 – Gantt chart for 4 tasks working on a system of 49 million of particles.

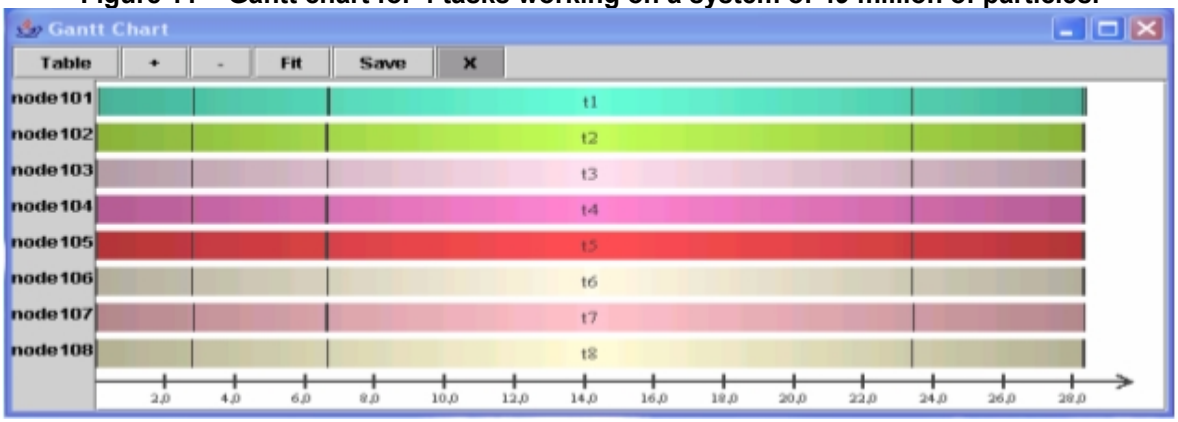

Figure 12 – Gantt chart for 8 tasks working on a system of 98 million of particles.

Table 1. Time instants of program events shown in Figure 12.

Task	Time0	Time1	Time2	Time3	Time4	Time5	Time6
1	0,05056	2,82642	6,65730	6,66988	23,40901	28,27691	28,35633
2	0,00000	2,77684	6,60735	6,62154	23,37406	28,26940	28,31549
3	0,00144	2,78168	6,63051	6,63118	23,40428	28,30632	28,35172
4	0,00152	2,78062	6,61694	6,62858	23,40176	28,30703	28,34064
5	0,00179	2,78098	6,61007	6,62900	23,39526	28,29311	28,34224
6	0,00212	2,78170	6,62215	6,63013	23,40304	28,30538	28,34612
7	0,00253	2,78297	6,61695	6,63267	23,44196	28,35466	28,35542
8	0,00285	2,78275	6,62684	6,63183	23,40427	28,30672	28,35122

From the obtained results it was possible to observe that the message passing implementation of the symplectic integrator presents better results in terms of total execution time than distributed shared memory implementation. We believe that these results are caused by the costs involved on distributed shared memory management, which involves kernel system calls, context switching, and communication latency. In our point of view, another positive aspect of the MPI implementation is that it forces the programmer to explicitly mark into the code the communication points, what permits an enhanced control and understanding of what is happening during runtime. Moreover, MPI has a better support community and documentation.

Using a tool like PM^{2}P makes it possible to visualize the tracing points and identify execution bottlenecks that lead to performance loss and try to solve them. In the present work, we used the PM^{2}P tool to monitor the execution in a homogeneous environment, bringing us a better understanding of the application behavior.

References

[1] Omelyan I. P., Mryglod I. M., Folk R., Optimized Forest-Ruth- and Suzuki-likealgorithms for integration of motion in many-body systems, Computer Phys. Comm., V.146, No 2.- P. 188-202, 2002.

[2] Tatekawa T., et al, Thermodynamics of the self-gravitating ring model, Physical Review E 71, 056111, The American Physical Society, 2005.

[3] Borstnik U, Janezic D., Symplectic Molecular Dynamics Simulations on Specially Designed Parallel Computers, J. Chem. Inf. Model, Nov-Dec;45(6):1600-4, 2005.

[4] Oh, K. J., Klein, M. L., A general purpose parallel molecular dynamics simulation program, Computer Physics Communications 174, pp. 560-568, 2005.

[5] Spinnato, P. F., van Albada, G.D., Sloot, P.M.A, Performance Modeling of Distributed Hybrid Architectures, IEEE Transactions on Parallel and Distributed Systems, vol. 15, no. 1, pp. 81-92, 2004.

[6] Pruett, C. D., Rudmin, J. W., Lacy, J. M., An adaptive N-body algorithm of optimal order, Journal of Computational Physics, vol. 187, Issue 1 , pp. 298-317, 2003.

[7] Sweatman, W. L., The development of a parallel N-body code for the Edinburgh concurrent supercomputer, Journal of Computational Physics, vol. 111, no. 1, pp. 110-119, 1994.

[8] Hungershöfer, J., Streit, A., Wierum, J-M, Efficient Resource Management for Malleable Applications, Technical Report PC 2 TR-003-01, Paderborn Center for Parallel Computing, Paderborn, Germany, December 2001.

[9] MPI Fórum www.mpi-forum.org/

[10] Parallel Virtual Machine http://www.csm.ornl.gov/pvm/

[11] Gropp, W., Lusk, E, Goals guiding design: PVM and MPI, IEEE International Conference on Cluster Computing Proceedings, pp. 257-265, 2002.

[12] Villa Verde F., Pfitscher, G. H., Viana D. M., Performance Characterization of a Parallel Code Based on Domain Decomposition on PCs Cluster, I2TS'2006 - 5th International Information and Telecommunication Technologies Symposium, Cuiabá, Brazil, 2006.

[13] Quarks http://www.cs.utah.edu/flux/quarks.html

[14] JIAJIA, JIAJIA Distributed Shared Memory Project, Institute of Computing Technology, CAS; http://www.ict.ac.cn/chpc/dsm/.

[15] TreadMarks http://www.cs.rice.edu/~willy/TreadMarks/

[16] DSM-PM2 http://www.irisa.fr/paris/pages-perso/Gabriel-Antoniu/dsm-pm2.htm

[17] Weiwu H., Weisong S., Zhimin T., The JIAJIA Software DSM System; Center of High Performance Computing, Institute of Computing Technology, Chinese Academy of Sciences, 1998.

[18] Nonaka, J., Pfitscher, G. H., Nakano, H., Onisi, K. Low-Cost Hybrid Internal Clock Synchronization Mechanism for COTS PC Cluster. EuroPar 2002, LNCS 2400, Springer-Verlag, Berlin, pp 121–124, 2002.

[19] Parhami, B., Introduction to Parallel Processing Algorithms and Architectures, Kluwer Academic Publishers, University of California at Santa Barbara, California, 2002.

[20] Ferreira, R. R., Performance characterization of the finite element method parallel aplication in heterogeneous environments, Computer Science MSc Thesis, University of Brasília, 2006.

[21] Kergommeaux, J. C., Maillet, E., Vincent, J-M., Monitoring parallel programs for performance tuning in cluster environments. In: Parallel program development for cluster computing: methodology, tools and integrated environments archive, pp 131–150, Nova Science Publishers, Inc. Commack, NY, USA, 2001.

[22] Tanenbaum, A. S., van Steen, M., Distributed Systems: Principles and Paradigms, Prentice Hall, 2002.

[23] Verissimo, P., Raynal, M., Clock and Temporal Order, in: Recent Advances in Distributed Systems, Springer-Verlag, 2000.

[24] Haridasan, M., Pfitscher, G. H., PM^{2}P: A tool for performance monitoring of message passing applications in COTS PC clusters, in Proceedings of the 15th Symposium on Computer Architecture and High Performance Computing (SBAC-PAD'03), 2003.

[25] Top 500 Supercomputer Sites, http://www.top500.org

[26] Gobbert, M. K, Configuration and Performance of a Beowulf cluster for Large Scale Scientific Simulations, Computing in Science and Engineering, pp. 14-26, 2005.

[27] Passos, L.B.C., Performance evaluation of a method to solve the temporal evolution of self-gravitating systems using two parallel programming paradigms: message passing and distributed shared memory, Computer Science MSc Thesis, University of Brasília, 2006.