A PyMPI-based Approach for Parallel Interactive MD Simulations

Eduardo R. Rodrigues, Airam J. Preto, Stephan Stephany

Brazilian Institute for Space Research (INPE)
CP 515, 12245-970 S. J. Campos SP, BRAZIL
E-mail: {erocha, airam, stephan}@lac.inpe.br

Abstract

A PyMPI-based approach for parallel interactive molecular dynamics (MD) simulations is proposed, derived for the ADKS sequential MD software. The ADKS software allows the simulation and interactive visualization of complex phenomena as fractures and grain boundary behaviour in solids. The simulation engine was parallelized using the MPI (Message Passing Interface) communication library. A finite automata model represents the interaction between the user interface and the simulation engine. The PyMPI extends the Python language for the MPI environment and it is used to integrate the user interface and the parallel simulation engine. Performance results are shown without/with visualization for execution in a distributed memory parallel machine.

Palavras-chave: *High Performance Cluster Computing, Molecular Dynamics, Python, MPI, Interactive Simulations*

1 Introduction

Molecular Dynamics (MD) is a technique used to simulate particles such as atoms and molecules in order to compute equilibrium and non-equilibrium properties of matter. That simulation consists of a set of particles distributed in a space region interacting with each other through a certain potential characteristic of the system being simulated. Resulting forces are calculated for each particle in order to estimate its motion according to classical mechanics. Macroscopic system properties can thus be obtained. MD has been used to study microscopic system ignoring quantical effects. However, semi-classical corrections can be employed in order to to include those effects. Many areas employ MD, as nanotecnology, biochemistry, molecular biology and material science.

MD implementations are CPU-bound and the high number of particles and timesteps present a heavy processing load. Typically, simulations involve many thousands or millions of particles modelling only picoseconds of sub-micron scale phenomena. When short range potentials are used in MD simulations, algorithms such as cell-subdivision and neighbor-list can be employed to improve the efficiency of the computations. Even using these algorithms, large systems require the use of parallel machines to produce results in a feasible time. The most common parallel programming paradigms are the distribution of atoms, cells or force calculations among processors.

These simulations can be executed in batch mode, as a sequence of steps such as: definition of input data, execution of the simulation engine, output of data to a storage device and post-processing analysis. The last simulation results can then be used with redefined input data to begin a new cycle. This approach has some disadvantages: the first one is that the amount of data stored may increase very rapidly and the second one is that the results are only avaliable after the end. Another approach, used in simulations, is to have visual interactive capabilities. Therefore the parameters of the simulation can be changed on the fly, allowing the visual feedback at run-time. Complex phenomena that could not be observed in a real experiment or in the *post-mortem* analysis of data obtained from a batch simulation, can thus be studied in interactive computational environment [4]. The need of parallel computing is emphasized, in order to have interactive simulations with execution times that are confortable for the user.

In parallel simulations with visual interactive capability it is difficult to control the execution of the simulation engine, that must be synchronized with the particle visualization. Some solutions have been proposed, for example, the Falcon system [3] that is based on a set of tools and libraries, and an user interface that provide means to gather processed data, analyse data and change simulation parameters in execution time. Another system is based on the SPaSM (*Scaleable, Parallel, Short-Range Molecular Dynamics*) code [2]. SPaSM creates an interactive environment employing the Python extensible scripting language that supports the inclusion of visualization and user-developed mod-

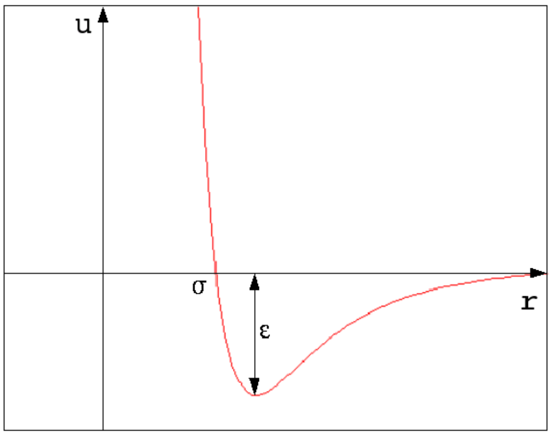

Figure 1. Lennard-Jones Potencial

ules.

This work presents a PyMPI implementation of a parallel MD simulator with interactive visual capability based on ADKS. The simulation engine, that was written as a library, was parallelized using the MPI communication library. The proposed scheme for the integration of the parallel simulation engine and the visualization module to the interactive environment PyMPI are also shown.

2 Molecular Dynamics

MD simulations can model the particles in different ways. The simplest model represents the particles as spheres that interact with each other according to a potential that depends only on the distance between each pair of particles. The potential must be selected according to the system being simulated. In the case of electrically neutral molecules, the interaction is characterized by a strong short-distance repulsion due to the Pauli exclusion principle and by a weak attraction that starts from a certain distance due to the van der Waals forces. Therefore, a possible potential for noble gases or generic substances is the Lennard-Jones short-range potential [6], defined in Equation 1, in function of the distance r between particles and two specific parameters: σ and ϵ. This potential is shown in figure 1.

$$u(r) = 4\epsilon\left[\left(\frac{\sigma}{r}\right)^{12} - \left(\frac{\sigma}{r}\right)^{6}\right] \qquad (1)$$

Considering this equation, it can be shown that the Lennard-Jones potential is negligible for $r > 3\sigma$. Thus, a cut-off distance r_c can be defined in order to have $u(r) = 0$ for $r > r_c$. The interaction force **f** derives from the potential according to $\mathbf{f} = -\nabla u(r)$. Therefore, the number of calculations can be drastically reduced since r_c is very small in comparison with the dimension of the system of particles. For each particle, the interaction forces with all surrounding particles has to be computed in order to obtain the resultant force. Newton's 2nd law is then employed to calculate the acceleration and sucessive numerical integrations updates velocity and position using the finite-difference method. Two simple schemes can be employed for these updating, the Verlet or the Leap-frog with good accuracy in comparison to more sophisticated schemes [6].

The relationship between the microscopic information that results from MD simulations and the macroscopic properties of the system is made by means of Statistical Mechanics. The termodynamical state can be described by a set of parameters as termperature, pressure and number of particles. At microscopic level a particles system can be descripted by its positions (coordinates x, y and z) and momenta (directions x, y and z), thus N particles can be represented a space with 6N dimensions that is called phase space. A set of points in this phase space that satisfy the conditions of a particular thermodynamic state is called a ensamble. Molecular Dynamics generates points in the phase space that belong to one ensemble, thus properties of the ensemble can be obtained.

The limit of computations imposed by the hardware restricts the number of particles that can be simulated and consequently the complexity that can be handled [6]. For short range potential the time spend to test the particles distance may consume up to 99%, thus some algorithms like cell subdivision and neighbors list are used to reduce that number of tests [6]. Nevertheless the independency among particles that are far from each other and the necessity to simulate large systems point to the need for parallel processing. The most common forms to parallelize MD are atom decomposition, force decomposition and space decomposition [5].

The ADKS is an interactive MD software with visulization capability that can simulate fractures and grain boundary behaviour in solids. It was developed for a event-driven environment provided by X11 Window System and the Motif library. In ADKS, the simulation parameters can be changed on the fly, thus the effects of these modifications can be visualized during the simulation.

3 Interactive MD simulation in a parallel environment

In an interactive simulation, executed on a sequential machine, a simple event-driven programming model allows simulation steering, i.e. the modification of simulation parameters without restarting it from the beginning. A parallel version of this model for shared

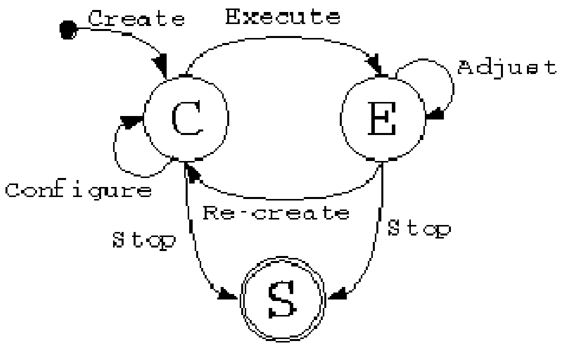

Figure 2. Finite automata model.

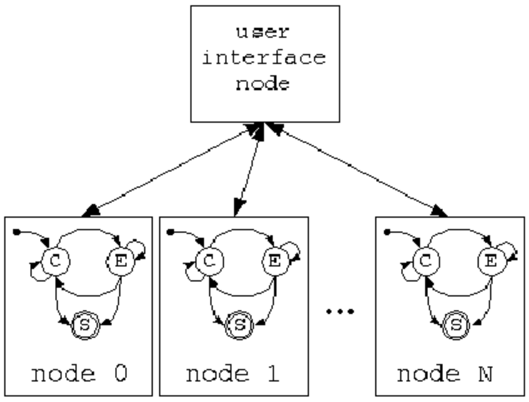

Figure 3. Interaction between user interface and a set of finite automata.

memory machines would simply require to decouple the visualization interface from the simulation engine. Similarly to the sequential model, events can modify parameters of the simulation that is executed by multiple threads, provided that race conditions are avoided. However, in a distributed memory parallel machine, this strategy can not be used. When a message passing program is run, multiple independent processes are started in different nodes.

A possible scheme for a distributed memory architecture is to assign the user interface and the coupled visualization to a specific processor, and to execute the simulation engine in the remaining processors. The visualization processor also collects the results of the simulation from the other processors at every timestep. User interface allows the modification of simulation parameters. In this case, messages are sent to the other processors. The simulation engine can thus be modeled as a finite automata, being its state transitions triggered by user commands.

A simplified model is depicted in Fig. 2. Initial state C represents the starting of a new simulation that can be changed to the execution state E. The E to C transition is used to interrupt the current simulation and to start a new one. A transition to the state S causes the simulation to end. The *adjust* transition allows to change parameters during execution, for example temperature and pressure, while the *configure* transition deals with "static" parameters, that can be configured only before the simulation starts, as the number of particles or spatial domain. In this way, the complete system models the interaction between an user interface executed in a terminal-provided node and a set of finite automata being executed in the processing nodes (see Fig. 3).

4 The Simulation engine

The ADKS sequential code employs the cell subdivision algorithm. The most intuitive way to parallelize is to use the domain decomposition approach. The 2D domain is divided in constant-width bands and each band is assigned to a processor. This distribution minimizes the number of processors that a processor communicates with.

A key issue is how to deal with the interaction between particles in the interband border. As the width of the cells is the cut-off distance, the borders are composed of two columns of cells, each of them in a different processor. Therefore, in order to calculate interaction forces and update particle positions in its part of the border, a processor must have particle positions of its neighboring part of the border. One possible approach is to communicate border particle positions between neighboring processors. However, this does not take advantage of Newton's 3rd law and force interactions between border particles of neighboring processors are calculated twice. A better approach is to have a sole processor calculating interaction forces for each border. For each border, the right side processor send its left border particle positions to the left side processor that calculates interaction forces and send results back. Considering a generic processor P_n with a left neighbor $P_(n-1)$ and a right neighbor $P_(n+1)$, it sends its left border to $P_(n-1)$ and concurrently receives the left border from processor $P_(n+1)$. Then, it calculates interaction forces for its right border and then send the results to $P_(n+1)$ while receives interaction forces of its left border from $P_(n-1)$.

Assynchronous comunication was used to improve performance in the comunication of the borders. The main ideia is to initiate the sending and receiving assynchrounously and compute the data that do not depend on node local data. After the communication has been accomplished, the dependent data is computed.

A further improvment of this scheme is as follows: processor P_n initiates force calculation and sends assynchronously its left border particle positions to processor $P_(n-1)$ and receives assynchronously the right border particle positions from processor $P_(n+1)$. Concurrently with the communication, P_n calculates interactions of its local particles away from the borders. As soon as the particle positions from the processor $P_(n+1)$ are received, P_n calculates the interaction forces in its right border cells using the received data. Then, these calculated forces are send assynchronously to $P_(n+1)$ while the results from $P_(n-1)$ are being received. Again, during the communication of forces P_n calculates the remaining interactions between its local particles that are far from the border. When communication ends at that iteration, P_n updates all local particle positions. This is followed by the updating of the cells according to the particles that have moved to/from each cell. Eventually, communication is required when migration of particles occurs between cells in different processors.

Performance results for the simulation engine without any visualization are shown in Table 1 and Figure 4 (p denotes the number of processors and N, the number of particles. Execution times refer to 1000 timesteps of a simulation of a micro-canonical ensemble. The parallel machine is an Itautec cluster based on Intel Pentium III Xeon 1.26 GHz dual processors and a Gigabit Ethernet standard network.

5 The PyMPI environment

Python is an interpreted, object-oriented programming language, that has gained popularity because of its clear syntax and readability. Python is said to be relatively easy to learn and portable to a number of operating systems. Its source code is freely available and open for modification and reuse. Python offers

Table 1. Parallel performance of the ADKS simulation engine (Intel PIII Xeon cluster).

	speed-up			efficiency		
N	2p	4p	8p	2p	4p	8p
32857	1.76	3.62	7.23	0.88	0.90	0.90
58855	1.71	3.39	6.53	0.85	0.85	0.82
91487	1.70	3.30	6.37	0.85	0.83	0.80
130899	1.70	3.25	6.38	0.85	0.81	0.80

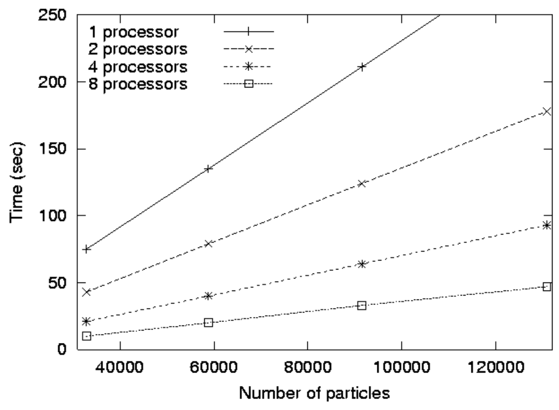

Figure 4. Execution times of the ADKS without pyMPI (Intel PIII Xeon cluster).

dynamic data type, ready-made class, and interfaces to many system calls and libraries.

Two important Python features are (i) a command driven user interface, that allows interactive execution of commands, and (ii) the dynamic extensibility of the language with codes written in C and C++, like functions and objects, that can be incorporated as Python commands. Those features are useful to prototype development since it provides a simple interactive environment and a way to dynamicaly integrate modules, as long as the modules have a well defined interface. PyMPI extends Python to be executed in a distributed memory architecture using the MPI library. In PyMPI, each node of the cluster executes an image of Python, but the user interface executes in a master node. Any command input at that node is sent to all the nodes by means of MPI calls. The PyMPI also initiates the MPI environment, and a SPMD approach is used to assign the tasks to the processors according to their ranks. Modules that are loaded in PyMPI can call MPI routines provided the MPI environment is initiated.

The interfaces that allow Python to be extended with C and C++ codes can be made by hand, as described in [7], or generated automatically [1]. In both cases, it is important to determine which objects and functions will be available for the Python environment as commands. In order to integrate the parallel simulation engine and the PyMPI environment interfaces are created for the main functions of the simulation engine, like the functions to run one timestep, configure initial particle positions and set initial conditions. Besides, specific functions and their respective interfaces were created to set or get simulation parameters. These interfaces compose a set of low level routines used to implement a Python object that represents a simulation. Each processor will have an instance of this object that communicate with other processors using the MPI environment. Thus when a particular method of an instance of simulation is called in the PyMPI command driven user interface, the corresponding low level routines are executed in each node. The advantage of this approach is the simple implementation of the finite automata, since the states of the simulation object and the messages to make the transitions of the automata are provided by the PyMPI environment. The disadvantage is the command-driven user interface, that must be integrated to a simulation engine that is event-driven.

Performance results of the simulation engine without visualization, but integrated to the PyMPI environment, are shown in Table 2 and Figure 7 for the same test cases of the previous section. Here, p denotes the number of processors executing the simulation engine, i.e. there is another processor in charge of the PyMPI user interface. A slight improvment of performance can be noted for $p = 2$ with PyMPI, in comparison to the same test case without PyMPI. This is probably due to the use of processors that are not from the same node in the latter case.

6 The visualization module

The integration of the visualization module to the parallel simulation software was implemented by a configurable processor scheduler in order to select specific set of processors for the modules. During the initialization of the simulation, each module determines how many processors are required and the scheduler assigns a specific MPI communicator to that module. However, in the current work, only the simulation engine module was executed in parallel, since the visualization was not processing demanding and could be executed sequentially in the master processor.

The visualization was written based on the Motif library, but other libraries could be employed and integrated without affecting the simulation engine. Considering p processors, the visualization module is executed by the master processor, and the remaining $(p - 1)$ processors are used by the simulation engine. The visualization module is composed of two parts: an interface to the simulation engine and the graphic visualizer itself. This interface sends configuration parameters to the simulation engine, in order to run the simulation for a specified number of timesteps, and also collects particle data from it. Such data is then displayed by the visualizer.

Table 2. **Parallel performance of the ADKS simulation engine inside pyMPI (Intel PIII Xeon cluster).**

	speed-up			efficiency		
N	2p	4p	8p	2p	4p	8p
32857	1.82	3.58	7.21	0.91	0.90	0.90
58855	1.79	3.28	6.53	0.89	0.82	0.82
91487	1.77	3.31	6.39	0.89	0.83	0.80
130899	1.78	3.26	6.38	0.89	0.81	0.80

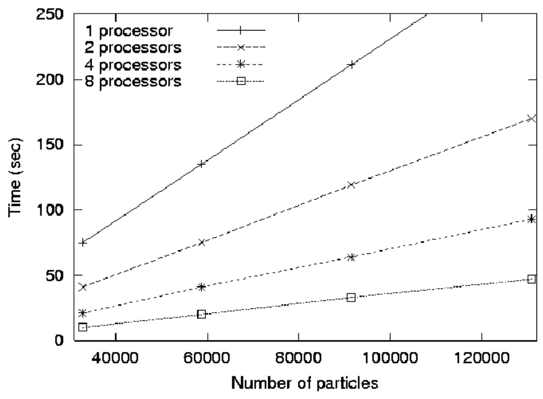

Figure 5. Execution times of the ADKS inside pyMPI (Intel PIII Xeon cluster).

The original ADKS software employed global storage as a means to exchange data between modules. In this implementation, an interface that employs accessor methods similar to those of object-oriented approaches, in order to obtain a better decoupling between the visualization modules and the simulation engine. Consequently, a object-oriented like message interchange can be performed, even with a code that was written in C.

The standard PyMPI interface returns only after a user command is executed by all processing nodes, and therefore another command cannot be input. Particularly, a user command may request a graphic window output, but the user would be unable to input another command while this window is open. In order to solve this issue, a thread based approach is required at the processor in charge of the visualization module and the user interface. A thread is assigned to receive user inputs and simulation data from the other processors, while another thread controls the Motif graphic window. This scheme is shown in Figure 6, with the threads being depicted by the outer retangles.

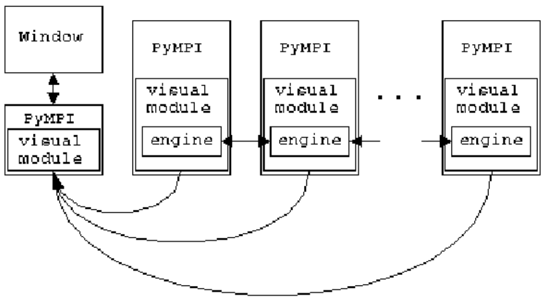

Figure 6. PyMPI

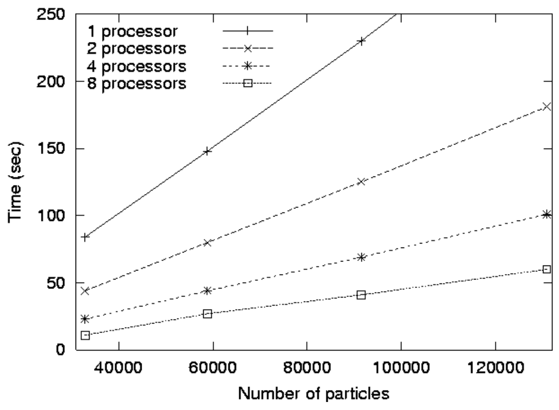

Figure 7. Execution times of the ADKS inside pyMPI with visualization (Intel PIII Xeon cluster).

7 Conclusions

A PyMPI-based approach for parallel interactive molecular dynamics (MD) simulations is presented. This implementation is derived for the ADKS sequential MD software. A PyMPI environment was used to integrate the visualization modules and the parallel MPI-based simulation engine. A multithread approach was required for the processor in charge of the visualization and user interface. Performance results show the suitability of the presented implementation for a distributed memory parallel architecture. Further works include to rewrite the code in order to enhance the object-oriented approach, and to include some visualization improvments, such as image magnification.

References

[1] Beazley, D. M. "Swig : An easy to use tool for integrating scripting languages with c and c++". In *Fourth Annual USENIX Tcl/Tk Workshop*. 1996.

[2] Beazley, D. M. & Lomdahl, P. S. "Extensible message passing application development and debugging with python". In *Proceedings of IPPS'97, IEEE Compuer Society*. pp. 650–655. 1997.

[3] Gu, W.; Eisenlianer, G.; Schwan, K. & Vetter, J. "Falcon: on-line monitoring for steering parallel programs". *Concurrency: Practice and Experience*, 1998, 10, 699–736.

[4] Merimaa, J.; Perondi, L. F. & Kaski, K. "An interactive simulation program for visualizing complex phenomena in solids". *Computer Physics Communications*, 2000, 124, 60–75.

[5] Plimpton, S. "Fast parallel algorithms for short-range molecular dynamics". *Journal of Computational Physics*, 1995, 117, 1–19.

[6] Rapaport, D. C. *The Art of Molecular Dynamics Simulation*. Cambridge University Press, Cambridge, UK, 1995.

[7] van Rossum, G. & Drake, F. L. "Extending and embedding the python interpreter". URL `http://www.python.org/doc/2.3.4/ext/ext.html`.

Table 3. Parallel performance of the ADKS simulation engine inside pyMPI with visualization (Intel PIII Xeon cluster).

	speed-up			efficiency		
N	2p	4p	8p	2p	4p	8p
32857	1.90	3.55	7.15	0.95	0.89	0.89
58855	1.84	3.31	5.47	0.92	0.83	0.68
91487	1.83	3.29	5.59	0.91	0.82	0.70
130899	1.85	3.30	5.51	0.93	0.83	0.69