Parallel Implementation of a Monte Carlo Molecular Simulation Program

Alfredo Palace Carvalho, José A. N. F. Gomes, and M. Natália D. S. Cordeiro*

CEQUP/Departamento de Química, Faculdade de Ciências do Porto, Rua do Campo Alegre 687, 4169-007 Porto, Portugal

Received August 11, 1999

Molecular simulation methods such as molecular dynamics and Monte Carlo are fundamental for the theoretical calculation of macroscopic and microscopic properties of chemical and biochemical systems. These methods often rely on heavy computations, and one sometimes feels the need to run them in powerful massively parallel machines. For moderate problem sizes, however, a not so powerful and less expensive solution based on a network of workstations may be quite satisfactory. In the present work, the strategy adopted in the development of a parallel version is outlined, using the message passing model, of a molecular simulation code to be used in a network of workstations. This parallel code is the adaptation of an older sequential code using the Metropolis Monte Carlo method. In this case, the message passing interface was used as the interprocess communications library, although the code could be easily adapted for other message passing systems such as the parallel virtual machine. For simple systems it is shown that speedups of 2 can be achieved for four processes with this cheap solution. For bigger and more complex simulated systems, even better speedups might be obtained, which indicates that the presented approach is appropriate for the efficient use of a network of workstations in parallel processing.

I. INTRODUCTION

In theoretical chemistry, molecular simulations are very important tools for the calculation of equilibrium thermodynamic properties as well as the microscopic structure of chemical and biochemical systems. The methods which are usually used in these calculations are molecular dynamics (MD) and Monte Carlo (MC).[1] These two methods are similar in their aim but follow slightly different approaches.

MD[1] is based on the solution of differential equations of classical mechanics such as Newton's or Lagrange's equations. Thermodynamic properties are calculated in this method by averaging over the trajectories generated by the dynamics.

MC,[2] on the other hand, is a method where the thermodynamic properties are calculated by solving the statistical mechanics' integrals and resorts to the generation of random configurations of the system for a particular ensemble.

Although these two methods differ in their approach, they both give equivalent results by the Gibbs postulate of the statistical mechanics if the simulated system is ergodic.

Any of these two kinds of calculations, however, are generally quite heavy since the simulated systems are necessarily big and the number of cycles required for obtaining a significant sampling are of the order of millions (apart from some initial equilibration steps). Therefore, in some cases, the amount of computations performed by these methods demands the use of more powerful parallel computers.

High-performance computing and parallel processing has been gathering, over these last years, considerable renewed interest. The availability of new types of more affordable parallel systems, such as the network-based clusters of workstations and some less expensive symmetric multiprocessor (SMP) machines, where communication is normally done in this case by shared memory, has made this kind of computing more accessible. Additionally, the development of programming standards, such as message passing interface (MPI),[3,4] parallel virtual machine (PVM),[5] and high-performance Fortran (HPF),[6] which make the writing of parallel codes more portable between different parallel systems, has also contributed largely to this enthusiasm.

Message passing models of parallelization such as MPI and PVM are becoming increasingly more popular, mainly because of their great flexibility. In this model, the various concurrent processes communicate via the exchange of messages, usually through the network, but shared memory may also be used in SMP systems.

This model is suitable for the development of codes based on the multiple instruction multiple data (MIMD) paradigm, in which the various processes are independent from each other, as opposed to the single instruction multiple data (SIMD) paradigm where the various processes must execute the same instructions at the same time, although on different sets of data.[7]

Much of the parallel code developed these days is, however, of the single program multiple data (SPMD) type. This is a special case of the MIMD paradigm, where identical copies of the same program are put in the various processing nodes, although each copy may afterward follow different paths of execution. This implies that they must be effectively independent, unlike the SIMD paradigm.

The fact that the model is very flexible, and that it can now be implemented in a practical and portable way through the use of the well-established MPI standard and the de facto standard PVM, accounts for this popularity. In fact, message passing parallel code, which is typical of distributed memory

* To whom correspondence should be addressed. Phone: +351-22-6082827. E-mail: ncordeir@fc.up.pt.

10.1021/ci990101k CCC: $19.00 © 2000 American Chemical Society
Published on Web 02/17/2000

architectures, is being developed and used nowadays even in shared memory machines. In these types of machines, the traditional way for processes to share data and communicate between them was not by the passing of messages but instead by the use of the operating system's memory sharing facilities. Additionally, before the release of the MPI standard, there were several different message passing parallel programming languages (or dialects of a language), released by the various vendors of the hardware and specific for a particular brand of machine. This made the development of portable parallel code an impossible task. With the implementation of the MPI standard in the various parallel systems, developers were presented a programming interface which is uniform across different types of hardware.

Despite the recent growth of the offer of SMP and NUMA (nonuniform memory access) solutions, one kind of solution for the setup of a parallel system that has been seen as particularly attractive is the assembly of a number of cheap PC-style or Unix workstation machines, operating as the processing elements, interconnected by cheap but reasonably fast communication hardware like Ethernet or Fast-Ethernet.

However, the limited bandwidth of the Ethernet hardware imposes a much lower scalability on the parallel code developed for this kind of setup. Still, for medium size problems that do not require too much communication between the processing elements, this solution is indeed a very interesting one. In addition, this setup could serve as an initial test environment for developing codes to be used in more sophisticated systems afterward. However, in this case one should not neglect the need to adapt the codes in order to take advantage of any features the architecture provides.

In summary, in order to efficiently take advantage of the network of workstations (NOW) as a parallel machine, one has to attempt to optimize the code in the communications side, even more than one usually does in other types of parallel systems where the communication hardware is much faster.

In the present work, an attempt to develop an approach for a parallelization suitable for a NOW is presented. The type of computation that is being parallelized is a molecular simulation's code based on the metropolis MC method. In spite of MC (and even MD, if the goal is to calculate only thermodynamic properties but not dynamic properties) being easily broken into multiple independent runs in a trivial parallelization approach, there are some issues that may justify a less trivial parallelization method. First of all, any simulation (MC or MD) always requires an initial period of equilibration, where the starting positions (and velocities in MD) are allowed to relax to equilibrium values. The breaking of one longer run into several shorter ones means increasing the total amount of equilibration relatively to the useful production CPU time. Additionally, this approach is not suitable for some kinds of simulations (such as free energy calculations, for example).

Performance results will be compared for different types of interconnection hardware, namely, SMP and Fast Ethernet network connected both by a switch or a repeater.

II. PARALLELIZATION STRATEGY

The Monte Carlo method of molecular simulation is suitable for the implementation of a parallelization strategy that attempts to optimize the amount of communication between the processing elements. The usual Metropolis algorithm for the canonical ensemble,[2] for example, can be outlined as follows:

(1) One particle is picked from the set of particles and is moved by a random displacement from its previous position.

(2) The energy of the system in this trial configuration is calculated.

(3) A test is made with the difference between the energies of the trial configuration and the previous one:

(a) If the energy of the trial configuration is lower than the previous one, this movement is accepted as the new configuration.

(b) Otherwise, a test is made, using a random number, to accept the movement with a probability proportional to $e^{-\beta\Delta E}$. If this test is failed, the previous configuration is taken as the new configuration.

(4) The thermodynamic properties of the system in the ensemble are calculated by averaging over the several configurations generated by this algorithm.

The fact that movements of molecules can be carried out one at a time (although they need not to be) makes it possible to distribute the work of interaction calculations between the processors with a little amount of data having to be exchanged between the several processing elements at the end of each MC cycle. These data are, basically, only the collective sum of the energies and, afterward, the update of the moved molecule's position.

The strategy of parallelization of the MC code proposed in this paper will be the following:

(1) One master process will be responsible, in the beginning, for processing the input data and setting up the details for the simulation, and, in the end, for outputting the results.

(2) Several slave processes will be responsible for the calculation of the interactions between the moved particle and only one subset of the whole system's particles.

(3) At each Metropolis MC cycle, the master process will perform the random displacement of one particle and announce the trial position to the slave processes.

(4) The slave processes will make the interaction calculations between the received particle trial position and its particular subset of particles.

(5) The master computes the new energy by summing the interactions collected from the slaves.

(6) The master compares the new energy with the energy of the previous configuration to perform the metropolis test. The master gets this past energy from the slaves, who keep this information for their subset of particles.

(7) The master announces to the slaves whether the new configuration was accepted, in which case, the slave who includes the moved particle in its subset, updates the particle's position.

The amount of communication involved in this scheme consists of an initial big amount of data (but only sent once) which distributes all the details of the simulation, including the particles' initial positions, among the slave processes. Then, the main communication load consists of broadcasts, at each cycle, of the trial position, the collection of the energies calculated by the slaves, and some control messages to notify the slaves of the acceptance/rejection of the movement and also of any error condition that may cause the premature end of the simulation.

The way the whole system particles are to be distributed by the several processes is a matter of crucial importance for a good parallelization. Load balancing, i.e. the even distribution of work to the slaves, is vital for the optimal performance of the parallelization scheme. The situation where some processes spend time doing nothing useful while others still have got enough work to be done should occur the least frequently as possible.

In the scheme that is presented, a good load balancing can be obtained with a very simple distribution method, assuming a homogeneous network environment, i.e. a network composed by nodes similar in their hardware and basic software (operating system, communications and message passing software, etc.). The work is evenly distributed to the slaves, in a static way, simply by attributing to each slave the task of calculating the interactions between the moved particle and one portion of the whole set of particles. For instance, with a system of 100 particles and 2 slave processes, the first process will calculate the interactions for particles 1−50 and the second process for particles 51−100. Since the duration of each interaction calculation is similar, every slave will take about the same time to complete its task provided they all take care of the same number of particles. In a heterogeneous network, instead of a static load distribution, a dynamic distribution can be adopted with few changes: tasks will have a size fixed a priori (instead of being divided evenly by the available processors) and will be attributed to the slaves as they finish computing their previous task and become available for the next. This dynamic scheme will, however, increase the communications overhead in comparison with the static scheme for a homogeneous environment.

In MD calculations, it is common to see the use of a distribution scheme known as the domain decomposition, which consists of distributing the particles to the processes on the basis of its spatial localization. In MD, where all the particles are moved in each cycle, less communication between the processes is necessary to find out the new positions of the particles if the neighboring particles are kept in the same processor. This is, of course, assuming that no long-range interactions are considered in the system, which is not often the case. However, in the present case, no additional benefit is obtained by using the domain decomposition instead of the method that is used here, since in this type of MC calculation only one particle is moved at each cycle (note that this implies that we are defining a MD step to be equivalent to N MC steps, N being the number of particles).

III. RESULTS

The parallelized code was tested in several types of communication hardware, namely, SMP, a four processor SMP system with Intel Pentium Pro 200 MHz CPUs; Fast Ethernet with switch, four Intel Pentium II 266 MHz PCs connected by a 100 Mb/s Ethernet switch; and Fast Ethernet with repeater, four Intel Pentium II 266 MHz PCs connected by a 100 Mb/s Ethernet repeater.

The system with the fastest communications is the SMP machine. It can exchange data by sharing part of the memory, in which case the bandwidth is that of the memory bus (which is typically fast). However, the memory bus is shared by all the processes, which means that the performance degrades with an increasing number of processes. Also, the use of shared memory in MPI is implementation dependent. Some implementations use Unix network sockets instead, which have a bigger overhead than plain shared memory. The implementation of MPI used in this work, LAM version 6.2beta,[8] has the option of using shared memory.

Table 1. Speedups for the Various Parallel Systems and Different Problem Sizes (N_{waters})

N_{waters}	system[a]	t_1/t_2[b]	t_1/t_4[b]	t_2/t_4
512	FE + R	1.38	1.25	
	FE + S	1.36	1.29	
	SMP	1.58	1.16	
1024	FE + R	1.62	1.81	
	FE + S	1.62	1.93	
	SMP	1.70	2.06	
2048	FE + R	1.72	2.42	
	FE + S	1.73	2.56	
	SMP	1.73	2.52	
4096[c]	FE + R			1.70

[a] FE + R = Fast Ethernet with Repeater; FE + S = Fast Ethernet with Switch; SMP = symmetric multiprocessor system. [b] Speedup(n) = t_1/t_n. The speedup with n processes is the ratio between the execution time for 1 process and the execution time for n processes. [c] Benchmark data are not available for the single processor case due to insufficient memory to treat this problem size.

The two Fast Ethernet-based systems differ in the way the machines are interconnected. Both networks have peak performances of 100 Mbit/s and minimum latencies of about 80 μs, but, while in one of them the Ethernet cables are connected by a repeater which simply copies the signals in each cable to all the others, in the other case a switch routes the Ethernet frames to their destination, isolating it from the rest of the participants of the network. The outcome of this is that the switch provides for a network with full 100 Mbit/s bandwidth for all the machines, while the network bandwidth of the Ethernet with the repeater is generally only a fraction of the 100 Mbit/s since it is shared by all the participants of the network, either the machines of the parallel system or the network traffic generated by other machines in the network. In addition, in a congested network, with lots of participants, latencies will become higher when supported by a repeater instead of a switch.

The present tests were performed on Linux systems, with kernel version 2.0.34, using the MPI implementation of LAM version 6.2beta over TCP/IP for the Ethernet clusters and the same version of LAM using the Unix shmem system calls in the SMP machine.

The calculation that served as a test for the performance of the parallel code consisted of a Monte Carlo simulation of a metal cation solvated by a certain number of water molecules (N_{waters} in Table 1). The water molecules were considered to be rigid bodies. The interactions were calculated by fairly standard pairwise additive potentials consisting of simple Coulombic terms and 12−6 Lennard-Jones terms. These types of interactions are the most frequently used in simulations of big systems such as proteins; however, more complex types of interactions (Ewald sums, three-body interactions, etc.) are also common in the simulations of systems such as the one in these tests.

In Table 1 are presented the speedups obtained for the tested computational systems and for two different numbers of water particles. The speedup is defined as the ratio between the time an identical calculation took in a single processor and the corresponding time in N processors.

As one can see, a greater number of particles increases the performance of the parallelization. This is an easily understandable fact since, for the same amount of communication, more work has to be done. Therefore, the ratio between work and communication increases and so does the benefit of parallelizing the code. In the same line of thought, more complex potential functions, which require more expensive computations, will increase the amount of work and the benefits from parallel execution. In the present example, the interactions considered were fairly simple ones. More complex interaction calculations (e.g. electrostatic interactions calculated by Ewald sums) will certainly lead to higher speedups.

The highest speedups are obtained, naturally, on the systems with faster communications' hardware, namely, the shared memory SMP systems, where the communications are made through the memory bus. But it looks promising that, even in the slowest Ethernet connected by a repeater, the speedup profiles seem quite satisfactory, provided the problem has a suitable size to be parallelized.

In comparison to available MD benchmark data[9] obtained in Beowulf systems, the best speedups obtained with the present scheme ($t_2/t_4 = 1.70$ for 4096 waters + 1 metal ion) compare well with speedups for analogous systems ($t_2/t_4 = 1.87-1.94$ for 3830 waters + MbCO), considering that the network used is a Gigabit Ethernet instead of the slower Fast Ethernet and that there is more computation to be performed in the treatment of the intramolecular degrees of freedom (the water molecules are flexible) in comparison with the present MC calculations (where the water molecules are rigid).

It must be noted that, as expected, these cheaper solutions do not present a very good scalability. Especially in the cases with the slowest communication hardware, the gains of parallelizing beyond four processes are somewhat modest (one can even obtain some speeddowns with Ethernet + Repeater and for small problem sizes). Scalability can be enhanced with some improvements in communications, both at the operating system's level and the hardware level (namely, with the use of several network cards), as proposed in the Beowulf project.[10] For problems that require much more computational power and, therefore, the use of a greater number of processes, massively parallel machines may be the only resort. However, since MPI is implemented for most of these machines, the porting of this code for those systems does not impose many important problems. Naturally, a successful parallelization must try to take into account the peculiarities of the hardware as well as the network topology of the parallel machine, trying to adapt the algorithm to it (for instance, to try to restrict communication between processor with closer/faster links). However, the simple scheme proposed does not seem to require any special adaptations. In fact, this code presents quite good scalability when run in a MPP system such as the Cray T3E, as can be seen in Table 2.

The comparison with benchmarks performed in similar systems with well-known MD programs[9,11] shows that the speedups obtained are interesting. Once again, one must remember that, frequently, MD calculations include the treatment of intramolecular terms not accounted for in this example, which would increase the computation/communication ratio and, therefore, lead to a better parallelizable problem with higher speedups.

Table 2. Speedups in the Cray T3E, for the Problem 4096 Waters + Metal Ion[a]

n	t_8/t_n	n	t_8/t_n
8	1.00	32	2.90
16	1.80	64	4.11

[a] Timings are relative to the run in 8 processors.

IV. CONCLUSIONS

In the present work, a scheme was proposed for the parallelization of a Monte Carlo molecular simulation code that would make it suitable to be used in a parallel environment composed of cheap slower communication hardware.

The main aspects of the proposed scheme which can make it appropriate for these types of setups are the particular characteristic of the Metropolis algorithm used, where a particle is moved at a time, diminishing considerably the amount of communication performed at the end of each MC cycle, and a load distribution based on the subdivision of the particles domain, which can easily assure a perfectly balanced work distribution, with little communication overhead if done statically in a homogeneous network. In a heterogeneous network, an extra communication overhead to implement a dynamic load balancing scheme may have an impact on performance.

The proposed parallelization method showed, in an Ethernet based environment, satisfactory speedups of up to four processes when the problem size is not too small. This type of parallel setup is typically not very scalable, because the communications overhead increases with the number of processes due to network packet collisions in the Ethernet, which cause some latency. On the other hand, this overhead is lowered as the amount of calculation is increased between communications at the end of the MC cycles. Therefore, more complex types of interaction functions (e.g. consideration of some intramolecular degrees of freedom or the use of Ewald sums in electrostatic interactions) and the simulation of bigger systems could show much better parallelization efficiencies. And, in fact, it is in these cases that one will want to speed up the execution of the code in a parallel machine.

Better speedups and an overall better scalability can be achieved in shared memory SMP machines, which are nowadays available at very interesting prices, built with cheap PC-style hardware. One should note that the tests for the SMP machine presented in this work were performed on a Intel Pentium Pro system, which has a memory bus working at a 66 MHz frequency and a 32-bit data width (266 MB/s bandwidth). Faster and/or wider data buses are available in the market which can give significantly better performances.

On the other hand, the portability of MPI makes it possible for the code to be run, if needed, on more powerful MPP machines, without the need for many significant modifications in the source code and with a good scaling behavior.

ACKNOWLEDGMENT

A.P.C. wishes to thank the Programa Praxis XXI for a doctoral scholarship. A.P.C. would like to express his gratitude to Prof. Fernando Silva, who guided him through his first steps in parallel programming. He would also like to acknowledge the support by the Edinburgh Parallel Computing Centre, with training as well as access to MPP computational facilities, in helping in the development of the code. Financial support from Fundação para a Ciência e a Tecnologia (Lisbon) through project PRAXIS/PCEX/QUI/61/96 is acknowledged.

REFERENCES AND NOTES

(1) Allen, M. P.; Tildesley, D. J. *Computer Simulation of Liquids*; Oxford University Press: Oxford, U.K., 1987.
(2) Metropolis, N.; Rosenbluth, A. W.; Rosenbluth, M. N.; Teller, A. H.; Teller, E. *J. Chem. Phys.* **1953**, *21*, 1087.
(3) Message Passing Interface Forum, MPI: A Message Passing Interface Standard, June 1995. http://www.mpi-forum.org/docs/mpi-11.ps.Z.
(4) Message Passing Interface Forum, MPI-2: Extensions to the Message-Passing Interface, July 1997. http://www.mpi-forum.org/docs/mpi-20.ps.Z.
(5) Geist, A.; Beguelin, A.; Dongarra, J.; Jiang, W.; Manchek, R.; Sunderam, V. *PVM: Parallel Virtual Machine. A Users' Guide and Tutorial for Networked Parallel Computing*; MIT Press: Cambridge, MA, 1994. Online version available at: http://www.netlib.org/pvm3/book/pvm-book.html.
(6) The High Performance Fortran Forum, The High Performance Fortran Home Page, http://www.crpc.rice.edu/HPFF/home.html.
(7) Kumar, V.; Grama, A.; Gupta, A.; Karypis, G. *Introduction to Parallel Computing: Design and Analysis of Algorithms*; Benjamin/Cummings: Redwood City, CA, 1994.
(8) LAM/MPI Parallel Computing, http://www.mpi.nd.edu/lam/.
(9) Hodoscek, M. Timing in hours for MbCO + 3830 water molecules (14026 atoms), 1000 steps, 12−14 Å shift using standard parallel CHARMM and executed on variety of machines, http://www.ki.si/parallel/summary.html.
(10) Beowulf Project at CESDIS, http://www.beowulf.org/.
(11) Allan, R. J. Timings for DL_POLY_2.0 Benchmarks, http://www.dl.ac.uk/TCSC/Subjects/Parallel_Applications/benchmarks/benchmarks/node56.html.

CI990101K