Experiments With Parallelizing a Tribology Application*

V. Chaudhary, W. L. Hase, H. Jiang, L. Sun, and D. Thaker†
Institute for Scientific Computing
Wayne State University
Detroit, MI 48202
www.isc.wayne.edu

Abstract

Different parallelization methods vary in their system requirements, programming styles, efficiency of exploring parallelism, and the application characteristics they can handle. Different applications can exhibit totally different performance gains depending on the parallelization method used. This paper compares OpenMP, MPI, and Strings(A distributed shared memory) for parallelizing a complicated tribology problem. The problem size and computing infrastructure are changed and their impacts on the parallelization methods are studied. All of the methods studied exhibit good performance improvements. This paper exhibits the benefits that are the result of applying parallelization techniques to applications in this field.

Key Words: Molecular Dynamics, OpenMP, MPI, Distributed Shared Memory.

1 Introduction

Traditionally supercomputers were the tools used to solve so-called "Grand challenge" problems. Recent improvements in processors and networks have provided an opportunity to conduct these experiments within an everyday computing infrastructure by utilizing clusters of symmetrical multiprocessors (SMPs) or even networks of workstations (NOWs). Friction, the resistance to relative motion between sliding surfaces that are in contact, is omnopresent in human life and is an expensive problem facing the industry today. Understanding the origin of frictional forces [14] and energy dissipation during this process [15] has both theoretical and practical importance and has, therefore, attracted considerable interest in the study of tribology. With recent developments of experimental techniques [16] and theories [14], physicists and chemists have been able not only to probe the atomic-level friction process but also to "see" what really takes place at the sliding interface via computer simulation.

In a MD simulation, the motion of each atom is governed by Newton's equations of motion and their positions are determined by the time evolution of the Newton's equation. At each time integration step, the force between atoms, the potential energies and kinetic energies are evaluated. The computational effort grows linearly with the number of Newton's equations, so it is an ideal method to treat mid-sized systems (e.g. 10^2 atoms). However, there are generally two factors limiting the application of MD to large scale simulations (e.g. 10^6 atoms). First, the time step of an integration in a MD simulation is usually about a femtosecond (10^{-15} s). In contrast to this, the time scale for tribology experiments is at minimum, in nanoseconds (10^{-9} s). As a result a large number of integration steps are required to reach a desired total evolution time. Second, when the number of atoms in the simulation system increases, the computation time for force evaluation increases rapidly.

In this paper, we report a practical implementation of parallel computing techniques for performing MD simulations of friction forces between sliding hydroxylated α-aluminum oxide surfaces. Besides system requirements, different parallelization approaches vary in programming style and performance gain. Some methods enable programmers to write code easily, or even provide parallelization service completely transparent to programmers. Other methods might require programmers to put a significant effort in order to achieve substantial gain. The tribology code is written using OpenMP, MPI, and Strings (a software distributed shared memory). OpenMP can be used only for shared memory systems (single SMPs) whereas MPI and Strings can be used for cluster of SMPs as well. The programming paradigms in each of these are very different; with labor requirements ranging from "little" for OpenMP to "large" for MPI.

*This research was supported in part by NSF IGERT grant 9987598, NSF MRI grant 9977815, NSF ITR grant 0081696, NSF grant 0078558, ONR grant N00014-96-1-0866, Institute for Manufacturing Research, and Institute for Scientific Computing.

†Corresponding author: ddthaker@ece.eng.wayne.edu

IEEE COMPUTER SOCIETY
Authorized licensed use limited to: SUNY Buffalo. Downloaded on October 23, 2008 at 12:08 from IEEE Xplore. Restrictions apply.

The remainder of this paper is organized as follows: Section 2 describes various parallelization approaches in high-performance computing, section 3 discusses the molecular dynamics program in detail and how we plan to parallelize it. In Section 4 we present some experiment results and discuss performance benefits. We wrap up with conclusions and continuing work in Section 5.

2 Parallelization Approaches

There are several approaches suitable for transforming sequential Tribology programs into parallel ones. These approaches impose different requirements on compilers, libraries, and runtime support systems. Some of them can execute only on shared memory multiprocessors whereas others can achieve speedups on networks of machines.

2.1 Parallelization with vendors' Support

Some vendors, such as Sun Microsystems, provide compiler or library options for parallel processing. Sun MP C is an extended ANSI C compiler that can compile code to run on SPARC shared memory multiprocessor machines. The compiled code, may run in parallel using the multiple processors on the system [4].

The MP C compiler generates parallel code for those loops that it determines are safe to parallelize. Typically, these loops have iterations that are independent of each other. For such loops, it does not matter in what order the iterations are executed or if they are executed in parallel. This compiler is also able to perform extensive automatic restructuring of user code. These automatic transformations expose higher degrees of loop level parallelization. They include: loop interchange, loop fusion, loop distribution and software pipelining. This C compiler provides explicit and automatic capabilities for parallelizing loops.

2.2 OpenMP

As an emerging industry standard, OpenMP is an Application Program Interface (API) that may be used to explicitly direct multi-threaded, shared memory parallelism in C/C++ and Fortran on all architectures, including Unix platforms and Windows NT platforms. It is comprised of three primary API components: compiler directives, runtime library routines, and environment variables. OpenMP is a portable, scalable model that gives shared-memory parallel programmers a simple and flexible interface for developing parallel applications for platforms ranging from the desktop to the supercomputer [6]. OpenMP utilizes the fork and join model of parallel computing (see Figure 1).

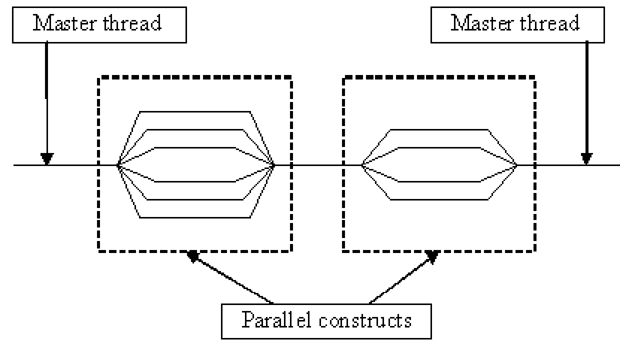

Figure 1. OpenMP execution model

2.3 MPI

MPI is a message-passing application programmer interface, together with protocol and semantic specifications for how its features must behave in any implementation (as a message buffering and message delivery progress requirement) [9]. The main advantages of establishing a message-passing standard are portability and ease-of-use. In a distributed memory communication environment in which the higher level routines and/or abstractions are build upon lower level message passing routines the benefits of standardization are particularly apparent. Furthermore, the definition of a message passing standard provides vendors with a clearly defined base set of routines that they can implement efficiently, or in some cases provide hardware support for, thereby enhancing scalability [8].

2.4 Distributed Shared Memory (DSM) systems

Developing parallel applications using Distributed Shared Memory systems is easier when compared to developing the same applications using MPI. Since hardware shared memory machines do not scale well and are relatively expensive to build, software distributed shared memory (DSM) systems are gaining popularity for providing a logically shared memory over physically distributed memory. These software DSM systems combine programming advantages of shared memory and the cost advantages of distributed memory. The programmer is given the illusion of a large global address space encompassing all available memory, thereby eliminating the task of explicitly moving data between processes located on separate machines.

Research projects with DSMs have shown good performance, for example *TreadMarks* [12], *Millipede* [11] and *Strings* [13]. This model has also been shown to give good results for programs that have irregular data access patterns which cannot be analyzed at compile time, or indirect data accesses that are dependent on the input data-set.

We parallelize the tribology program by using a multi-

IEEE COMPUTER SOCIETY

Authorized licensed use limited to: SUNY Buffalo. Downloaded on October 23, 2008 at 12:08 from IEEE Xplore. Restrictions apply.

threaded DSM, *Strings*, designed for clusters of Symmetrical Multiprocessors (SMPs). *Strings* was developed at Wayne State University and consists of a library that is linked with a shared memory parallel program. *Strings* is built using POSIX threads, which can be multiplexed on kernel lightweight processes. The kernel can schedule these lightweight processes across multiple processors on symmetrical multiprocessors (SMPs) for better performance. Therefore, in *Strings*, each thread could be assigned to any processor on the SMP if there is no special request, and all local threads could run in parallel if there are enough processors. *Strings* is designed to exploit data parallelism by allowing multiple application threads to share the same address space on a node. Additionally, the protocol handler is multi-threaded. The overhead of interrupt driven network I/O is avoided by using a dedicated communication thread. *Strings* is designed to exploit data parallelism at the application level and task parallelism at the run-time level.

Strings starts a master process that forks child processes on remote nodes using rsh(). Each of these processes creates a dsm_server thread and a communication thread. The forked processes then register their listening ports with the master. The master process enters the application proper and creates shared memory regions. It then creates application threads on remote nodes by sending requests to the dsm_server threads on the respective nodes. Shared memory identifiers and global synchronization primitives are sent as part of the thread create call. The virtual memory subsystem is used to enforce coherent access to the globally shared regions.

2.4.1 Shared memory

Strings implements shared memory by using the mmap() call to map a file to the bottom of the stack segment. Allowing multiple application threads on the same node leads to a peculiar problem. Once a page has been fetched from a remote node, its contents must be written to the corresponding memory region, so the protection has to be changed to writable. At this time no other thread should be able to access this page. Suspending all kernel level threads can lead to a deadlock and also reduce concurrency. In Strings, every page is mapped to two different addresses. It is then possible to write to the shadow address without changing the protection of the primary memory region.

A release consistency model using an update protocol has been implemented. When a thread tries to write to a page, a twin copy of the page is created. When either a lock is released or a barrier is reached, the difference (diff) between the current contents and its twin are sent to threads that share the page. Multiple diffs are aggregated to decrease the number of messages sent.

3 Molecular Dynamics

3.1 Model system

The sequential code has been used to study the friction force of sliding hydroxylated α-aluminum surfaces. Structure of an α-aluminum surface has been described in detail before [18]. The model system consists of a smaller block of Al_2O_3 surface (upper surface) moving on a much larger slab of Al_2O_3 surface(bottom surface). The broken bonds at the contacting surfaces are saturated by bonding with H atoms. To simulate experiments, pressure is applied on top of the upper surface and the driving force that moves the upper surface with respect to the bottom surface is added to the system. By selecting "iop" options as described in the code [26], different pressure and driving forces, i.e. different energy dissipative systems, are selected. Besides the driving force that moves the upper sliding surface, each atom in the system is exposed to the interaction with other atoms. The general types of interaction can be divided into two categories: intramolecular bonded and inter-molecular non-bonded forces. The bonded forces are represented by internal coordinate bond distance, bond angles, and constants determined by the interacting atoms. The inter-molecular forces are Van der Waals interaction. The simulation are carried out with a constant number of atoms, constant volume and constant temperature (NVT). Temperature control is achieved by Berenden's method [19]. The integration of Newton's equation of motion is done by using Velocity Verlet algorithm [20].

3.2 Simulation procedure

The simulation is carried out by solving the classical equations of motion. Initial velocities are either set to zero or calculated by the program according to the user's demand. Newton's equation is numerically integrated to predict the position of all atoms in the next short period of time. The atomic forces are evaluated during each of the integration step. In the hydroxylated α-alumina systems, the type of forces are bonded and non-bonded. The sequential code used in the tribology study here has the structure depicted in Figure 2 [26].

3.2.1 Bonded forces calculation

The interactions between adjacent atoms connected by chemical bonds are described by bonded forces. The bonded forces are two-centered harmonic stretches with three centered harmonic bends. Their interaction potential functions are modelled by harmonic potential energy functions

Proceedings of the International Conference on Parallel Processing Workshops (ICPPW'02)
1530-2016/02 $17.00 © 2002 IEEE

IEEE COMPUTER SOCIETY
Authorized licensed use limited to: SUNY Buffalo. Downloaded on October 23, 2008 at 12:08 from IEEE Xplore. Restrictions apply.

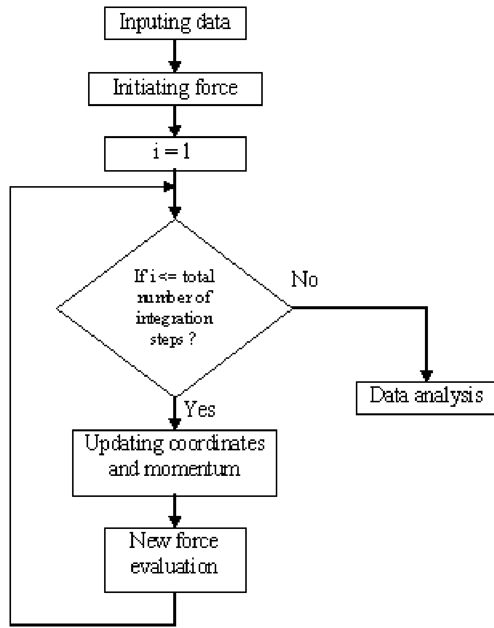

Figure 2. Flow chart of MP simulation

$$V_{str} = \frac{1}{2}k_{str}(r - r_0)^2 \quad (1)$$

where k_{str}, r and r_0 are bond stretching force constant, bond length, and equilibrium bond distance and

$$V_\theta = \frac{1}{2}k_\theta(\theta - \theta_0)^2 \quad (2)$$

where k_θ, θ and θ_0 are the bond angle bending force constant, bond angle, and equilibrium bond angle, respectively.

The forces are assigned to each involved atom by taking the first derivatives of the potential.

3.2.2 The nonbonded calculation

The nonbonded interactions here contain only Lennard-Jones type of potentials

$$V_{L-J}(r_{ij}) = 4\varepsilon[\frac{\sigma}{r_{ij}^6} - \frac{\sigma}{r_{ij}^{12}}] \quad (3)$$

where r_{ij} is the distance between atom i and atom j. ε and σ represent the nonbonded iteraction parameters.

Although the computation effort for bonded interactions grows linearly with the size of the system, the nonbonded interaction exhibits a quadratic dependence on the number of atoms. Hence, the evaluation of the nonbonded Lennard-Jones terms are generally the most computationally intensive constituent in the MD code.

Lennard-Jones type of interaction is long range interaction that vanishes slowly at large distance. To reduce the computation effort for calculating the small forces on atoms at large distance, a cut-off radius is generally introduced. A neighbor search is carried out to find the atoms within the cut off radius. By introducing the cut off radius the computational effort scales linearly with the number of atoms. However, the nonbonded force is still the most time consuming part in the each iteration of the force evaluation.

3.3 Implementation

There are various data partition schemes in parallel molecular dynamics simulation[8-12]. In general three parallel algorithms are often used to decompose and distribute the computational load.

First, the number of atoms in the simulation system is equally divided and assigned to each processor; Second, the forces of interaction are equally divided and assigned to each processor; Third, the spacial region is equally divided and assigned to each processor. Each algorithm has its advantages and therefore they are often implemented according to the specific problem under study, i.e., system size and evolution time. For example, when use MPI to implement the third method, the molecular system are divided into subspaces, each processor calculates the forces on the atoms within the subspace and update the corresponding positions and velocities. However, the extent of forces always cover the neighboring subspaces or even the whole space, the updating of forces on atoms requires communication at least among neighboring subspaces at each integration step. The cost increases with number of processors and increase in size of integration steps. Therefore, this algorithm is often used for large molecular system with relatively fewer integration steps.

In the tribology application considered here, the evaluation of forces (98-99% execution time) is the most time consuming. So the parallelization is focused on evaluation of forces. To compare the performance between OpenMP, MPI, and DSM Strings methods, the basic parallel algorithm is maintained. Forces on atoms are evenly assigned to each processor. For bonded forces, the computational load on each processor/threads equals the number of harmonic stretch forces divided by the number of processors/threads in MPI, OpenMP, and Strings. For the nonbonded force terms, there are two situations. The nonbonded interaction with the same surfaces are distributed to each processor/thread in the same way as for bonded forces. The Lennard-Jones interactions between different surface atoms are calculated by searching the neighbor list and therefore the atom dividing scheme is employed. There are obvious shortcomings for this simple algorithm for both MPI and DSM Stings implementation. Even though the force calculation is divided into small parts, the communication between all processors to update the coordinates has to be

IEEE COMPUTER SOCIETY
Authorized licensed use limited to: SUNY Buffalo. Downloaded on October 23, 2008 at 12:08 from IEEE Xplore. Restrictions apply.

done at each integration step. Therefore, it is necessary for comparison to be done for different size of system and different time integration step.

4 Experiments and Analysis

The computing environment used and the analysis of data from the experiments is described in this section.

4.1 Experiment Infrastructure

The experiments were carried out using a cluster of SMPs. The SMPs used were a SUN Enterprise E6500 with 14 processors (4Gbytes of RAM) , and three SUN Enterprise E3500s with 4 processors (and 1Gbytes of RAM) each. Each of these processors were 330 MHz Ultra-SparcIIs. The operating system on the machines was Sun Solaris 5.7. The interconnect was fast ethernet using a NetGear switch.

The MPICH implementation was used for MPI. The OpenMP code was compiled using the SUN High Performance Compiler. Both the MPI and the Strings version of the application were also run on the large SMP in order to compare their performance with OpenMP. Two data sizes, one small another large were used. The comparisons were done for the application on one node using one, two, four and eight processors each, on two nodes with one, two and four processors each and finally on four nodes with one, two and four processors each.

4.2 Results and Analysis

This section describes the results. In case of one large SMP, it can be seen from Figures 3 and 4, that immaterial of the problem size, the results are consistent. OpenMP outperforms the others on the large SMP. For OpenMP, the SUN High Performance Compiler was used, which was able to optimize it for the SUN Enterprise machines. For MPI, we used the MPICH implementation, which being portable loses out on performance compared to OpenMP. The performance for MPI and Strings is very similar on one SMP.

When considering multiple SMPs, we could only use the MPI version and the Strings version of the application. We used up to four SMPs each with four processors. Again for both program sizes, the results are consistent. For MPI, it was observed that performance degraded when we used 4 processes per nodes, for both 2 nodes and 4 nodes. This can be directly attributed to the substantial increase in communication as seen from Figures 7 and 8. Another observation was that for MPI, increasing the number of processes per machine increases the total communication time. This is because the MPI code uses MPI_Reduce and MPI_Broadcast calls at the end of each computation cycle. This is an area where performance could be improved by using other MPI primitives.

For the distributed shared memory (Strings) version of the application, it can be seen that increasing the number of compute threads always results in an increase in performance. As we increase the number of nodes that the application uses, the performance degrades as this increases communication. For example, the application on 1 machine and 4 compute threads performs better than on 2 machines with 2 compute threads, which in turn is better than 4 machines with 1 compute thread. This shows that within an SMP, Strings is able to effectively use shared memory to communicate. Another interesting observation was that the total execution time when using 4 compute threads on 4 machines, is very close to the execution time when using 2 compute threads on 4 machines. It can be seen from Figure 10, that increasing the number of nodes increases the number of page faults, both read and write.

In the final analysis, it can be seen that Strings outperforms MPI for this application by a big margin when running on a cluster of SMPs. The fraction of time spent in communication for Strings is much less than that of MPI (see Figures 7, 8, and 9). Also using the SUN High Performance Compiler and OpenMP provides the best results for a single SMP.

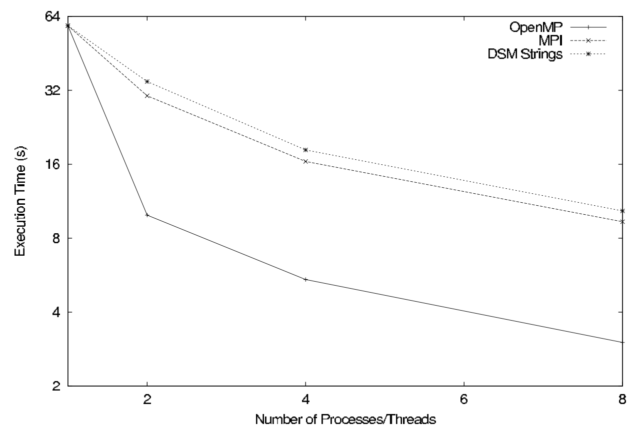

Figure 3. The smaller MD program executed on 1 node.

5 Conclusion and future work

This paper compared OpenMP, MPI, and Strings based parallelization for a tribology application. These parallelization methods vary in their system requirements, programming styles, efficiency of exploring parallelism, and the application characteristics they can handle. For OpenMP and Strings, one writes threaded code for an

IEEE COMPUTER SOCIETY

Authorized licensed use limited to: SUNY Buffalo. Downloaded on October 23, 2008 at 12:08 from IEEE Xplore. Restrictions apply.

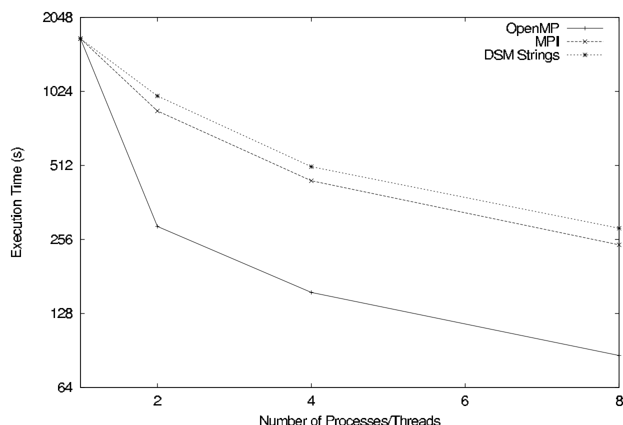

Figure 4. The bigger MD program executed on 1 node.

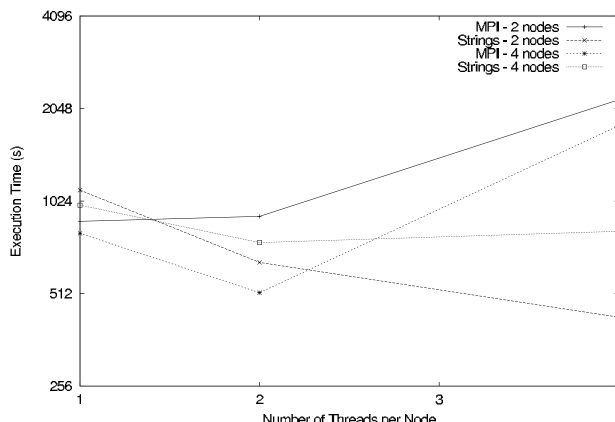

Figure 6. The bigger MD program executed on 2 and 4 nodes.

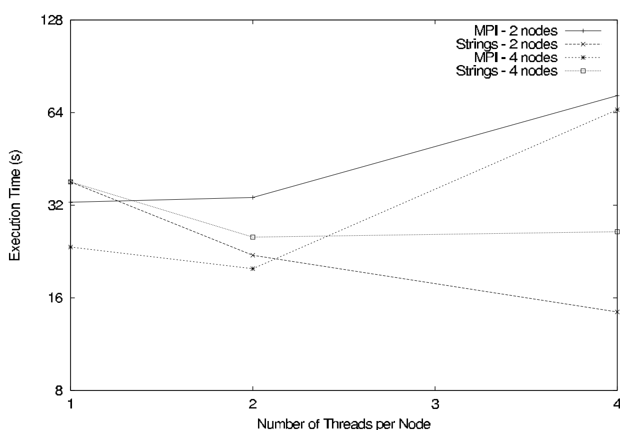

Figure 5. The smaller MD program executed on 2 and 4 nodes.

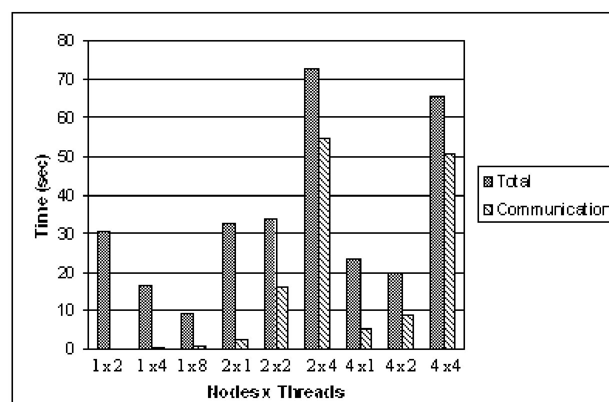

Figure 7. MPI communication time in the smaller MD execution.

SMP and they are relatively easy to program. MPI on the other hand requires writing a program with message passing primitives and is more cumbersome to program. The effort in programming is least for OpenMP and most for MPI. For SMPs, the SUN High Performance Compiler and OpenMP provides the best results for a single SMP. For cluster of SMPs, Strings outperforms MPI for this application by a big margin when running on a cluster of SMPs.

It appears that combining OpenMP and Strings would yield best results for a cluster of SMPs. We are currently implementing OpenMP and Strings together. Also, we are looking into different types of parallelization of the tribology code. One method would divide the atoms in the simulations equally among the processors. Another method would divide the spatial region equally among the processors.

References

[1] Sumit Roy, Ryan Yong Jin, Vipin Chaudhary, and William L. Hase, Parallel Molecular Dynamics Simulations of Alkane/Hydroxylated α-Aluminum Oxide Interfaces, *Computer Physics Communications*, pp. 210-218, 128 (2000).

[2] D. Cociorva, G. Baumgartner, C. Lam, P. Sadayappan, J. Ramanujam, M. Nooijen, D. Bernholdt and R. Harrison, Space-Time Trade-Off Optimization for a Class of Electronic Structure Calculations, *Proceedings of ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation (PLDI)*, June 2002, To Appear.

[3] G. Baumgartner, D. Bernholdt, D. Cociorva, R. Harrison, M. Nooijen, J. Ramanujam and P. Sadayappan, A Performance Optimization Framework for Compilation of Tensor Contraction Expressions into Parallel Programs, *7th International Workshop on High-Level Parallel Programming Models and Supportive Environments (held in conjunction with IPDPS '02)*, April 2002.

IEEE COMPUTER SOCIETY

Authorized licensed use limited to: SUNY Buffalo. Downloaded on October 23, 2008 at 12:08 from IEEE Xplore. Restrictions apply.

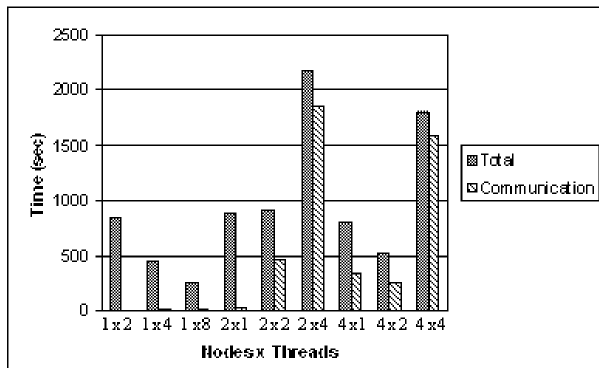

Figure 8. MPI communication time in the bigger MD execution.

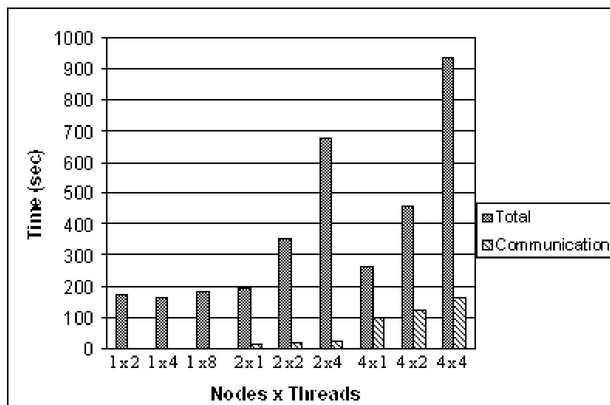

Figure 9. DSM communication time in a certain MD execution.

[4] Vinod Grover and Douglas Walls, Sun MP C Compiler, http://docs.sun.com/.

[5] Sun Microsystems, Sun Performance Library User's Guide, http://docs.sun.com/.

[6] The OpenMP Forum, OpenMP: Simple, Portable, Scalable SMP Programming, http://www.openmp.org/.

[7] National Energy Research Scientific Computing Cetner, NERSC OpenMP Tutorial, http://hpcf.nersc.gov/training/tutorials/openmp/.

[8] Message Passing Interface (MPI) Forum, MPI: A message-passing interface standard, *International Journal of Supercomputing Applications*, 8(3/4), 1994.

[9] William Gropp and Ewing Lusk and Nathan Doss and Anthony Skjellum, High-performance, portable implementation of the MPI Message Passing Interface Standard, *Parallel Computing*, Vol. 22, 6, 1996.

[11] A. Itzkovitz, A. Schuster, and L. Wolfovich, Thread Migration and its Applications in Distributed Shared Memory Systems, *Journal of Systems and Software*, vol. 42, no. 1, pp. 71–87, 1998.

[12] P. Keleher, A. Cox, S. Dwarkadas and W. Zwaenepoel, TreadMarks: Distributed Shared Memory on Standard Workstations and Operating Systems, *Proc. of the Winter 1994 USENIX Conference*, 1994.

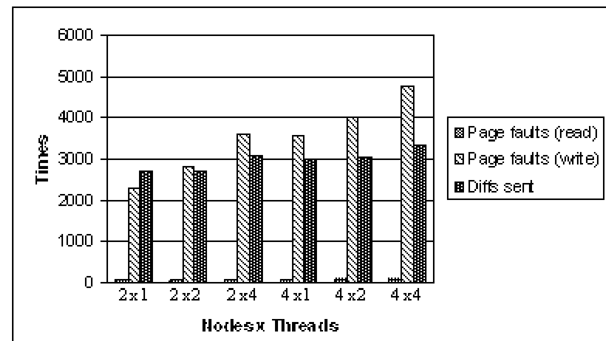

Figure 10. DSM Strings statistics in a certain MD execution.

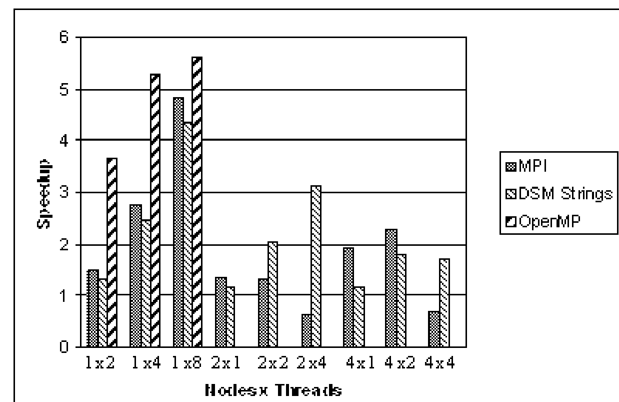

Figure 11. Speedups for the smaller MD program.

[13] S. Roy and V. Chaudhary, Design Issues for a High-Performance DSM on SMP Clusters, *Journal of Cluster Computing*, 2(1999) 3, 1999, pp. 177–186.

[14] Mark O. Robbins, Martin H. Mser, Computer Simulation of Friction, Lubrication, and Wear, *Modern Tribology Handbook*, B. Bhushan, Ed. (CRC press, Boca Raton, 2001) pp. 717.

[15] I. L. Singer, Friction and Energy Dissipation at the Atomic Scale: A Review, *Journal of Vacuum Science and Technology* , A. 12, 5 1994.

[16] Robert W. Carpick, Miquel Salmeron, Scratching the Surface: Fundamental Investigations of Tribology with atomic Force Microscopy, *Chemical Review*, pp. 1163-1194, 97, 1997.

[17] Parallel Computing in Computational Chemistry: developed from a symposium sponsored by the Division of Computers in Chemistry at the 207th National Meeting of the American Chemical Society, San Diego, California, March 13-17, 1994, Timthy G. Mattson, Ed., Washington, DC, American Chemical Society.

IEEE COMPUTER SOCIETY

Authorized licensed use limited to: SUNY Buffalo. Downloaded on October 23, 2008 at 12:08 from IEEE Xplore. Restrictions apply.

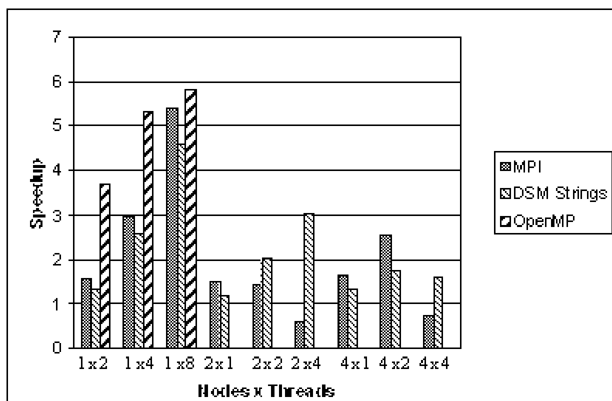

Figure 12. Speedups for the bigger MD program.

[18] J. M. Wittbrodt, W. L. Hase, H. B. Schlegel, Ab Initio Study of the Interaction of Water with Cluster Models of the Aluminum Terminated (0001) α-Aluminum Oxide Surface, *Journal of Physical Chemistry B*, pp. 6539-6548,102 1998.

[19] H. J. C. Berendsen, J. P. M. Postma, W. F. van Gunsteren, A. Di-Nola, J. R. Haak, Molecular-Dynamics With Coupling to an External Bath, *Journal of Chemical Physics*, pp. 3684-3690, 81, 1984.

[20] W. C. Swope, H. C. Andersen, P. H. Berens, K. R. Wilson, A Computer-Simulation Method for the Calculation of Equilibrium-Constants for the Formation of Physical Clusters of Molecules- Application to Small Water Clusters, *Journal of Chemical Physics*, pp. 637-649, 76, 1982.

[21] S. Y. Liem, D. Brown, J. H. R. Clarke, A Loose-Coupling, Constant Pressure, Molecular Dynamics Algorithm for Use in the Modeling of Polymer Materials, *Computer Physics Communication*, pp. 360-369, 62, 1991.

[22] D. Brown, J. H. R. Clarke, M. Okuda, T. Yamazaki, A Domain Decomposition Parallel-Processing Algorithm for Molecular Dynamics Simulations of Polymers, *Computer Physics Communication*, pp. 1-13, 83 1994.

[23] D. Brown, J. H. R. Clarke, M. Okuda, T. Yamazaki, A Domain Decomposition Parallelization Strategy for Molecular Dynamics Simulations on Distributed Memory Machines, *Computer Physics Communication*, pp. 67-80, 74, 1993.

[24] S. Plimpton, Fast Parallel Algorithms for Short Range Molecular Dynamics, *Journal of Computational Physics*, pp. 1-19, 117, 1995.

[25] R. Murty, D. Okunbor, Efficient Parallel Algorithms for Molecular Dynamics Simulation, *Parallel Computing*, pp. 217-230, 25, 1999.

[26] V.Chaudhary, W.L.Hase, H.Jiang, L.Sun, D.Thaker, Comparing Various Parallelizing Approaches for Tribology, Technical Report 02-03-61, Parallel and Distributed Computing Laboratory, Wayne State University, 2002

IEEE COMPUTER SOCIETY

Authorized licensed use limited to: SUNY Buffalo. Downloaded on October 23, 2008 at 12:08 from IEEE Xplore. Restrictions apply.