The Journal of Supercomputing, 28, 323–343, 2004

Experiments with Parallelizing Tribology Simulations

V. CHAUDHARY* vipin@wayne.edu
Institute for Scientific Computing, Wayne State University, Detroit, MI 48202

W. L. HASE bill.hase@ttu.edu
Department of Chemistry and Biochemistry, Texas Tech University

H. JIANG hai@wayne.edu
Institute for Scientific Computing, Wayne State University, Detroit, MI 48202

L. SUN lpsun@chem.northwestern.edu
Department of Chemistry, Northwestern University

D. THAKER ddthaker@ece.eng.wayne.edu
Institute for Scientific Computing, Wayne State University, Detroit, MI 48202

Abstract. Different parallelization methods vary in their system requirements, programming styles, efficiency of exploring parallelism, and the application characteristics they can handle. For different situations, they can exhibit totally different performance gains. This paper compares OpenMP, MPI, and Strings for parallelizing a complicated tribology problem. The problem size and computing infrastructure is changed to assess the impact of this on various parallelization methods. All of them exhibit good performance improvements and it exhibits the necessity and importance of applying parallelization in this field.

Keywords: molecular dynamics, OpenMP, MPI, distributed shared memory

1. Introduction

Traditionally supercomputers are the essential tools to solve these so-called "Grand challenge" problems. Recent improvements in commodity processors and networks have provided an opportunity to conduct these kind of task within an everyday computing infrastructure, such as symmetrical multiprocessors (SMPs) or even networks of workstations (NOWs).

Friction, the resistance to relative motion between contact sliding surfaces, happens everywhere in human life and it is a costly problem facing industry. Understanding the origin of friction force [13] and the energy dissipation during the friction process [14], therefore, has both theoretical and practical importance and it has attracted considerable interest in tribology study. A complete understanding of

*Author for correspondence.

these friction processes requires detailed information at the atomic level. With recent development of experimental techniques [1, 2, 3, 15] and the theories [13, 21–24], physicists and chemists have been able not only to probe the atomic-level friction process but also to "see" what really takes places at the sliding interface via computer simulation. Because computer simulation which utilizes the molecular dynamics (MD) method can follow and analyze the dynamics of all atoms, it is has become a powerful tool to investigate various tribology phenomena.

In a MD simulation, the motion of each atom is governed by the Newton's equation 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 simulation (e.g., 10^6 atoms). First, the time step of integration in a MD simulation is usually about a femtosecond (10^{-15} s) due to the numerical and physical consideration. In contrast to simulation, the time scale for tribology experiments is at least in nanoseconds (10^{-9} s). The time step in simulation is so small compared with experiment that it generally requires a large number of integration steps 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.

Parallel and distributed processing is promising solution for these computational requirements discussed above [16]. Significant advances in parallel algorithms and architecture have demonstrated the potential for applying current computation techniques into the simulation of the friction process. Among the different approaches of parallel processing, there are numerous implementation cases for MD simulation using MPI, the parallelization with thread method so far is limited.

In this paper, we report a practical implementation of parallel computing techniques for performing MD simulations of friction forces of sliding hydroxylated α-aluminum oxide surfaces. There are many systems and tools exploring parallelism for different types of programs. Besides system requirements, different parallelization approaches vary in programming style and performance gain. Some methods enable programrs to write code easily, or even provide parallelization service completely transparent to programmers. But normally these kind of methods cannot provide expected performance improvement all the time. Other methods might require programmers to put reasonable effort in order to achieve substantial gain.

The tribology code is written using OpenMP, MPI, and DSM Strings (a software distributed shared memory). OpenMP can be used only for shared memory systems whereas MPI and Strings can be used for both shared memory systems and network of workstations. The programming paradigms in each of these are very different with the labor requirements ranging from "little" for OpenMP to "large" for MPI. The programming effort for Strings is considerably less than MPI. Therefore, to evaluate an approach's ability to exploit parallelism for a particular application domain, many factors need to be factored in, including system requirement, programming style, time to program, performance gain, etc.

The remainder of this paper is organized as follows: Section 2 describes various

parallelization approaches in high-performance computing. In Section 3 we discuss molecular dynamics program in detail and how we plan to parallelize it. Section 4 presents some experiment results and discuss the performance. 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 workstations.

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.

Sun Performance Library can be used with the shared or dedicated modes of parallelization, that are user selectable at link time. The dedicated multiprocessor model of parallelism has the following features: Specifying the parallelization mode improves application performance by using the parallelization enhancements made to Sun Performance Library routines [5].

The shared multiprocessor model of parallelism has the following features:

- Delivers peak performance to applications that do not use compiler parallelization and that run on a platform shared with other applications.
- Parallelization is implemented with threads library synchronization primitives.

The dedicated multiprocessor model of parallelism has the following features:

- Delivers peak performance to applications using automatic compiler parallelization and running on an multiprocessor platform dedicated to a single processor-intensive application.
- Parallelization is implemented with spin locks.

On a dedicated system, the dedicated model can be faster than the shared model due to lower synchronization overhead. On a system running many different tasks, the shared model can make better use of available resources.

To specify the parallelization mode:

- Shared model—Use -mt on the link line without one of the compiler parallelization options.
- Dedicated model—Use one of the compiler parallelization options [-xparallel | -xexplicitpar | -xautopar] on the compile and link lines.
- Single processor—Do not specify any of the compiler parallelization options or -mt on the link line.

Due to the potential of aliasing in programming languages, it is especially hard to determine the safety of parallelization. Normally vendors' compilers and libraries do not offer any capabilities to automatically parallelize arbitrary regions of code. Therefore, this loop parallelization strategy can achieve good performance only when many big loops exist and their iterations are independent or with regular dependency patterns.

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. Jointly defined by a group of major computer hardware and software vendors, 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].

The OpenMP API defines a set of program directives that enable the user to annotate a sequential program to indicate how it should be executed in parallel. In C/C++, the directives are implemented as #pragma statement, and in Fortran 77/90 they are implemented as comments. A program that is written using OpenMP directives begins execution as a single process, called the master thread of execution. The master thread executes sequentially until the first parallel construct is encountered. The PARALLEL/END PARALLEL directive pair constitutes the parallel construct. When a parallel construct is encountered, the master thread creates a team of threads, and the master thread becomes the master of the team. The program statements that are enclosed in a parallel construct, including routines called from within the construct, are executed in parallel by each thread in the team.

Upon completion of the parallel construct, the threads in the team synchronize and only the master thread continues execution. Any number of parallel constructs can be specified in a single program. As a result, a program may fork and join many times during execution (see Figure 1).

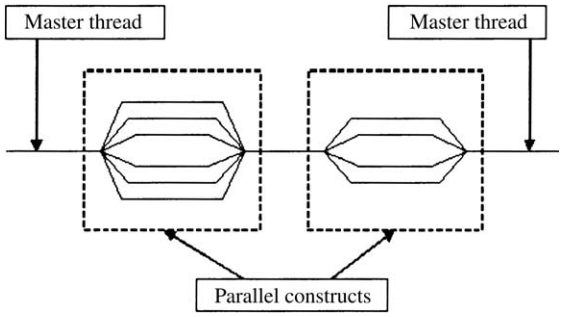

Figure 1. OpenMP execution model.

The degree of parallelism in an OpenMP code is dependent on the code, the platform, the hardware configuration, the compiler, and the operating system. In no case are you guaranteed to have each thread running on a separate processor [7].

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 built 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].

MPI includes point-to-point message passing and collective operations, all scoped to a user-specified group of processes. MPI provides process abstraction at two levels. First, processes are named according to the rank of the group in which the communication is being performed. Second, virtual topologies allow for graph or Cartesian naming of processes that help relate the application semantics to the message passing semantics in a convenient, efficient way.

MPI also provides some additional services: environmental inquiry, basic timing information for application performance measurement, and a profiling interface for external performance monitoring. To support data conversion in heterogeneous environments, MPI requires datatype specification for communication operations. Both built-in and user-defined datatypes are provided.

MPI supports both SPMD and MPMD modes of parallel programming. Furthermore, MPI supports communication between groups and within a single group. MPI provides a thread-safe API, which is useful in multi-threaded environments.

Unlike OpenMP which could only be used in shared-memory systems, MPI can work for both shared-memory and distributed memory models. Therefore, MPI can utilize processors in single workstation or network of workstations.

2.4. Distributed shared memory (DSM) systems

Applications for distributed memory systems are cumbersome to develop due to the need for programmers to handle communication primitives explicitly, just as coding in MPI. In addition, applications have to be tuned for each individual architecture to achieve reasonable performance. Since hardware shared memory machines do not scale well and are relatively expensive to build, software 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 [11], Millipede [10] and Strings [12]. 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.

DSMs share data at the relatively large granularity of a virtual memory page and can suffer from a phenomenon known as "false sharing", wherein two processes simultaneously attempt to write to different data items that reside on the same page. If only a single writer is permitted, the page may ping-pong between the nodes. One solution to this problem is to "hold" a freshly arrived page for some time before releasing it to another requester. Relaxed memory consistency models that allow multiple concurrent writers have also been proposed to alleviate this symptom. The systems ensure that all nodes see the same data at well defined points in the program, usually when synchronization occurs. Extra effort is required to ensure program correctness in this case. One technique that has been investigated to improve DSM performance is the use of multiple threads of control in the system. Up to now, the third generation DSM systems utilize relaxed consistency models and multithreading technologies.

We parallelize the tribology program by using a multithreaded 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. The program thus uses calls to the distributed shared memory allocator to create globally shared memory regions.

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 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. Kernel threads. Thread implementations can be either user-level, usually implemented as a library, or kernel-level in terms of light-weight processes. Kernel level threads are more expensive to create, since the kernel is involved in managing them. User level threads suffer from some limitations, since they are implemented as a user-level library, they cannot be scheduled by the kernel. If any thread issues a blocking system call, all associated threads will also be blocked. Also on a multi-processor system, user-level threads bound to a light-weight process can only on one processor at a time. User level threads do not allow the programmer to control their scheduling within the process, on the other hand kernel level threads can be scheduled by the operating system across multiple processors.

2.4.2. Shared memory. Strings implements shared memory by using the mmap() call to map a file to the bottom of the stack segment. With dynamically linked programs, it was found that mmap() would map the same page to different addresses on different processors. 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 [17]. 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 (see Appendix A), 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 nonbonded 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 [18]. The integration of Newton's equation of motion is done by using Velocity Verlet algorithm [19].

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 (see Appendix A for more details).

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 modeled by harmonic potential energy functions

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

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

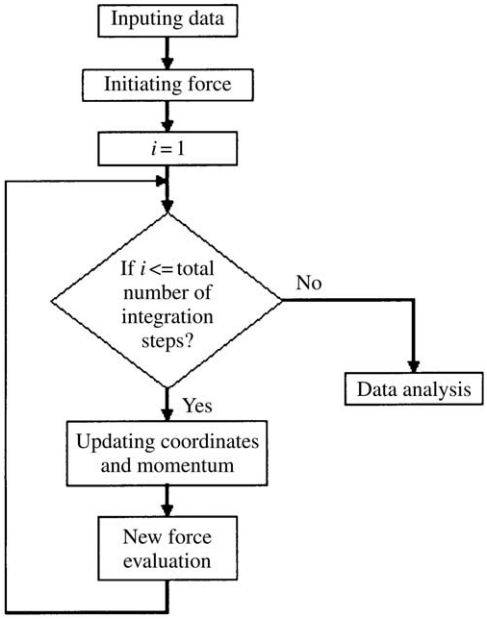

Figure 2. Flow chart of MP simulation.

bond distance and

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

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

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

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. Lennard–Jones interaction beyond the cut-off distance is then treated as zero. Therefore, 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 using 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 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 (4 Gbytes of RAM), and three SUN Enterprise E3500s with four processors (and 1 Gbyte 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 Net-Gear switch.

The application was run sequentially, using OpenMP (on the large SMP), using MPI (the MPICH implementation was used) on the cluster of SMPs, and using Strings on the same cluster. 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 that were obtained from the experiments described earlier. 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, as shown in Figures 5 and 6.

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 four processes per nodes, for both two nodes and four 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

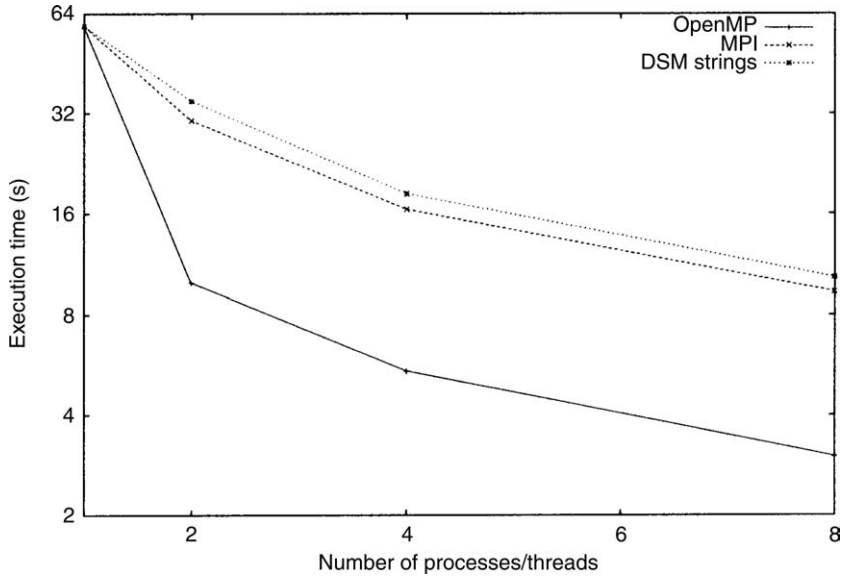

Figure 3. The smaller MD program executed on one node.

performance. As we increase the number of nodes that the application uses, the performance degrades as this increases communication. For example, the application on one machine and four compute threads performs better than on two machines with two compute threads, which in turn is better than four machines with one compute thread. This shows that within an SMP, Strings is able to effectively use

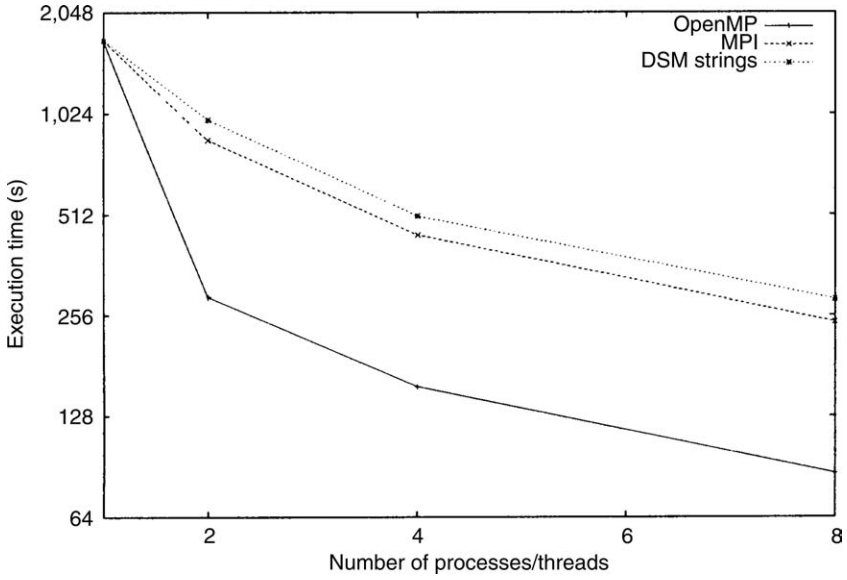

Figure 4. The bigger MD program executed on one node.

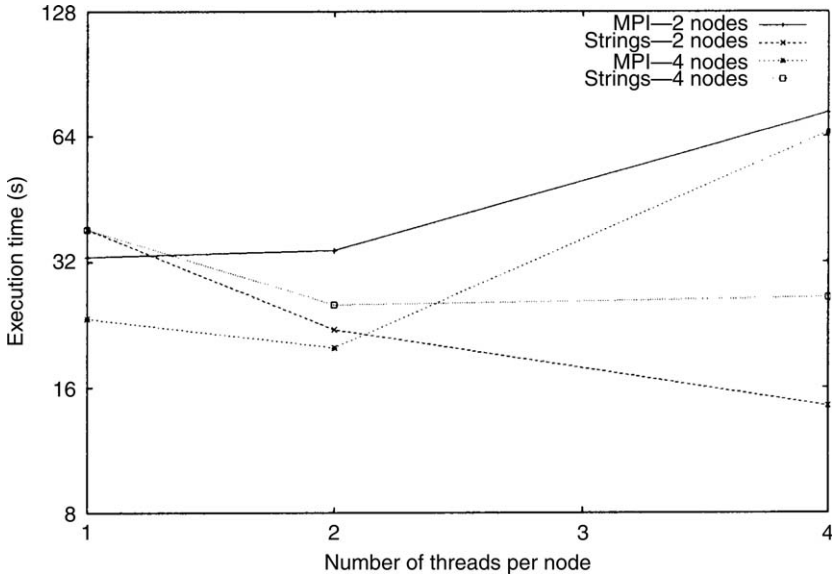

Figure 5. The smaller MD program executed on two and four nodes.

shared memory to communicate. Another interesting observation was that the total execution time when using four compute threads on four machines, is very close to the execution time when using two compute threads on four machines. In Figure 10, it can be seen that increasing the number of nodes increases the number of page faults, both read and write.

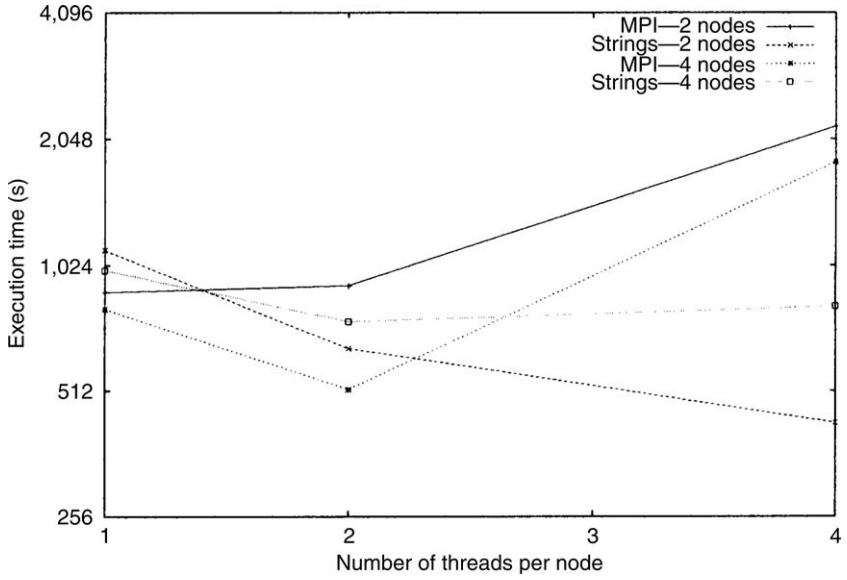

Figure 6. The bigger MD program executed on two and four nodes.

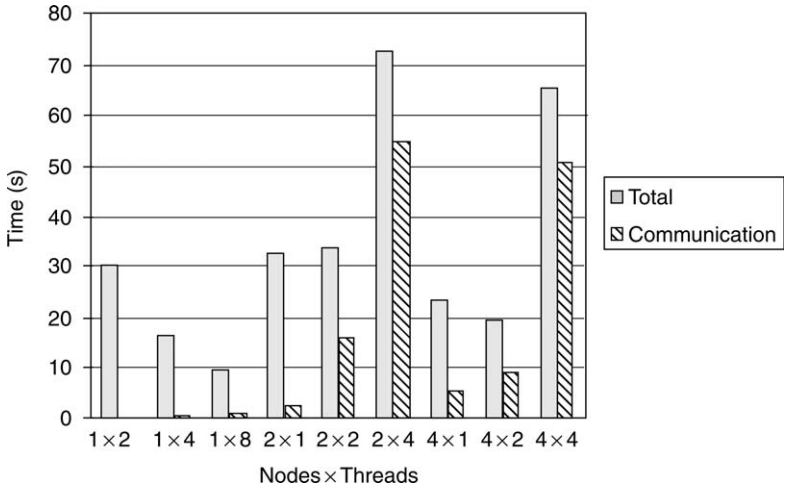

Figure 7. MPI communication time in the smaller MD execution.

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.

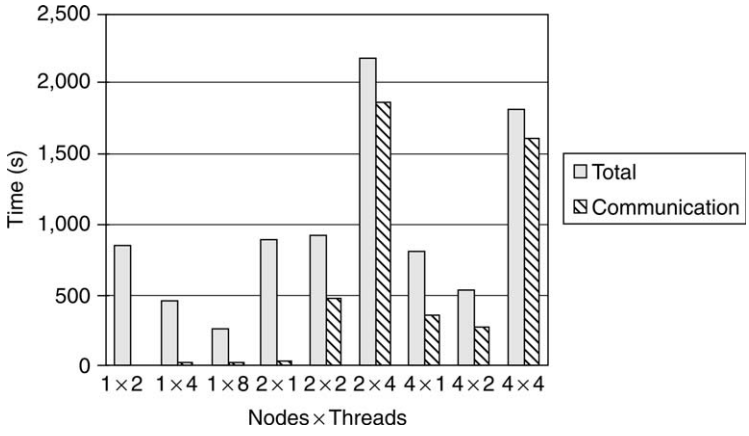

Figure 8. MPI communication time in the bigger MD execution.

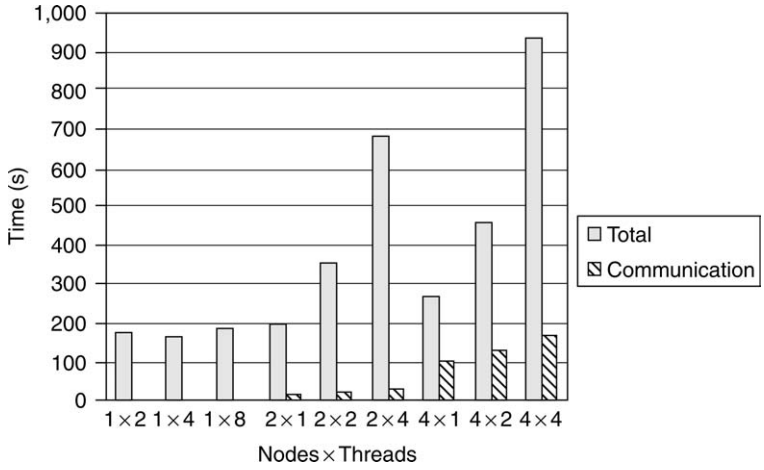

Figure 9. DSM communication time in a certain MD execution.

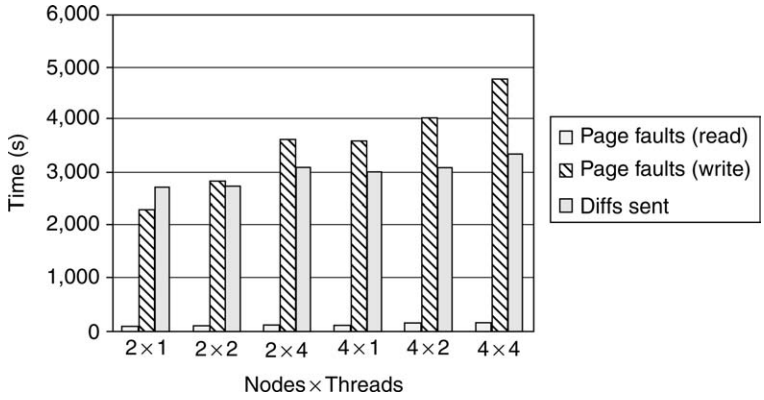

Figure 10. DSM Strings statistics in a certain MD execution.

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 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

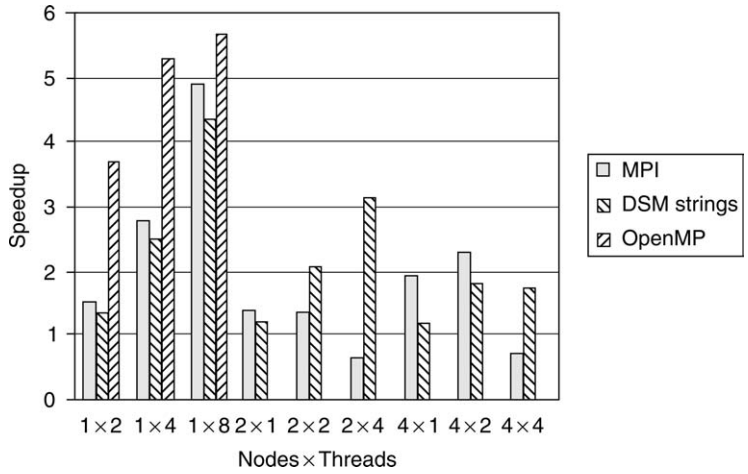

Figure 11. Speedups for the smaller MD program.

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.

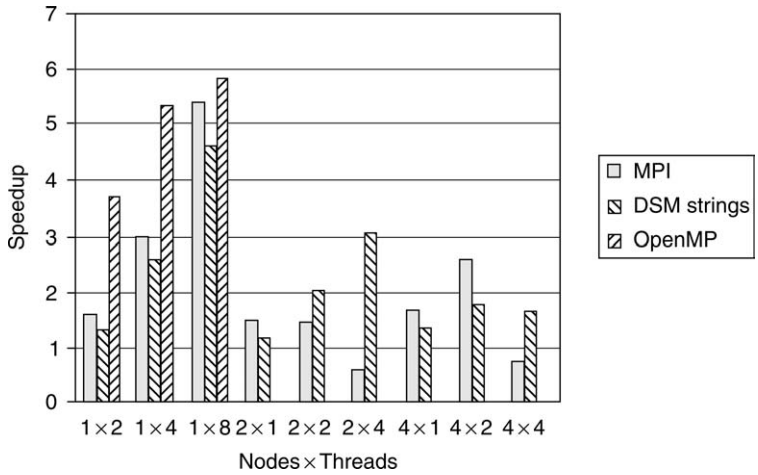

Figure 12. Speedups for the bigger MD program.

Appendix

Appendix A. Sequential code

```
program tribology

program main ()
read in information required for computation
when time = 0    -----> initial condition
call force  -----> calculate forces
call energy  -----> calculate energies
write initial results
if iop* = 1, 2 ...      -----> select surface sliding conditions
  do i = 1, i< = total number of integration steps
    call verlet  -----> velocity verlet algorithm to update
            -----> coordinates and momenta
    if print data = true
       call energy
       write intermediate results
    endif
  enddo
else
write ''input iop error''
endif
end program main

subroutine verlet()
if iop = 1, 2 ...
    adjust velocities for different iops
endif
do i = 1, number of atoms
    update the velocity of each atom
    except thermal bath  -----> velocity verlet algorithm
enddo
call force               -----> force evaluation
do i = 1, number of atoms
    update momenta of each atom
    except thermal bath  -----> velocity verlet algorithm
enddo
if iop = 1, 2 ...
    adjust momenta
    apply Berendsen methods
endif
end verlet subroutine

subroutine forces()
```

```
do i = 1, number of stretches  -----> harmonic strech forces
   r[i] = ...      -----> bond length calculation
   f[i] = constant1* (r[i]-constant2)^2  -----> force evaluation
enddo
do i = 1, number of bending terms  -----> harmonic bending forces
   angle [i] = ...  ----->bond angle calculation
   f[i] = constant1* (angle [i]-constant2)  -----> force evalua-
tion
enddo
call itralj          ----->intra-molecular Lennar_Jones forces
do i = 1, number of 1j forces
   1j evaluation
  enddo
  call interlj  -----> inter-molecular Lennard-Jones forces
  do i = 1, number of atoms
     build neighbor count list of 1j terms
     calculate 1j forces
  enddo
  end force subroutine

  subroutine energy()
     total energy = potential energy + kinetic energy
  end energy subroutine

*iop is the option to select different model systems

iop = 1: constant load, pull center of mass of upper surface
iop = 2: constant load, pull outer layer atoms in the upper surface
iop = 3: constant distance, pull center of mass of the upper surface
iop = 4: constant distance, pull outer layer atoms in the upper
surface

where load is the force applied to the upper surface that brings two
surfaces together and the pull force is the lateral force that moves
the upper surface with respect to the bottom surface so that upper
surface can slide on the bottom surface.
```

Appendix B. OpenMP code

```
parallel for loops in force subroutine, master thread takes care of
the rest.
#pragma omp parallel for default (shared) private (local variable)
        reduction (+:inter)

do i = 1, number of forces terms (strech, bend, Lennard-Jones)
```

```
    local variable = local variable evaluation
    r[i] = ...
    angle[i] = ...
    f[i] = function of (local variables, r[i], angle[i], etc)
    inter = + calculated intermediate physical quantities
enddo
```

Appendix C. MPI code

```
MPI initialization
if processor id = 0
       call readin()        -----> read in data
  write initial information
endif
do i = 1, total number of integration steps
    if processor id = 0
      MPI_Bcast initial coordinates
    endif
  localload = calculate work load for each processor
  do i = 1, locaload  -----> for each processor
      local_r[i] = ...
      local_angle[i] = ...
      . . . . .
      local[i] = . . . .
  enddo
  localload = distribute evenly the atoms in neighborlist to each
processor
  do i = 1, localload
      f[i] = Lennard-Jones terms on each processor
  enddo
  if processor id = 0
      MPI_Reduce  -----> force summation on to processor 0
      MPI_Gather  -----> gather bond length, angle and
      ----->other variables
      update coordinates and momenta using
      velocity verlet algorithm
      call energy to calculate energy
      write intermediate results
    endif
enddo
MPI_Finalize
```

Appendix D. Strings DSM code

```
DSM initialization
Create DSM global regions for shared data
fork threads on local and remote machines
for all threads
    if current thread is the first one on current node
       call readin()  -----> read in data
      write initial information
    endif
    do i = 1, total number of integration steps
      reach a barrier to wait for all threads' arrival
      localload = calculate work load for each processor
      do i = 1, locaload  -----> for each processor
        local_r[i] = ...
        local_angle[i] = ...
        .....
        local[i] = ....
      enddo
      localload = distribute evenly the atoms in neighborlist to each
processor
    do i = 1, localload
      f[i] = Lennard-Jones terms on each processor
    enddo
    acquire critical section lock
       update coordinates and momenta using
       velocity verlet algorithm
       call energy to calculate energy
       write intermediate results
    release critical section lock
  enddo
  DSM terminates
endfor
```

Acknowledgments

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.

References

1. S. Roy, R. Y. Jin, V. Chaudhary, and W. L. Hase. Parallel molecular dynamics simulations of alkane/hydroxylated α-aluminum oxide interfaces. *Computer Physics Communications*, 128:210–218, 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.
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.
4. V. Grover and D. 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. W. Gropp, E. Lusk, N. Doss, and A. Skjellum. High-performance, portable implementation of the MPI message passing interface standard. *Parallel Computing*, 22(6), 1996.
10. A. Itzkovitz, A. Schuster, and L. Wolfovich. Thread migration and its applications in distributed shared memory systems. *Journal of Systems and Software*, 42(1):71–87, 1998.
11. P. Keleher, A. Cox, S. Dwarkadas, and W. Zwaenepoel. TreadMarks: Distributed shared memory on standard workstations and operating systems. *Proceeding of the Winter 1994 USENIX Conference*, 1994.
12. S. Roy and V. Chaudhary. Design issues for a high-performance DSM on SMP clusters. *Journal of Cluster Computing*, 2(3):177–186, 1999.
13. M. O. Robbins and M. H. Mser. Computer simulation of friction, lubrication, and wear. In B. Bhushan, ed., *Modern Tribology Handbook*, p. 717. CRC press, Boca Raton, 2001.
14. I. L. Singer. Friction and energy dissipation at the atomic scale: A review. *Journal of Vacuum Science and Technology*, *A*, 12:5, 1994.
15. R. W. Carpick and M. Salmeron. Scratching the surface: Fundamental investigations of tribology with atomic force microscopy. *Chemical Review*, 97:1163–1194, 1997.
16. 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, Timothy G. Mattson, ed., Washington, DC, American Chemical Society. March 13–17, 1994.
17. J. M. Wittbrodt, W. L. Hase, and 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*, 102:6539–6548, 1998.
18. H. J. C. Berendsen, J. P. M. Postma, W. F. van Gunsteren, A. Di-Nola, and J. R. Haak. Molecular-dynamics with coupling to an external bath. *Journal of Chemical Physics*, 81:3684–3690, 1984.
19. W. C. Swope, H. C. Andersen, P. H. Berens, and 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*, 76:637–649, 1982.
20. S. Y. Liem, D. Brown, and J. H. R. Clarke. A loose-coupling, constant pressure, molecular dynamics algorithm for use in the modeling of polymer materials. *Computer Physics Communication*, 62:360–369, 1991.
21. D. Brown, J. H. R. Clarke, M. Okuda, and T. Yamazaki. A domain decomposition parallel-processing algorithm for molecular dynamics simulations of polymers. *Computer Physics Communication*, 83:1–13, 1994.
22. D. Brown, J. H. R. Clarke, M. Okuda, and T. Yamazaki. A domain decomposition parallelization strategy for molecular dynamics simulations on distributed memory machines. *Computer Physics Communication*, 74:67–80, 1993.
23. S. Plimpton. Fast parallel algorithms for short range molecular dynamics. *Journal of Computational Physics*, 117:1–19, 1995.
24. R. Murty and D. Okunbor. Efficient parallel algorithms for molecular dynamics simulation. *Parallel Computing*, 25:217–230, 1999.