Chapter 3 Massively Parallel Molecular Simulation Program

3.1 Introduction

The work on developing our primary simulation tool builds up on many years of work in the Materials and Process Simulation Center at Caltech and particularly the programming efforts of Kian-Tat Lim and is more extensively described by Lim et.al.

More recently, the MPSim program has been used for a wide variety of other general purpose molecular dynamics simulations, including study of nanotubes by Gao and study of the mechanism of phosphoglycerate kinase (PGK) enzyme by Vaidehi et al.

3.2 Structure of the MPSim program

The basic structure of the MPSim program is shown on Fig 3.1. Fig 3.2 depicts the structure of the energy evaluator.

3.3 Message-passing implementation

The original design of the message passing implementation was to use active messages. Active messages were the preferred model for programming the experimental fine grain J- and M-machines from MIT. The low latency of the active messages programming model, along with its natural, asynchronous, multithreaded style of programming made this model a prime candidate for preferred programming model for the then-upcoming generations of supercomputers.
On the other hand, recent development in parallel programming has placed emphasis on more traditional message-passing models, such as PVM, NX, and MPI. MPI especially has been adopted as the *de facto* standard with support from all vendors. Native active-message libraries did not materialize for machines to which we have access (Cray T3D, Intel Paragon, and Silicon Graphics.) Therefore we have decided to proceed with using the MPI message-passing library. This ensures high degree of portability and allows for future optimization by using more advanced collective operations provided by the MPI library.

As a first implementation we have preserved the active-message paradigm and especially the “pull” strategy as described in Lim et.al. Here is an example of using the “pull” strategy to compute the kinetic energy of a system.

*For each node* compute its local kinetic energy.

*If I am not node 0, then* {

Send my local kinetic energy to node 0 and

Wait for receiving back the global kinetic energy.

}

*Else* {

Start a loop for receiving from N-1 nodes a double number and add them together.

Broadcast the global energy to all nodes

}
The above algorithm can be optimized by using the MPI global operations, such as MPI_Reduce. The flow chart for the current implementation using the simplest MPI calls is given in Appendix A.

### 3.4 Shared memory implementation

**Portable shared memory programming model**

Shared memory programming models have the advantage of greatly simplifying the initial parallel-programming task. However attention must be paid to the actual location of data, to ensure high parallel efficiency without extensive rewriting of code. Another challenge comes from the fact that there is no standard for shared memory programming. In contrast there are many standards for message-passing programming, e.g. MPI or PVM. Since portability is extremely important for us, we have attempted to formulate a simple shared memory programming model which presents to the program standardized interface and hides low level machine specific features from the rest of the program. The idea is that the standardized interface consists in a number of machine independent function calls, which then provide the necessary machine dependent code in the body of the function. A shared memory programming model requires two kinds of software support: locks and barriers. Barriers are generally provided by all thread libraries, so we just implement a `do_sync()` function which does barrier check in and immediately follow it by check out. Locks are handled differently on different machines. For example on SGI architectures they need to be allocated from arena space, whereas on KSR machines any processor can lock any 128-byte subpage of memory.
To ensure portability we define a set of macros:

lock_structure(a)
unlock_structure(a)
lock_variable(a)
unlock_variable(a)

Then on each machine we expand those macros to do the proper machine-dependent code. The exact expansion is given in appendix B.

Care must be taken also in dealing with private data, which is generally used to compute global sums. (Each CPU computes its own value for the data, which are later added together globally.) The following macros are defined to help with global sums:

```
def_local (Type,a); /*defines a of type Type and local for each CPU*/
local(a) = 5 ;     /* assigns our local copy of a to 5 */
global_sum (g,a) ; /* sums all local a’s globally */
```

Efficiency considerations for shared memory on various machines.

Just as for sequential programs it is important to keep data local in order to avoid cache thrashing (also known as false cache sharing.) On the KSR computer 128-byte subpages tend to migrate to the processor that needs them most often. That means we should make sure we don’t have two processors accessing the same 128-byte subpage with approximately the same frequency.
On the SGI Power Challenge and Origin architecture a memory page is 16K. This presents a particular problem with the implementation of private data. On the SGI machine there is no "private" primitive so we implement private variables as arrays. Our implementation of the "local(private_a)" macro expands to \texttt{private_a [my_cpu]}, where \texttt{private_a} is defined as an array of dimension the maximum number of CPUs. To ensure efficient memory management it is necessary that each element of the array be of size 16K (one page.) Thus it is more efficient to group all private variables in a structure and then have just one array of structures.

\textbf{3.5. Implementation on machines that provide one-sided communication.}

Some machines like the Cray T3D, provide one-sided communication (shared memory get and put in Cray's parlance.) This programming model is somewhere in the middle between the shared memory and message passing programming models. The algorithm presented on Fig.3.5.1 was designed to perform non-local updates. It can be shown that this algorithm terminates for all processors and produces the correct updated value of X. Moreover, for most of the cases when there is no contention for simultaneous update from two processors, this algorithm is much faster than the traditional, message passing approach which would require sending, receiving and processing a request.
Fig. 3.5.1 Common location update algorithm for architectures that provide one-sided communication.
3.5 Performance results

3.5.1 Performance on shared-memory architectures

Figure 3.2 Performance on the Kendall Square computer.
3.5.2 Performance on message-passing architectures

Figure 3.2 Comparative performance on various architectures using message-passing libraries.

The program provides general-purpose force and energy evaluation engine and a set of bookkeeping utilities, so that another modules can be added with relative ease. The flowchart on Figure 3.6.1 is an example how a driver that implements a different integration method (these are sometimes called *movers* since they move the atoms to new positions. A pseudo-code implementing this algorithm is given in appendix C. It is also possible to interface to the program via a macro language, which is described in appendix D.
Initialize all utilities.

Read Force Field; Read Structure; Set up Energy Expression; Compute degrees of freedom; Initialize velocities

Set up non bonds
Set up hydrogen bonds

Calculate Potential Energy and Forces (optionally compute stresses, if necessary)

Perform the moving operation

Perform bookkeeping operations:
Load balancing, lists reassignment, Garbage collection and saving of desired dynamics variables.

Check termination condition: number of iterations, desired RMS, etc.

Need new setups?
Yes

New module. All the other modules are implemented in MPSim as libraries.

No

End

Yes

No

Figure 3.6.1 Flowchart for the API interface to the MPSim program
http://www.wag.caltech.edu/research.html

\[\text{Lim, K.T., Ph.D. Thesis, Caltech 1995}\]


\[\text{Vaidehi, N, Goddard, W., in preparation.}\]