A PRIMER FOR PARALLEL IMPLEMENTATION OF MOLECULAR DYNAMICS SIMULATIONS

David Keffer
Department of Chemical Engineering
University of Tennessee, Knoxville
Developed: February, 2003

Table of Contents:

INTRODUCTION

These notes are intended to address some practical issues facing people who want to convert an existing code designed to run on a single-processor to a code designed to run on parallel machines.

The examples used here are molecular dynamics simulation codes written in FORTRAN 90. All massively parallel machines have both C and FORTRAN compilers. FORTRAN 90 is used rather than FORTRAN 77 because FORTRAN 90 can dynamically allocate arrays, therefore, a code can easily be written so that it does not need to be recompiled when the number of available processors changes.

We use the MPI (message passing interface) library of subroutines rather than PVM, because every massively parallel machine has MPI. It seems to be the library of choice. There are applications in which PVM is superior. (Dr. Mark Rader works on a good example in SERF.)

In these codes, all processors see and execute the same code. The numerical values of the variables on each process will be different, but each processor is executing each line of the source code, unless a line is specifically designated for only a particular processor. We shall see that later.

We assume, as is the case today, that speed—not memory—is our bottleneck. Therefore, in these codes we keep global copies of all positions on each node. For simulations up to 10,000 molecules, this presents no problem whatsoever.

I. USEFUL MPI SUBROUTINES FOR MD SIMULATIONS

In this section I list all of the MPI routines I use in any of my current molecular dynamics codes.

These routines are:

1. MPI_INIT
2. MPI_COMM_RANK
3. MPI_COMM_SIZE
4. MPI_WTIME
5. MPI_BCAST
6. MPI_SCATTERV
7. MPI_ALLGATHERV
8. MPI_ALLREDUCE
9. MPI_FINALIZE

In the following examples, all variables starting with letters a-h or o-z are double precision variables. All variables starting with letters i-n are integers.

In order to use any of these MPI routines, every program, subroutine, and function that calls an MPI routine must include the header file, 'mpif.h', immediately after the implicit statement, i.e. the third line of the routine.

```
program prog_001
implicit double precision (a-h,o-z)
include 'mpif.h'
```

I.A. Initializing MPI

1. MPI_INIT

usage:

```
call MPI_INIT(ierr)
```

This routine initializes MPI
This routine is called once, before any other MPI routines are called.

ierr = 0 if the subroutine exits without error

2. MPI_COMM_RANK

usage:

```
call MPI_COMM_RANK(MPI_COMM_WORLD, irank, ierr)
```

This routine assigns a unique number from 0 to N_{proc}-1 (in the variable named irank) to each processor.
This routine is called once, immediately after MPI_INIT.

MPI_COMM_WORLD = MPI intrinsic variable, defined and used by MPI
irank = rank of processor
ierr = 0 if the subroutine exits without error

One processor will be the root processor. You can set a variable iroot = 0, in which case when you have a task that you want only iroot to perform, you can write something like

```
if (irank .eq. iroot) then
        ... only iroot processor does this...
endif
```

3. MPI_COMM_SIZE

usage:

```
call MPI_COMM_SIZE(MPI_COMM_WORLD, nproc, ierr)
```

This routine determines the number of processors and stores that number in the variable named nproc.
This routine is called once, immediately after MPI_COMM_RANK.

MPI_COMM_WORLD = MPI intrinsic variable, defined and used by MPI
nproc = number of processors
ierr = 0 if the subroutine exits without error

4. MPI_WTIME

usage:

```
time_start = MPI_WTIME()
... some fortran code in here ...
time_end = MPI_WTIME()
time_elapsed = time_end - time_start
```

This routine returns the current time.
This routine is called twice. First it is called immediately after MPI_COMM_SIZE. Second it is called immediately before MPI_FINALIZE which terminates the program.
An advantage of this timer is that it is available for any system where MPI is available. Other timers, like etime, are system specific and may or may not be available.

I.B. Communicating between processors

5. MPI_BCAST

usage:

```
      call MPI_BCAST(rx_glob,Ng, MPI_DOUBLE_PRECISION,
     &  iroot, MPI_COMM_WORLD, ierr)
```

This routine broadcasts a vector of values from the processor with irank=iroot to all other processors. In this example, it broadcasts a double precision vector of length Ng named rx_glob.

MPI_DOUBLE_PRECISION is a default MPI variable
MPI_INTEGER is another useful default MPI variable

6. MPI_SCATTERV

usage:

```
      call MPI_SCATTERV(rx_glob,nlongA,nposA, MPI_DOUBLE_PRECISION,
     &  rx_local,Nlocal, MPI_DOUBLE_PRECISION, iroot, MPI_COMM_WORLD, ierr)
```

This routine scatters a vector of values from the processor with irank=iroot to all other processors. Each processor gets a different piece of the source vector. In this example, the iroot processor broadcasts a double precision vector of length Ng named rx_glob to a vector called rx_local of length Nlocal. The value of Nlocal may be different on each processor.

The vector nlongA (of length nproc) provides the value of Nlocal for each processor.
The vector nposA (of length nproc) provides the starting position less one of the each processors data as stored in the global vector.

For example, if rx_glob is a vector of length Ng=40, which we want to split evenly among nproc=4 processors then nlongA =(10,10,10,10) and nposA=(0,10,20,30). In this manner
processor 0 get rx_glob(1:10) stored in rx_local(1:10)
processor 1 get rx_glob(11:20) stored in rx_local(1:10)
processor 2 get rx_glob(21:30) stored in rx_local(1:10)
processor 3 get rx_glob(31:40) stored in rx_local(1:10).

A second example, if rx_glob is a vector of length Ng=15, which we want to split as evenly as possible among nproc=4 processors then nlongA =(4,4,4,3) and nposA=(0,4,8,12). In this manner, we have that
processor 0 gets rx_glob(1:4) stored in rx_local(1:4)
processor 1 gets rx_glob(5:8) stored in rx_local(1:4)
processor 2 gets rx_glob(9:12) stored in rx_local(1:4)
processor 3 gets rx_glob(13:15) stored in rx_local(1:3).

7. MPI_ALLGATHERV

usage:

```
      call MPI_ALLGATHERV(rx_local, Nlocal, MPI_DOUBLE_PRECISION,
     &  rx_glob,nlongA,nposA, MPI_DOUBLE_PRECISION, MPI_COMM_WORLD, ierr)
```

This routine gathers a scattered vector of values from all processors, and gathers them back into the global vector, storing a copy on each processor. In this example, the scattered vector is a double precision vector of length Nlocal (again Nlocal may be different on each processor) called rx_local. That vector is gathered into a vector of length Ng, rx_glob. The vectors nlongA and nposA are the same vectors described in MPI_SCATTERV.

For example, if rx_local is a vector of length 4 on processors 0, 1, and 2, but of length 3 on processor 3, then rx_glob is a vector of length Ng=15. We have nlongA =(4,4,4,3) and nposA=(0,4,8,12). In this manner, we have that all processors get a copy of rx_glob where
rx_glob(1:4) = rx_local(1:4) from processor 0
rx_glob(5:8) = rx_local(1:4) from processor 1
rx_glob(9:12) = rx_local(1:4) from processor 2
rx_glob(13:15) = rx_local(1:3) from processor 3

8. MPI_ALLREDUCE

usage:

```
      call MPI_ALLREDUCE(xlocal, xtot, Ng, MPI_DOUBLE_PRECISION,
     &  MPI_SUM, MPI_COMM_WORLD, ierr)
```

This routines performs a matrix addition, where you are adding copies of xlocal stored on each processor and storing the resulting sum in a vector of the same length and type name xtot. A copy of xtot exists on all processors.

I.C. TERMINATING MPI

9. MPI_FINALIZE

usage:

```
      call MPI_FINALIZE(ierr)
```

Generally this routine appears once immediately before the FORTRAN ‘stop’ statement. It terminates processor communication.

II. PRACTICAL MACHINE SPECIFIC COMMANDS

II.A. Eagle (eagle.ccs.ornl.gov)

machine: a 184-node IBM RS/6000 SP
relevant webpages: http://www.ccs.ornl.gov/Eagle.html

II.A.1. How to connect

for interactive window:
ssh –l username eagle.ccs.ornl.gov
for file transfer:
sftp eagle.ccs.ornl.gov

Educators and students can download a free non-commercial version of ssh (including sftp) that works on Windows operating systems, at least 98, 2000, and XP (those are the OS's that we tested) from www.ssh.com. There are many other freeware versions of ssh but this one is the most user friendly and, besides that, it is free. ssh comes by default on Linux operating systems.

II.A.2. How to compile and link

location: ~dkeffer/prog_001

directory contents:

```
-rw-rw-r--   1 dkeffer  users       74292 Feb 12 09:58 adriver.f
-rw-rw-r--   1 dkeffer  users       11887 Feb 12 09:58 linkedcell.f
-rw-rw-r--   1 dkeffer  users        1050 Feb 05 12:18 makefile
-rw-r--r--   1 dkeffer  users        2040 Feb 05 12:21 md.in
-rw-rw-r--   1 dkeffer  users         342 Feb 05 12:20 md_inter.cmd
-rw-rw-r--   1 dkeffer  users       68486 Feb 12 09:58 md_mix_v17.f
-rw-rw-r--   1 dkeffer  users        1412 Feb 12 09:58 md_mpi_extras.f
-rw-r--r--   1 dkeffer  users       11408 Feb 12 09:58 onsagerL.f
-rw-r--r--   1 dkeffer  users        6617 Feb 12 09:58 onsagerL_corr.f
-rw-rw-r--   1 dkeffer  users         755 Feb 12 09:58 pbc_multi.f
-rw-rw-r--   1 dkeffer  users        7976 Feb 12 09:58 self_d_corr.f
-rw-r--r--   1 dkeffer  users       15998 Feb 12 09:58 transport_d.f
-rw-r--r--   1 dkeffer  users        4521 Feb 12 09:58 transport_dmut.f
```

source code is located in all of the *.f FORTRAN files

command: make mddriver

The make command uses the macros defined in the file "makefile" to compile and link the code. The contents of the makefile are shown in Figure 1.
Additional comments on non-intuitive traits of the makefile in Figure 1.

1) Tabs appear after all colons (:). If you use spaces, it won't work.
2) If a space appears after back-slash (\), it won't work.
3) The pound sign (#) at the start of a line comments out the line.
4) The locations of the libraries in Figure 1 are specific to eagle.

The actual commands generated by typing "make mddriver" are given below. It compiles each source file (*.f). Then it links all the files and libraries.

```
mpxlf90  -qfixed=132 -O5 -qnoipa -bmaxdata:0x40000000   -c md_mix_v17.f
mpxlf90  -qfixed=132 -O5 -qnoipa -bmaxdata:0x40000000   -c transport_d.f
mpxlf90  -qfixed=132 -O5 -qnoipa -bmaxdata:0x40000000   -c md_mpi_extras.f
mpxlf90  -qfixed=132 -O5 -qnoipa -bmaxdata:0x40000000   -c pbc_multi.f
mpxlf90  -qfixed=132 -O5 -qnoipa -bmaxdata:0x40000000   -c linkedcell.f
mpxlf90  -qfixed=132 -O5 -qnoipa -bmaxdata:0x40000000   -c self_d_corr.f
mpxlf90  -qfixed=132 -O5 -qnoipa -bmaxdata:0x40000000   -c transport_dmut.f
mpxlf90  -qfixed=132 -O5 -qnoipa -bmaxdata:0x40000000   -c onsagerL.f
mpxlf90  -qfixed=132 -O5 -qnoipa -bmaxdata:0x40000000   -c onsagerL_corr.f
mpxlf90  -qfixed=132 -O5 -qnoipa -bmaxdata:0x40000000   -c adriver.f
mpxlf90  -o mddriver -qfixed=132 -O5 -qnoipa -bmaxdata:0x40000000  md_mix_v17.o
transport_d.o md_mpi_extras.o pbc_multi.o linkedcell.o  self_d_corr.o transport_dmut.o
onsagerL.o onsagerL_corr.o adriver.o  /usr/apps/lib/libpblas.a -lessl
/usr/apps/lib/libblacsF77init.a  /usr/apps/lib/libblacsCinit.a  /usr/apps/lib/libblacs.a
```

II.A.3. How to execute codes

1. Move to the scratch directory

   If you are generating large data files, you have to run in the scratch directory. Your personal directory has a small quota. The eagle scratch directory is located at:

   /tmp/gpfs200a/dkeffer

   where I have created a subdirectory for my own jobs. You have to create your own subdirectory. Don't use mine.

2. Copy the executable, input and command file to the working directory

   Copy your executable file, any input files, and the eagle command file to your working subdirectory of the scratch directory. The executable file and input files are obvious. A sample command file, named, "md.cmd", is shown in Figure 2. The command file determines the files for standard output and standard error. It sets the maximum time. The maximum time per processor is twelve hours on eagle. You also set the number of nodes. Each node has four processors (as designated by tasks_per_node), so setting node=4, yields 16 processors. You must set your initial directory to the correct working subdirectory of the scratch directory, where your executable and input files are located. The command "poe mddriver", starts your executable in the parallel operating environment.

3. Submit the job
   llsubmit md.cmd

   This command submits your job to the LoadLeveler queue.

4. monitor job progress
   llq
   llqn –a

   These two commands return information about all jobs in the queue. For more information type man llq.

When the job is done, all your output files, standard output, and standard error are located in the working directory.

If you make a mistake and need to kill a job, use llcancel with the job id reported by llq.

II.A.4. Debugging

There are debugging tools on eagle. I don't use them. I use the old tried and true method of putting in a lot of print statements to find the line where the code crashed then fixing that line. See the eagle website for information on debugging tools.

I prefer to debug my codes on a small machine. If you want to debug on eagle, then you can submit an interactive job. An interactive job has a time limit of 2 hours and 5 minutes. It usually moves much more rapidly through the queue than a standard batch job. A sample command file for an interactive job is given in Figure 3. There is one line different. Submitting an interactive job is the same as for a standard batch job.

```
.SUFFIXES:  .f

COMP = mpxlf90

OPT = -qfixed=132 -O5 -qnoipa -bmaxdata:0x40000000

MAIN_LIB_DIR = /usr/local/lib
OTHER_MAIN_LIB_DIR = /usr/apps/lib

SCALAPACK_LIB_DIR = $(OTHER_MAIN_LIB_DIR)
BLACS_LIB_DIR = $(OTHER_MAIN_LIB_DIR)
PLBAS_LIB_DIR =  $(OTHER_MAIN_LIB_DIR)

LIB_SCALAPACK = $(SCALAPACK_LIB_DIR)/libscalapack.a \
             $(SCALAPACK_LIB_DIR)/libtools.a
LIB_BLACS = $(BLACS_LIB_DIR)/libblacsF77init.a \
          $(BLACS_LIB_DIR)/libblacsCinit.a \
          $(BLACS_LIB_DIR)/libblacs.a
LIB_PBLAS = $(PLBAS_LIB_DIR)/libpblas.a
LIB_BLAS = -lessl

LIB = $(LIBSCALAPACK) $(LIB_PBLAS) $(LIB_BLAS) $(LIB_BLACS)

TARGETS = mddriver

OBJS =  md_mix_v17.o transport_d.o md_mpi_extras.o pbc_multi.o linkedcell.o \
       self_d_corr.o transport_dmut.o onsagerL.o onsagerL_corr.o adriver.o

mddriver:       $(OBJS)
       $(COMP) -o mddriver $(OPT) $(OBJS) $(LIB)

clean:
       \rm -f *.o $(TARGETS)

all:    $(TARGETS)
```

Figure 1. Contents of makefile on eagle.ccs.ornl.gov

```
#@ job_type = parallel
#@ error = md.$(jobid).err
#@ output = md.$(jobid).scr
#@ wall_clock_limit = 12:00:00
#@ network.MPI = css0,shared,US
#@ tasks_per_node = 4
#@ node = 4
#@ node_usage = not_shared
#@ initialdir = /tmp/gpfs200a/dkeffer/work_eagle/task_015/000
#@ queue
pwd
echo $LOADL_PROCESSOR_LIST
cp md.in_000 md.in
poe mddriver
```

Figure 2. Sample command file, "md.cmd", for a batch job on eagle.ccs.ornl.gov

```
#@ job_type = parallel
#@ class = interactive
#@ error = md.$(jobid).err
#@ output = md.$(jobid).scr
#@ wall_clock_limit = 12:00:00
#@ network.MPI = css0,shared,US
#@ tasks_per_node = 4
#@ node = 4
#@ node_usage = not_shared
#@ initialdir = /tmp/gpfs200a/dkeffer/work_eagle/task_015/000
#@ queue
pwd
echo $LOADL_PROCESSOR_LIST
cp md.in_000 md.in
poe mddriver
```

Figure 3. Sample command file, "md.cmd", for an interactive job on eagle.ccs.ornl.gov

II.B. Falcon (falcon.ccs.ornl.gov)

machine: 64-node Compaq AlphaServer SC
relevant webpages: http://www.ccs.ornl.gov/Falcon.html

II.B.1. How to connect

for interactive window:
    ssh –l username falcon.ccs.ornl.gov
for file transfer:
    sftp falcon.ccs.ornl.gov

Once a month, you have to execute the command dfs_login, or falcon can't find any of your files.

II.B.2. How to compile and link

This step requires the same commands as on eagle. However, the makefile is different. The contents of the makefile on falcon are shown in Figure 4. Note, a code compiled on one machine won't run on another machine. You have to compile your code separately on eagle and falcon, if you want versions that run on eagle and falcon.

II.B.3. How to execute codes

1. Move to the scratch directory
    If you are generating large data files, you have to run in the scratch directory. Your personal directory has a small quota. The falcon scratch directory is located at:

    /cfs500a/dkeffer

    where I have created a subdirectory for my own jobs. This is a different scratch directory than eagle uses.

2. Copy the executable and input files to the working directory
    Copy your executable file and input files to your working subdirectory of the scratch directory. There is no command file.

3. Submit the job
    prun –n 16 mddriver > md_scr.out &

Here I have specified the number of processors, not nodes. There are still 4 processors per node, so pick a number that is divisible by 4. You have to redirect the standard output of the executable, mddriver, into a file, in this case md_scr.out, if you want to save any information that the code prints to the screen. The ampersand (&) runs this code in the background so that you can log out without killing the job. The time limit on falcon is 12 hours per node.

4. monitor job progress
   rinfo

II.B.4. Debugging

I don't know anything about debugging on falcon. Make sure the code works on another machine. Then run it on falcon.

```
.SUFFIXES:  .f

COMP = f90

OPT = -g -O5

MAIN_LIB_DIR = /usr/local/lib
SCALAPACK_LIB_DIR = $(MAIN_LIB_DIR)
BLACS_LIB_DIR = $(MAIN_LIB_DIR)
PLBAS_LIB_DIR =  $(MAIN_LIB_DIR)

LIB_SCALAPACK = $(SCALAPACK_LIB_DIR)/libscalapack.a \
            $(SCALAPACK_LIB_DIR)/libtools.a
LIB_BLACS = $(BLACS_LIB_DIR)/libblacsF77init.a \
          $(BLACS_LIB_DIR)/libblacsCinit.a \
          $(BLACS_LIB_DIR)/libblacs.a
LIB_PBLAS = $(PLBAS_LIB_DIR)/libpblas.a
LIB_BLAS = -lcxml
LIB_MPI = -lmpi -lelan -lelan3

LIB = $(LIB_SCALAPACK) $(LIB_PBLAS) $(LIB_BLAS) $(LIB_BLACS) $(LIB_MPI)

TARGETS = mddriver

OBJS =  md_mix_v17.o transport_d.o md_mpi_extras.o pbc_multi.o linkedcell.o \
      self_d_corr.o transport_dmut.o adriver.o onsagerL.o onsagerL_corr.o

mddriver:       $(OBJS)
      $(COMP) -o mddriver $(OPT) $(OBJS) $(LIB)

clean:
      \rm -f *.o $(TARGETS)

all:    $(TARGETS)

.f.o:
      $(COMP) $(OPT) $(INCCH)  -c $<
```

Figure 4. Contents of makefile on falcon.ccs.ornl.gov

II.C. Plato (plato.engr.utk.edu)

machine: 8-node Beowulf Cluster (2 Athlon AMD 1.6 GHz procs per node)
relevant webpages: none

II.C.1. How to connect

for interactive window:
    ssh –l username plato.engr.utk.edu
for file transfer:
    sftp plato.engr.utk.edu

II.C.2. How to compile and link

This step requires the same commands as on eagle. However, the makefile is different. The contents of the makefile on plato are shown in Figure 5. Note, a code compiled on one machine won't run on another machine. You have to compile your code separately on eagle and plato, if you want versions that run on eagle and plato.

Furthermore, plato uses Portland Group Fortran. This requires a license manager to be running. You must have the following line in the file .cshrc located in your home directory.

setenv LM_LICENSE_FILE /usr/pgi/license.dat

Once you add this file to .cshrc, log out and log back in. This only has to be done once.

II.C.3. How to execute codes

1. Define an .rhost file

In your home directory, e.g. /home/dkeffer, you must have a file named .rhosts which has the following content:

```
master      dkeffer
S01         dkeffer
S02         dkeffer
S03         dkeffer
S04         dkeffer
S05         dkeffer
S06         dkeffer
S07         dkeffer
```

In addition to this file, you can create files with subsets of this file. If you want to run a job only using the two processors of node S07, then you must create a file, called .rhosts7 with the following content:

```
S07         dkeffer
```

If you want to run a job using only the four processors of nodes S06 and S07, then you must create a file, called .rhosts67 with the following content:

```
S06        dkeffer
S07        dkeffer
```

Any combination of nodes can be used, but you must define a corresponding .rhosts file.

2. Configure LAMMPI

You must have the following line in the file .cshrc located in your home directory.

```
setenv LAMRSH "rsh"
```

Once you add this file to .cshrc, log out and log back in or source .cshrc. This only has to be done once.

3. Work in your home directory
   There is no scratch directory on plato.

4. Copy the executable and input files to the working directory
   Copy your executable file and input files to your working subdirectory. There is no command file.

5. rsh to one of the nodes that you intend to use.
   If you are going to run a job using the two processors of node 7, then you must rsh to node 7.

```
rsh S07
```

6. run the job

Execute the following sequence of commands.

```
lamboot -v ~dkeffer/.rhosts7
cd ~dkeffer/md_base/work/task_020
mpirun n0 C mddriver > md_scr.out_000 &
ps -u dkeffer
exit
```

The first line starts the MPI manager for your job using only node 7. Note this requires that you created the file .rhosts7. The second line changes to your working directory, where the executable and input files are located. The third line runs your program, mddriver, using (n0) the first node in the .rhosts7 list, using (C) all processors on that node, and redirecting (>) the output to the file md_scr.out_000. This job is run in the background, so that you can log out and

it will keep running. (&) The fourth line will confirm that your job started properly. The fifth line exits from your remote shell (rsh) to node S07.

When the job is finished, all files are located in the working directory on the master node.

4. monitor job progress
   rsh S07 ps –u dkeffer

II.C.4. Debugging

There are no special debugging tools on plato. As mentioned before, I use the plentiful print statement method.

II.C.5. Etiquette

There is no job manager on Plato. For performance reasons, each processor should only run one job at a time. Therefore, we have to think up some reasonable and efficient way of dividing processors up among jobs. This machine should not really be used for massive data production unless it is sitting idle. It is intended to provide an easy to use resource for developing parallel codes to be used on eagle, falcon, and other massively parallel machines. Since this machine was built mostly with Engineering Technology Fee funds, classroom education uses take priority over research.

```
.SUFFIXES:  .f

INCDIR = /usr/pgi/lammpi/include

OPT = -O3 -I$(INCDIR)

COMP_DIR = /usr/pgi/lammpi/bin/
COMP = $(COMP_DIR)mpif77

LIB_MPI_DIR = /usr/pgi/lammpi/lib/
LIB_MPI_01 = $(LIB_MPI_DIR)libmpi.a
LIB_MPI_02 = $(LIB_MPI_DIR)libpmpi.a
LIB_MPI_03 = $(LIB_MPI_DIR)liblammpio.a
LIB_MPI_04 = $(LIB_MPI_DIR)liblam.a
LIB_MPI_05 = $(LIB_MPI_DIR)liblamf77mpi.a

LIB5 =   $(LIB_MPI_01) $(LIB_MPI_02) $(LIB_MPI_03) $(LIB_MPI_04) $(LIB_MPI_05)

TARGETS = mddriver

OBJS =  md_mix_v16.o transport_d.o md_mpi_extras.o pbc_multi.o linkedcell.o \
      self_d_corr.o transport_dmut.o onsagerL.o onsagerL_corr.o adriver.o

mddriver:       $(OBJS)
      $(COMP) -o mddriver $(OPT) $(OBJS) $(LIB)

clean:
      \rm -f *.o $(TARGETS) *.pc *.pcl

all:    $(TARGETS)

.f.o:
      $(COMP) $(OPT) $(INCCH)  -c $<
```

Figure 5. Contents of makefile on plato.engr.utk.edu

III. TWO ALTERNATIVE PARALLELIZATION SCHEMES FOR MD SIMULATIONS

Studies have shown that in a molecular dynamics simulation using single-center Lennard-Jones molecules, approximately 90% of the computational effort is expended in the subroutine that evaluates the energy and forces.(Keffer 2002) We can visualize an NxN matrix, where N is our number of molecules in the simulation. The i,j^{th} element holds the potential energy due to the interaction of molecule i with molecule j. The diagonal elements of this matrix are 0, because molecule i does not interact with itself. The natural scheme is to recognize that the matrix is symmetric. We then compute only those interactions for i = 1 to N-1 and j = i+1 to N. See Figure 6. Analogous matrices can be constructed for the x-, y-, and z-components of the forces.

Taking advantage of the symmetry makes perfect sense on one processor. However, when we have multiple processors, we find that other considerations enter the equation.

III.A. The Symmetric And Balanced Neighbor List Method

The goal of this method is to take advantage of the symmetry of the energy matrix and at the same time have each processor perform an equal amount of work. (This is called running a balanced job.) If we naively divide the atoms among processors and compute the interactions for i and j = i+1 to N, we end up with an unbalanced code. See Figure 7. In Figure 7, we divide the atoms among four processors. When we look at this schematic of the force loop, clearly processor 0 has more work than any other node. This is not a balanced code and will give poor parallel performance.

We have fixed this problem using a scheme illustrated in Figure 8. In Figure 8, each processor has the same number of i-j interactions to compute. Moreover, the calculation is constructed in such a way that only the independent pairwise interactions are computed. We have taken advantage of the symmetry and have a balanced code.

This schematic design is implemented in creating the neighbor list. The neighbor list for each molecule is then fed into energy and force calculation.

We can characterize this method as minimizing computational effort at the expense of communication.

III.B. The Double Work Method

The double work method is very simple. Ignore the symmetry. Calculate i-j and j-i interactions independently. This is shown in Figure 9. Here the code is perfectly balanced; each processor must compute the same number of interactions.

We can characterize this method as minimizing interprocessor communication at the expense of computational effort.

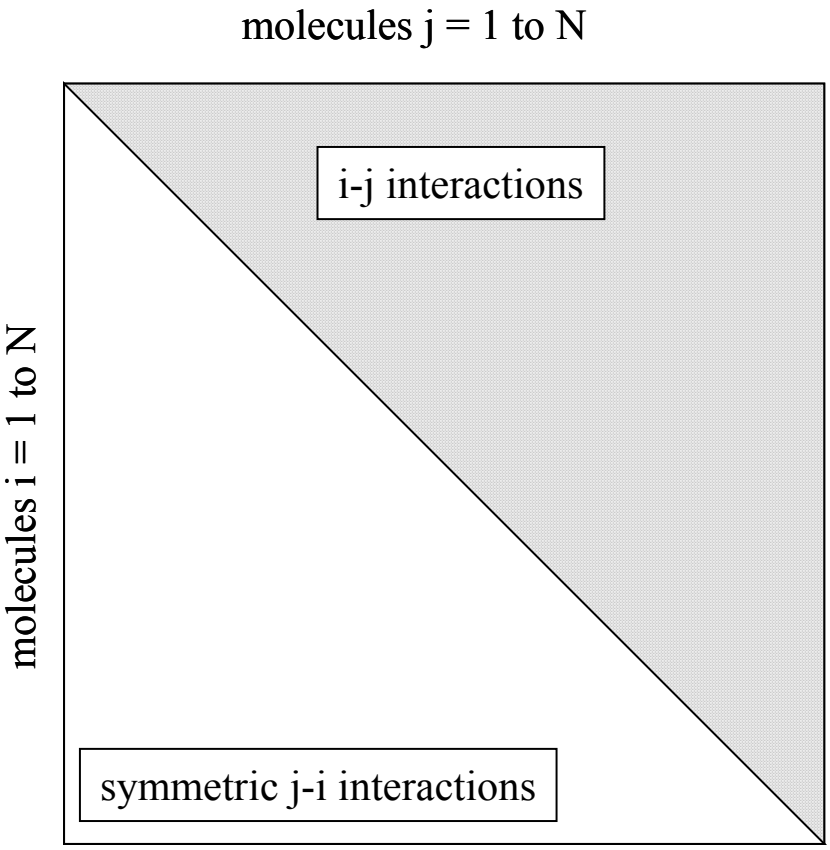

Figure 6. Schematic of Symmetric Matrix

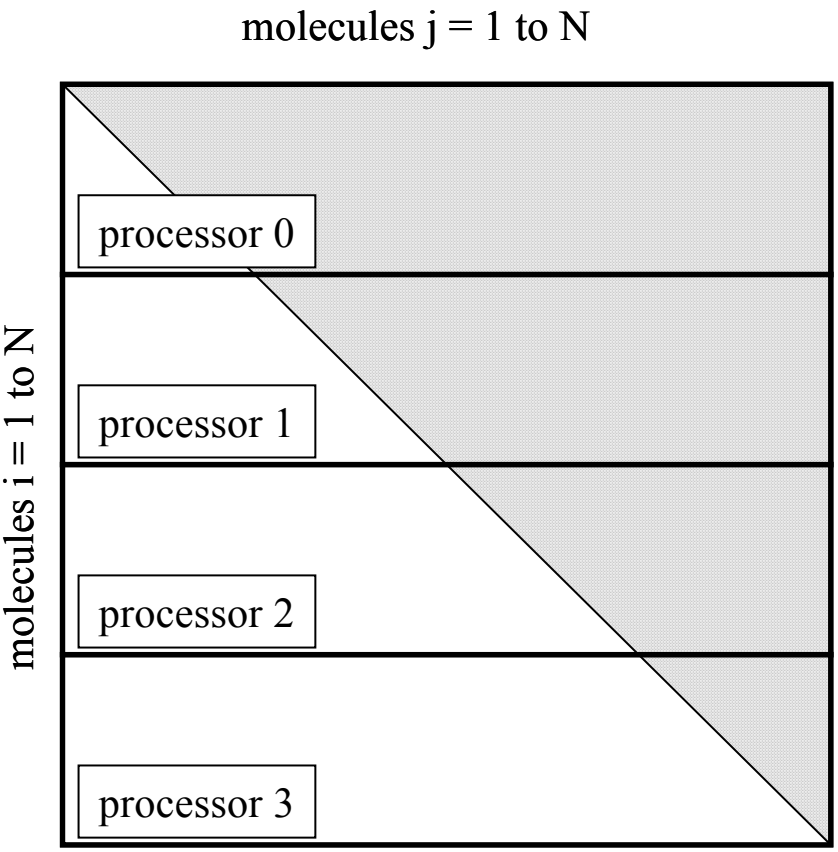

Figure 7. Schematic of Symmetric Matrix split naively among processors

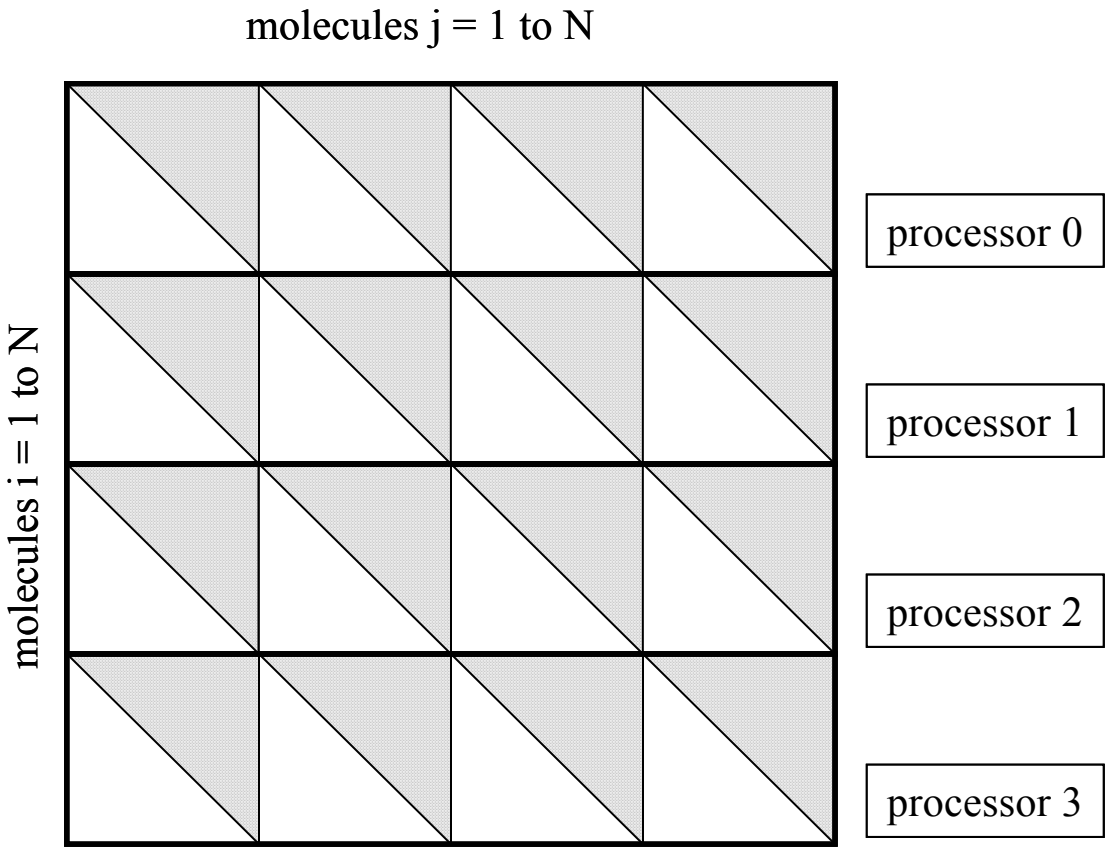

Figure 8. Schematic of Symmetric Matrix split to balance processor load.

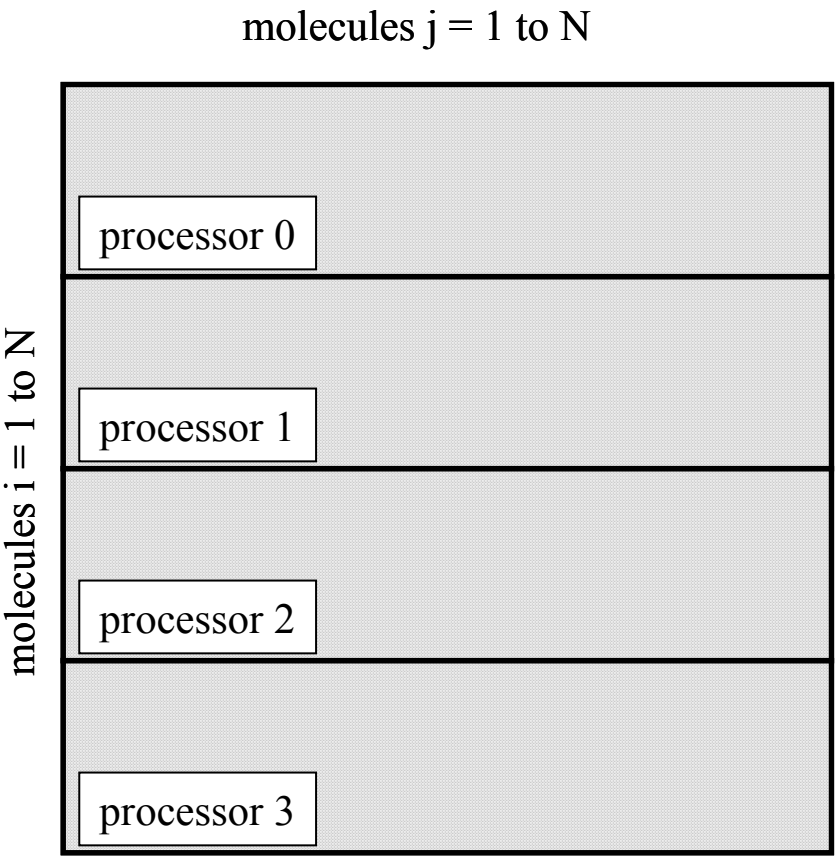

Figure 9. Schematic of Double Work Matrix split to balance processor load.

III.C. Case Studies

The purpose of this case study is to determine the parallel efficiency of the symmetric and balanced neighbor list method and the double work method on different machines, as a function of system size.

Below we provide all information provided to the code that could affect computation time:

simulation: isobaric-isothermal
potential: Lennard-Jones 12-6
chemical identity: methane
Number of molecules N=1,000 or 10,000
temperature = 300 K
pressure = 1 atm
equilibration steps: 10000
production steps: 10000
time step = 2 fs
cut-off distance = 15 Å
neighbor distance = 18 Å
interval for updating neighbor list = 40
interval for sampling thermodynamic properties = 1
interval for saving mean square displacements = 100000

In the following case studies, the numbers are only a rough estimate, because especially on eagle, there can be fluctuations in the running time due to events outside the enduser's knowledge and control.

III.C.1. dirichlet.engr.utk.edu

operating system: Red Hat Linux 7.2
Fortran: Intel Fortran 90 (free Fortran compiler for Linux)
MPI: MPICH (Argonne National Labs, public domain code)
processors: Athlon AMD 1.6 GHz

N =1,000 molecules				
processors	method	CPU time total (sec)	CPU time per processor	total CPU time ratio
1	symmetric and balanced nbr list	105	105	1
2	symmetric and balanced nbr list	161	80	1.44
1	double work	178	178	1.69
2	double work	207	104	1.97

N =10,000 molecules				
processors	method	CPU time total (sec)	CPU time per processor	total CPU time ratio
1	symmetric and balanced nbr list	2898	2898	1
2	symmetric and balanced nbr list	3627	1813	1.25
1	double work	4646	4646	1.60
2	double work	5021	2511	1.73

Conclusions for Dirichlet:

This system has fast communication. The two processors are communicating across the motherboard. There are no cables involved. As a result, for both 1,000 and 10,000 molecules, the symmetric and balanced nbr list (SBNL) method is a more computationally efficient method than the double work (DW) method. While it is true that the communication penalty in moving from one to two nodes is greater for the SBNL method, it does not outweigh the factor of two penalty intrinsic in the DW method.

III.C.2. plato.engr.utk.edu

operating system: Red Hat Linux 7.3
Fortran: Intel Fortran 90
MPI: LAM-MPI (University of Notre Dame, default MPI with Red Hat Linux)
processors: Athlon AMD 1.9 GHz

N =1,000 molecules				
processors	method	CPU time total (sec)	CPU time per processor	total CPU time ratio
1	symmetric and balanced nbr list	126	126	1
2	symmetric and balanced nbr list	179	89	1.42
1	double work	224	224	1.78
2	double work	250	125	1.98

N =10,000 molecules				
processors	method	CPU time total (sec)	CPU time per processor	total CPU time ratio
1	symmetric and balanced nbr list	3528 (3495)	3528 (3495)	1 (0.99)
2	symmetric and balanced nbr list	4330 (4353)	2165 (2176)	1.23 (1.23)
4	symmetric and balanced nbr list	14608	3652	4.14
8	symmetric and balanced nbr list	51227	6403	14.52
1	double work	6678	6678	1.89
2	double work	6984	3491	1.98
4	double work	9439	2359	2.68
8	double work	30490	3811	8.64

Conclusions for plato:

When we run with only one or two processors, the communication is fast (across a motherboard). Once we use more than two processors, we have communication over ethernet, which is much slower. Therefore, when we use two or less processors the advantage lies with the SBNL method. When we use more than two processors, the advantage lies with the DW method, which minimizes communication.

We also see that we take a huge hit when communicating across nodes (via ethernet). Therefore, the optimal mode of simulation on plato is to run several simulations simultaneously, each using two nodes.

III.C.3. eagle.ccs.ornl.gov

operating system: AIX 4.3.3.2 (an IBM type of Unix)
Fortran: IBM's XL Fortran (Fortran 95)
MPI: unknown
processors: IBM RS/6000 SP with four 375 MHz Power3-II processors

N =1,000 molecules				
nodes (processors)	method	CPU time total (sec)	CPU time per processor	total CPU time ratio
1 (4)	symmetric and balanced nbr list	211	53	1
2 (8)	symmetric and balanced nbr list	680	85	3.22
1 (4)	double work	240	60	1.14
2 (8)	double work	469	59	2.22

N =10,000 molecules				
nodes (processors)	method	CPU time total (sec)	CPU time per processor	total CPU time ratio
1 (4)	symmetric and balanced nbr list	2974	744	1
2 (8)	symmetric and balanced nbr list	4964	621	1.67
1 (4)	double work	4489	1122	1.51
2 (8)	double work	5387	673	1.81

Conclusions for Eagle:

It's hard to believe this data even though I generated it myself. The trends are inconsistent with respect to method, number of processors, and system size. Clearly, there is a lot of noise in data, meaning that eagle sometimes can run a job quickly and sometimes slowly, depending on unknown variables. I would have to run these jobs numerous times to obtain a statistical average of the CPU usage, or the data is real.

In the past, when I have performed efficiency tests on eagle similar to this one, I convinced myself that the double work model was best. Here we would be forced to draw the opposite conclusion. I find this highly dubious. Before I embraced the SBNL method, I would reconfirm this and I would make sure that this trend holds for more nodes.

III.C.4. falcon.ccs.ornl.gov

operating system: Tru64 version 5.1 (a Compaq type of Unix)
Fortran: Compaq Fortran
MPI: unknown
processors: Each node has four 667 MHz Alpha EV67 processors

N =1,000 molecules				
nodes (processors)	method	CPU time total (sec)	CPU time per processor	total CPU time ratio
1 (4)	symmetric and balanced nbr list	185	46	1
2 (8)	symmetric and balanced nbr list	371	46	2.01
1 (4)	double work	253	63	1.37
2 (8)	double work	358	44	1.94

N =10,000 molecules				
nodes (processors)	method	CPU time total (sec)	CPU time per processor	total CPU time ratio
1 (4)	symmetric and balanced nbr list	3896	974	1
2 (8)	symmetric and balanced nbr list	4991	624	1.28
1 (4)	double work	6846	1711	1.76
2 (8)	double work	7364	921	1.89

Conclusions for Falcon:

For small systems, the SBNL and DW methods are roughly the same in terms of computational efficiency. For large systems, the SBNL method is best. I would verify that this trend continues for larger number of nodes.

III.C.5. falcon.ccs.ornl.gov(statistically significant test)

MD Simulation
steps: 1.2 million
molecules; 10,000
4 nodes (16 processors)
11 runs
Maximum allowable run-time: 12 hour/proc
Maximum total run time for 16 procs: 691200

DW METHOD: All 11 programs terminated normally

md_scr.out_000: Program has used 632543.376568600 seconds of CPU time.
md_scr.out_001: Program has used 637204.925352000 seconds of CPU time.
md_scr.out_002: Program has used 598534.869334800 seconds of CPU time.
md_scr.out_003: Program has used 573450.151012600 seconds of CPU time.
md_scr.out_004: Program has used 566060.170955400 seconds of CPU time.
md_scr.out_005: Program has used 605724.629120200 seconds of CPU time.
md_scr.out_006: Program has used 597739.679417000 seconds of CPU time.
md_scr.out_007: Program has used 617203.354518000 seconds of CPU time.
md_scr.out_008: Program has used 582136.182068000 seconds of CPU time.
md_scr.out_009: Program has used 600283.754023800 seconds of CPU time.
md_scr.out_010: Program has used 610876.779116200 seconds of CPU time.

average: 601978 sec
standard deviation: 22428 sec (or 3.7% of average)

SBNL METHOD: Only 7 of 11 programs terminated before maximum run time reached

000/md_scr.out_000: Program has used 553003.578095000 seconds of CPU time.
001/md_scr.out_001: Program has used 544714.794630600 seconds of CPU time.
004/md_scr.out_004: Program has used 510505.479113200 seconds of CPU time.
005/md_scr.out_005: Program has used 540618.279511600 seconds of CPU time.
007/md_scr.out_007: Program has used 537163.579330400 seconds of CPU time.
008/md_scr.out_008: Program has used 521297.539395000 seconds of CPU time.
010/md_scr.out_010: Program has used 546039.489724200 seconds of CPU time.

average: 536192 sec
standard deviation: 15025 sec (or 2.8% of average)

Possible Explanation: There is fast communication between adjacent nodes and slow communication between distant nodes. The job manager does not always assign adjacent nodes. If we get adjacent nodes, the SBNL method is faster. If we don't get adjacent nodes, the SBNL method cannot even finish in the allotted time constratints. The SBNL method relies more on communication and is thus more susceptible to poor node distribution.

IV. NOTES ON THE CODES

IV.A. md_mix_v16.f

Capabilities:

1. This is a molecular dynamics code for multicomponent single-center Lennard-Jones materials written in FORTRAN 90 and using subroutines from the MPI library.
2. All inputs are read from the input file, md.in.
3. It can run microcanonical, canonical, or isobaric-isothermal simulations.
4. The canonical simulations use the Melchionna formulation of the Nose-Hoover thermostat.(Melchionna, Ciccotti et al. 1993)
5. The isobaric-isothermal simulations use the Melchionna barostat.(Melchionna, Ciccotti et al. 1993)
6. The equilibration stage is canonical via velocity-scaling.
7. The integrator is the Gear 5^{th} order predictor corrector method.(Gear 1966; Gear 1971)
8. While in microcanonical mode, this code has been tested to conserve energy and momentum. The momentum conservation is to 12 significant figures. The energy conservation of course depends on the size of time step.
9. The potential is a truncated Lennard-Jones potential. The long-range correction is added to the energy.
10. This code uses a neighbor list.
11. **For parallelization purposes, this code uses the symmetric and balanced neighbor list method.**
12. This code saves a mean-square displacement file for calculation of self-diffusivities via the Einstein relation and for calculation of Onsager phenomenological coefficients via linear response theory.
13. This code has compiled and executed without error on the following systems:
    a) 2-processor Linux workstation with Intel Fortran and MPICH
    b) 16-processor Linux cluster with PGI Fortran and LAMMPI
    c) 16-procs of eagle with IBM FORTRAN and unknown MPI
    d) 16-procs of falcon with Compaq FORTRAN and unknown MPI

Short-comings:

1. The symmetric and balanced neighbor list method requires that the number of molecules be an integer multiple of the number of molecules. Each processor must have the same number of molecules.
2. This code has a good deal of communication in the force calculation.

IV.B. md_mix_v17.f

Capabilities:

1. This is a molecular dynamics code for multicomponent single-center Lennard-Jones materials written in FORTRAN 90 and using subroutines from the MPI library.
2. All inputs are read from the input file, md.in.
3. It can run microcanonical, canonical, or isobaric-isothermal simulations.
4. The canonical simulations use the Melchionna formulation of the Nose-Hoover thermostat.(Melchionna, Ciccotti et al. 1993)
5. The isobaric-isothermal simulations use the Melchionna barostat.(Melchionna, Ciccotti et al. 1993)
6. The equilibration stage is canonical via velocity-scaling.
7. The integrator is the Gear 5^{th} order predictor corrector method.(Gear 1966; Gear 1971)
8. While in microcanonical mode, this code has been tested to conserve energy and momentum. The momentum conservation is to 12 significant figures. The energy conservation of course depends on the size of time step.
9. The potential is a truncated Lennard-Jones potential. The long-range correction is added to the energy.
10. This code uses a neighbor list.
11. **For parallelization purposes, this code uses the double work method.**
12. This code saves a mean-square displacement file for calculation of self-diffusivities via the Einstein relation and for calculation of Onsager phenomenological coefficients via linear response theory.
13. This code has compiled and executed without error on the following systems:
    a) 2-processor Linux workstation with Intel Fortran and MPICH
    b) 16-processor Beowulf cluster with PGI Fortran and LAMMPI
    c) 16-procs of eagle with IBM FORTRAN and unknown MPI
    d) 16-procs of falcon with Compaq FORTRAN and unknown MPI

Short-comings:

1. This code does not take advantage of the symmetry in the energy and force matrices.

References:

Gear, C. W. (1966). The Numerical Integration of Ordinary Differential Equations of Various Orders, Argonne National Laboratory.

Gear, C. W. (1971). Numerical Initial Value Problems in Ordinary Differential Equations. Englewood Cliffs, New Jersey, Prentice Hall, Inc.

Keffer, D. (2002). "A Working Person's Guide to Molecular Dynamics Simulations." http://clausius.engr.utk.edu/che548/pdf/md_sim.pdf.

Melchionna, S., G. Ciccotti, et al. (1993). "Hoover NPT dynamics for systems varying in size and shape." Mol. Phys. **78**(3): 533-544.