Introduction to Parallel Programming with MPI

PICASso Tutorial
October 22-23, 2007

Stéphane Ethier
(ethier@pppl.gov)
Computational Plasma Physics Group
Princeton Plasma Physics Lab
Why Parallel Computing?

• Want to speed up a calculation.
• Solution:
  – Split the work between several processors.
• How?
  – It depends on the type of parallel computer
    • Shared memory (usually thread-based)
    • Distributed memory (process-based)
  – MPI works on all of them!
• “Classic” shared memory node
• Renewed interest due to multi-core chips
• Processors communicate through memory
Shared memory parallelism

• Program runs inside a single process
• Several “execution threads” are created within that process and work is split between them.
• The threads run on different processors.
• All threads have access to the shared data through shared memory access.
• Must be careful not the have threads overwrite each other’s data.
Shared memory programming

• Easy to do loop-level parallelism.
• Compiler-based automatic parallelization
  – Easy but not effective
  – Better to do it yourself with OpenMP
• Coarse-grain parallelism can be difficult
• Becoming important for multi-core processors due to very low latency compared to conventional SMP nodes
Distributed memory parallelism

- The processor can only access its local memory
- Data exchange between processors must be explicit
Distributed memory parallelism

- Process-based programming.
- Each process has its own memory space that cannot be accessed by the other processes.
- The work is split between several processes.
- For efficiency, each processor runs a single process.
- Communication between the processes must be explicit, e.g. Message Passing
How to split the work between processors?

• Most widely used method for grid-based calculations:
  – **DOMAIN DECOMPOSITION**

• Split particles in particle-in-cell (PIC) or molecular dynamics codes.

• Split arrays in PDE solvers

• etc…

• Keep it **LOCAL**
What is MPI?

• MPI stands for Message Passing Interface.
• It is a message-passing specification, a standard, for the vendors to implement.
• In practice, MPI is a set of functions (C) and subroutines (Fortran) used for exchanging data between processes.
• An MPI library exists on most, if not all, parallel computing platforms so it is highly portable.
How much do I need to know?

• MPI is small (6 functions)
  – Many parallel programs can be written with just 6 basic functions.

• MPI is large (125 functions)
  – MPI's extensive functionality requires many functions
  – Number of functions not necessarily a measure of complexity

• MPI is just right
  – One can access flexibility when it is required.
  – One need not master all parts of MPI to use it.
How MPI works

• Launch the parallel calculation with:
  mpirun –np #proc a.out
  mpiexec –n #proc a.out

• Copies of the same program run on each processor within its own process (private address space).
• Each processor works on a subset of the problem.
• Exchange data when needed
  – Can be exchanged through the network interconnect
  – Or through the shared memory on SMP machines (Bus?)
• Easy to do coarse grain parallelism = scalable
Good MPI web sites

- http://www.llnl.gov/computing/tutorials/mpi/
- http://www.nersc.gov/nusers/help/tutorials/mpi/intro/

- MPI on Linux clusters:
  - MPICH (http://www-unix.mcs.anl.gov/mpi/mpich/)
  - Open MPI (http://www.open-mpi.org/)
Structure of an MPI program

Program mpi_code
    ! Load MPI definitions
    use mpi (or include mpif.h)

    ! Initialize MPI
    call MPI_Init(ierr)

    ! Get the number of processes
    call MPI_Comm_size(MPI_COMM_WORLD,nproc,ierr)

    ! Get my process number (rank)
    call MPI_Comm_rank(MPI_COMM_WORLD,myrank,ierr)

    Do work and make message passing calls...

    ! Finalize
    call MPI_Finalize(ierr)

end program mpi_code
Structure of an MPI program

```
#include "mpi.h"
int main( int argc, char *argv[] )
{
    int nproc, myrank;
    /* Initialize MPI */
        MPI_Init(&argc,&argv);
    /* Get the number of processes */
        MPI_Comm_size(MPI_COMM_WORLD,&nproc);
    /* Get my process number (rank) */
        MPI_Comm_rank(MPI_COMM_WORLD,&myrank);

        Do work and make message passing calls...

    /* Finalize */
        call MPI_Finalize();
    return 0;
}
```
Compilation

• mpich provides scripts that take care of the include directories and linking libraries
  – mpiccc
  – mpiCC
  – mpif77
  – mpif90

• Otherwise, must link with the right MPI library
Makefile

• Always a good idea to have a Makefile

```bash
%cat Makefile
CC=mpiccc
CFLAGS=-O

% : %.c
    $(CC) $(CFLAGS) $< -o $@
```
mpirun and mpiexec

• Both are used for starting an MPI job
• If you don’t have a batch system, use **mpirun**

```bash
mpirun -np #proc -machinefile mfile a.out >& out < in &
```

%cat mfile
```
machine1.princeton.edu
machine2.princeton.edu
machine3.princeton.edu
machine4.princeton.edu
```

• PBS usually takes care of arguments to mpiexec
Batch System: PBS primer

- Submit a job script: `qsub script`
- Check status of jobs: `qstat -a` (for all jobs)
- Stop a job: `qdel job_id`

```bash
### --- PBS SCRIPT ---
#PBS -l nodes=4:ppn=2,walltime=02:00:00
#PBS -q dque
#PBS -V
#PBS -N job_name
#PBS -m abe
cd $PBS_O_WORKDIR
mpiexec -np 8 a.out
```
Basic MPI calls to exchange data

**Point to point:** 2 processes at a time

\[
\begin{align*}
\text{MPI\_Send} & (\text{buf, count, datatype, dest, tag, comm, ierr}) \\
\text{MPI\_Recv} & (\text{buf, count, datatype, source, tag, comm, status, ierr})
\end{align*}
\]

\[
\text{MPI\_Sendrecv} (\text{sendbuf, sendcount, sendtype, dest, sendtag, recvbuf, recvcount, recvtype, source, recvtag, comm, status, ierr})
\]

where types are: MPI\_INTEGER, MPI\_REAL, MPI\_DOUBLE\_PRECISION, MPI\_COMPLEX, MPI\_CHARACTER, MPI\_LOGICAL, etc...

Predefined Communicator: MPI\_COMM\_WORLD
Collective MPI calls

Collective calls: All processes participate

One process sends to everybody:
MPI_Bcast(buffer,count,datatype,root,comm,ierr)

All processes send to “root” process and the operation “op” is applied
MPI_Reduce(sendbuf,recvbuf,count,datatype,op,root,comm,ierr)

where \( op = \text{MPI\_SUM, MPI\_MAX, MPI\_MIN, MPI\_PROD, etc…} \)

You can create your own reduction operation with \text{MPI\_Op\_create()}

All processes send to everybody and apply the operation “op” (equivalent to an
MPI_Reduce followed by an MPI_Bcast)
MPI_Allreduce(sendbuf,recvbuf,count,datatype,op,comm,ierr)

Synchronize all processes
MPI_Barrier(comm,ierr)
More MPI collective calls

All processes send a different piece of data to one single “root” process which gathers everything (messages ordered by index)

\[
\text{MPI\textunderscore\text{Gather}(sendbuf, sendcnt, sendtype, recvbuf, recvcount, recvtype, root, comm, ierr)}
\]

All processes gather everybody else’s pieces of data

\[
\text{MPI\textunderscore\text{Allgather}(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, comm, info)}
\]

One “root” process send a different piece of the data to each one of the other processes

\[
\text{MPI\textunderscore\text{Scatter}(sendbuf, sendcnt, sendtype, recvbuf, recvcount, recvtype, root, comm, ierr)}
\]

Each process performs a scatter operation, sending a distinct message to all the processes in the group in order by index.

\[
\text{MPI\textunderscore\text{Alltoall}(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, comm, ierr)}
\]