Introduction to MPI Programming

Rocks-A-Palooza II Lab Session

Modes of Parallel Computing

- SIMD - Single Instruction Multiple Data

  processors are “lock-stepped”: each processor executes single instruction in synchronism on different data

- SPMD - Single Program Multiple Data

  processors run asynchronously a personal copy of a program

- MIMD - Multiple Instruction Multiple Data

  processors run asynchronously: each processor has its own data and its own instructions

  - MPMD - Multiple Program Multiple Data

MPI in Parallel Computing

- MPI addresses message-passing mode of parallel computation
  - Processes have separate address spaces
  - Processes communicate via sending and receiving messages
- MPI is designed mainly for SPMD/MIMD (or distributed memory parallel supercomputer)
  - Each process is run on a separate node
  - Communication is over high-performance switch
  - Paragon, IBM SP2, Meiko CS-2, Thinking Machines CM-5, NCube-2, and Cray T3D
- MPI can support shared memory programming model
  - Multiple processes can read/write to the same memory location
  - SGI Onyx, Challenge, Power Challenge, Power Challenge Array, IBM SMP, Convex Exemplar, and the Sequent Symmetry
- MPI exploits Network Of Workstations (heterogeneous)
  - Sun, DEC, Hewlett-Packard, SGI, IBM, Intel and Pentium (various Linux OS)

What is MPI?

- Message Passing application programmer Interface
  - Designed to provide access to parallel hardware
    - Clusters
    - Heterogeneous networks
    - Parallel computers
  - Provides for development of parallel libraries
  - Message passing
    - Point-to-point message passing operations
    - Collective (global) operations
  - Additional services
    - Environmental inquiry
    - Basic timing info for measuring application performance
    - Profiling interface for external performance monitoring

MPI advantages

- Mature and well understood
  - Backed by widely-supported formal standard (1992)
  - Porting is “easy”
- Efficiently matches the hardware
  - Vendor and public implementations available
- User interface:
  - Efficient and simple (vs. PVM)
  - Buffer handling
  - Allow high-level abstractions
- Performance

MPI disadvantages

- MPI 2.0 includes many features beyond message passing

- Execution control environment depends on implementation

MPI features

- Thread safety
- Point-to-point communication
  - Modes of communication

    standard | synchronous | ready | buffered

  - Structured buffers
  - Derived datatypes
- Collective communication
  - Native built-in and user-defined collective operations
  - Data movement routines
- Profiling
  - Users can intercept MPI calls and call their own tools

Communication modes

- standard
  - *send* has no guarantee that corresponding *receive* routine has started
- synchronous
  - *send* and *receive* can start before each other but complete together
- ready
  - used for accessing fast protocols
  - user guarantees that matching *receive* was posted
  - use with care!
- buffered
  - *send* may start and return before matching *receive*
  - buffer space must be provided

Communication modes (cont'd)

- All routines are
  - Blocking - return when they are locally complete
    - Send does not complete until buffer is empty
    - Receive does not complete until buffer is full
    - Completion depends on
      - size of message
      - amount of system buffering
  - Non-blocking - returns immediately and allows next statement to execute
    - Use to overlap communication and computation when time to send data between processes is large
    - Immediately returns "request handle" that can be used for querying and waited on,
    - Completion detected by MPI_Wait() or MPI_Test()

Point-to-point vs. collective

- point-to-point, blocking MPI_Send/MPI_Recv

  MPI_Send(start, count, datatype, dest, tag, comm)

  MPI_Recv(start, count, datatype, source, tag, comm, status)
  - simple but inefficient
  - most work is done by process 0:
    - Get data and send it to other processes (they idle)
    - May be compute
    - Collect output from the processes
- collective operations to/from all

  MPI_Bcast(start, count, datatype, root, comm)

  MPI_Reduce(start, result, count, datatype, operation, root, comm)
  - called by all processes
  - simple, compact, more efficient
  - must have the same size for “count”and “datatype”
  - “result” has significance only on node 0

MPI complexity

- MPI extensive functionality is provided by many (125+) functions
- Do I Need them all ?
  - No need to learn them all to use MPI
  - Can use just 6 basic functions

    MPI_Init
    MPI_Comm_size
    MPI_Comm_rank
    MPI_Send / **MPI_Recv** or **MPI_Bcast** / **MPI_Reduce**
    MPI_Finalize
  - Flexibility: use more functions as required

To be or not to be MPI user

- Use if:
  - Your data do not fit data parallel model
  - Need portable parallel program
  - Writing parallel library
- Don't use if:
  - Don't need any parallelism
  - Can use libraries
  - Can use fortran

Writing MPI programs

- provide basic MPI definitions and types

  #include “mpi.h”
- start MPI

  MPI_Init(&argc, &argv);
- provide local non-MPI routines
- exit MPI

  MPI_Finalize();

see /opt/mpich/gnu/examples
/opt/mpich/gnu/share/examples

Compiling MPI programs

- From a command line:
  - mpicc -o prog prog.c
- Use profiling options (specific to mpich)
  - **-mpilog** Generate log files of MPI calls
  - **-mpitrace** Trace execution of MPI calls
  - **-mpianim** Real-time animation of MPI (not available on all systems)
  - **--help** Find list of available options
- Use makefile!
  - get Makefile.in template and create Makefile

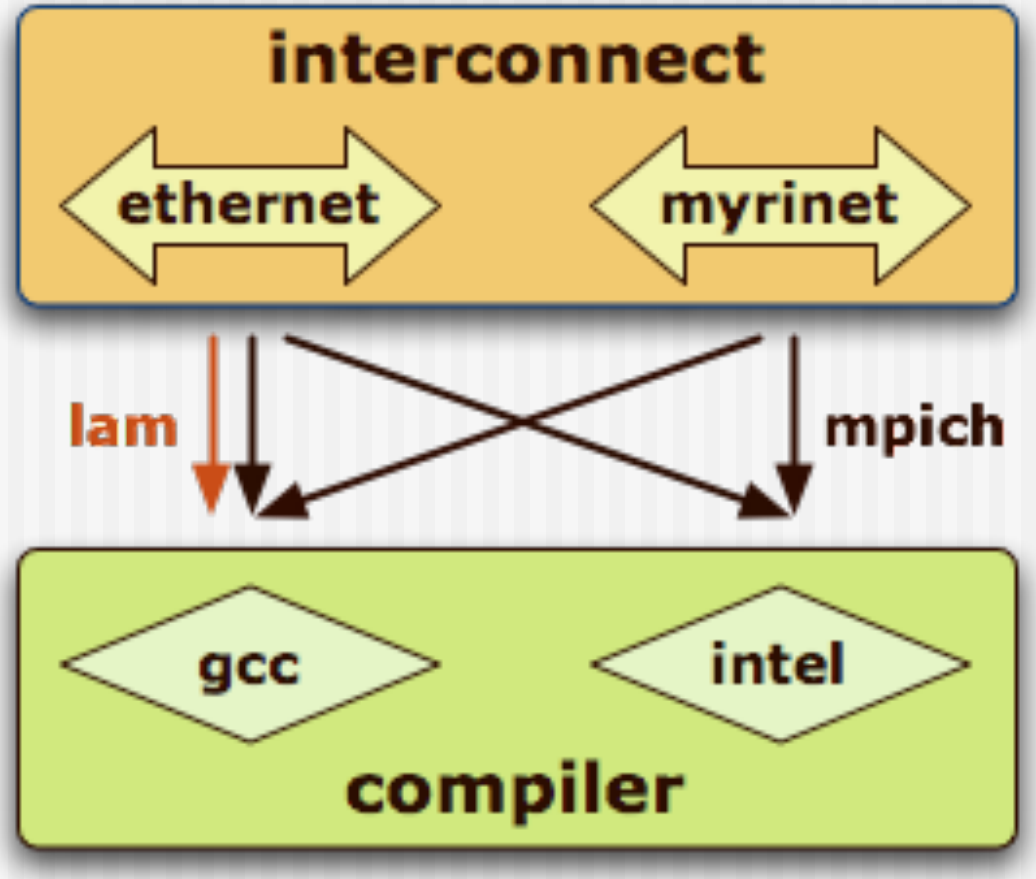
    mpireconfig Makefile
  - compile

    make progName

Running MPI program

- Depends on your implementation of MPI
  - For mpich:
    - mpirun -np2 foo # run MPI program
  - For lam:
    - lamboot -v lamhosts # starts LAM
    - mpirun -v -np 2 foo # run MPI program
    - lamclean -v # rm all user processes
    - mpirun … # run another program
    - lamclean …
    - lamhalt # stop LAM

Common MPI flavors on Rocks



MPI flavors path

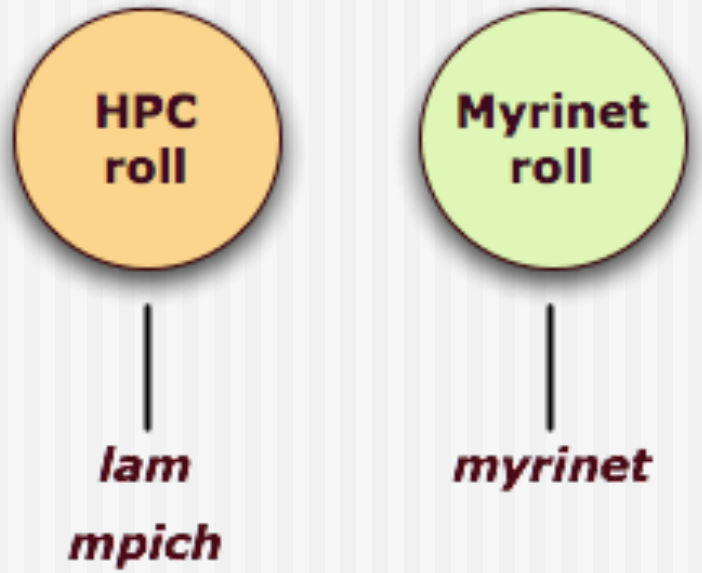
/opt + MPI flavor + interconnect + compiler + bin/ + executable

- MPICH + Ethernet + GNU
  /opt/mpich/ethernet/gnu/bin/…
- MPICH + Myrinet + GNU
  /opt/mpich/myrinet/gnu/bin/…
- MPICH + Ethernet + INTEL
  /opt/mpich/ethernet/intel/bin/…
- MPICH + Myrinet + INTEL
  /opt/mpich/myrinet/intel/bin/…

- LAM + Ethernet + GNU
  /opt/lam/ethernet/gnu/bin/…
- LAM + Myrinet + GNU
  /opt/lam/myrinet/gnu/bin/…
- LAM + Ethernet + INTEL
  /opt/lam/ethernet/intel/bin/…
- LAM + Myrinet + INTEL
  /opt/lam/myrinet/intel/bin/…

C: mpicc **C++: mpiCC**
F77: mpif77 **F90: mpif90**

What provides MPI



Example 1: LAM hello

Execute all commands as a regular user

1. Start ssh agent for key management
   $ ssh-agent $SHELL
2. Add your keys
   $ ssh-add
   (at prompt give your ssh passphrase)
3. Make sure you have right mpicc:
   $ which mpicc
   (output must be /opt/lam/gnu/bin/mpicc)
4. Create program source hello.c (see next page)

hello.c

```
#include "mpi.h"
#include <stdio.h>
int main(int argc ,char *argv[])
{
    int myrank;
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
    fprintf(stdout, "Hello World, I am process  %d\n", myrank);
    MPI_Finalize();
    return 0;
}
```

Example 1 (cont'd)

5. compile

   $ mpicc -o hello hello.c

6. create machines file with IP's of two nodes. Use your numbers here!

   198.202.156.1

   198.202.156.2

7. start LAM

   $ lamboot -v machines

8. run your program

   $ mpirun -np 2 -v hello

9. clean after the run

   $ lamclean -v

10. stop LAM

    $ lamhalt

Example1 output

$ ssh-agent $SHELL	
$ ssh-add	Enter passphrase for /home/nadya/.ssh/id_rsa: Identity added: /home/nadya/.ssh/id_rsa (/home/nadya/.ssh/id_rsa)
$ which mpicc	/opt/lam/gnu/bin/mpicc
$ mpicc -o hello hello.c	
$ lamboot -v machines	LAM 7.1.1/MPI 2 C++/ROMIO - Indiana University n-1<27213> ssi:boot:base:linear: booting n0 (rocks-155.sdsc.edu) n-1<27213> ssi:boot:base:linear: booting n1 (10.255.255.254) n-1<27213> ssi:boot:base:linear: finished
$ mpirun -np 2 -v hello	27245 hello running on n0 (o) 7791 hello running on n1 Hello World, I am process 0 Hello World, I am process 1
$ lamclean -v	killing processes, done closing files, done sweeping traces, done cleaning up registered objects, done sweeping messages, done
$ lamhalt	LAM 7.1.1/MPI 2 C++/ROMIO - Indiana University

Example 2: mpich cpi

1. set your ssh keys as in example 1 (if not done already)
   $ ssh-agent $SHELL
   $ ssh-add
2. copy example files to your working directory
   $ cp /opt/mpich/gnu/examples/*.c .
   $ cp /opt/mpich/gnu/examples/Makefile.in .
3. create Makefile
   $ mpireconfig Makefile
4. make sure you have right mpicc
   $ which mpicc
   If output lists path /opt/lam… update the path:
   $ export PATH=$/opt/mpich/gnu/bin:$PATH
5. compile your program
   $ make cpi
6. run
   $ mpirun -np 2 -machinefile machines cpi

or $ mpirun -nolocal -np 2 -machinefile machines cpi

Example 2 details

- If using frontend and compute nodes in machines file use

  mpirun -np 2 -machinefile machines cpi
- If using only compute nodes in machine file use

  mpirun -nolocal -np 2 -machinefile machines cpi

  - -nolocal - don't start job on frontend
  - -np 2 - start job on 2 nodes
  - -machinefile machines - nodes are specified in machinesfile
  - cpi - start program cpi

More examples

- See CPU benchmark lab
  - how to run linpack
- Additional examples in
  - /opt/mpich/gnu/examples
  - /opt/mpich/gnu/share/examples

Cleanup when an MPI Program Crashes

- MPICH in Rocks uses shared memory segments to pass messages between processes on the same node
- When an MPICH program crashes, it doesn't properly cleanup these shared memory segments
- After a program crash, run:

  $ cluster-fork sh /opt/mpich/gnu/sbin/cleanipcs

- NOTE: this removes all shared memory segments for your user id
  - If you have other live MPI programs running, this will remove their shared memory segments too and cause that program to fail

Steps to combine SGE and MPI

1. create SGE submit script
2. run *qsub* command

   ```
   $ qsub runprog.sh
   $ qsub -pe mpich 32 runprog.sh
   ```

3. check job status

   ```
   $ qstat -j
   ```

SGE submit script

- Script contents

#!/bin/tcsh

#$ -S /bin/tcsh

setenv MPI=/path/to/MPI/binaries

…

$MPI/mpirun -machinefile machines -np $NSLOTS appname

- make it executable

$ chmod +x runprog.sh

Submit file options

```
# meet given resource request
    #$ -l h_rt=600
# specify interpreting shell for the job
    #$ -S /bin/sh
# use path for stadard output of job
    #$ -o path
# execute from current dir
    #$ -cwd
# run on 32 processes in mpich PE
    #$ -pe mpich 32
# Export all environmental variables
    #$ -V
# Export these environmental variables
    #$ -v MPI_ROOT,FOOBAR=BAR
```

See “man qsub” for more options

Online resources

MPI standard:

www-unix.mcs.anl.gov/mpi

Local Area Multicomputer MPI (LAM MPI):

www.osc.edu/lam.html

MPICH:

www.mcs.anl.gov/mpi/mpich

Aggregate Function MPI (AFMPI):

garage.ecn.purdue.edu/~papers

Lam tutorial

www.lam-mpi.org/tutorials/one-step/lam.php

Glossary

MPI - message passing interface

PVM - parallel virtual machine

LAM - local area multicomputer

P4 - 3rd generation parallel programming library, includes message-passing and shared-memory components

Chameleon - high-performance portability package for message passing on parallel supercomputers

Zipcode - portable system for writing of scalable libraries

ADI - abstract device architecture

Glossary (cont'd)

SIMD - Single Instruction Multiple Data

SPMD - Single Program Multiple Data

MIMD - Multiple Instruction Multiple Data

MPMD - Multiple Program Multiple Data