ECE-451/ECE-566 - Introduction to Parallel and Distributed Programming

Lecture 12: Hands-On MPI

Department of Electrical & Computer Engineering
Rutgers University

Compiling/Executing MPI Programs

- To compile MPI program (in C/C++):
  `mpicc -o file file.c`
  or `mpiCC -o file file.cpp`
- To execute MPI program:
  `mpirun <OPTIONS> -np num_procs file <ARGS>`
  Options:
  - -v ⇒ verbose
  - -h ⇒ help
  - -machinefile hostfile ⇒ specify list of possible machines to run on
  - -nolocal ⇒ do not run on the local machine
  - -gdb ⇒ start the first process under gdb (GNU debugger)

  Example:
  `mpirun -v -machinefile all8 -gdb -np 4 file | tee myout`
- To find process status of your jobs:
  `ps -u <user_name>`
- To terminate suspended or hung processes:
  `kill -9 <process_num_id>`

3

Environmental Reqmts.

- Provide a way to specify and track processors (Group of procs is called a communicator)
- Provide a way to split communicators
- Provide means of transferring data between processors
- Provide means of synchronization
- Provide a means of closing communicators

4

Basic MPI Commands

- MPI_Init() : Initialize communicator
- MPI_Comm_size(..) : Determine total no. of procs in communicator
- MPI_Comm_rank(..) : Determine my proc_no in the communicator
- MPI_Send(..) : Send data to another proc
- MPI_Recv(..) : Recv data from another proc
- MPI_Barrier(..) : Barrier synchronization
- MPI_Finalize() : Shut down communicator

5

Demo Programs

Example ("hello world")

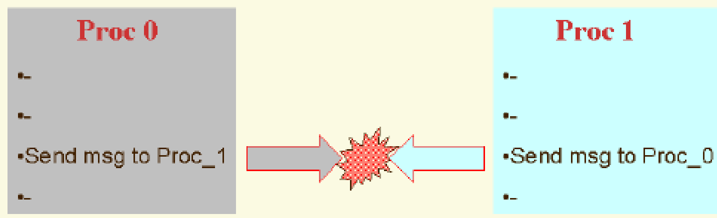
- AIM : 2 procs send messages to each other and print an acknowledgement

Proc 0		Proc 1
•Form communicator		•Form Communicator
•Get my_proc_no		•Get my proc_no
•Send msg to proc_1	➡	•Recv msg from proc_0
•Print acknowledgement		•Print acknowledgement
•Recv msg from proc_1	⬅	•Send msg to proc_0
•Print acknowledgement		•Print acknowledgement
•Close communicator		•Close communicator

7

Notable Points

- ✓ Synchronized sends and recvs
- ✓ Beware of deadlocks
- ✓ Need : Generic programs for "P" procs



8

Algorithm ("hello world")

```
Form communicator;
my_proc_no = get_proc_no();

if (my_proc_no == 0)
{
  send msg to Proc_1;
  write acknowledgement;
  recv msg from Proc_1;
  write acknowledgement ;
}
else
 {
  recv msg from Proc_0 ;
  write acknowledgement ;
  send msg to Proc_0 ;
  write acknowledgement;
 }
```

9

Hello World (1)

```
/*
"Hello World" example for 2 processors. Initially, both processors have
   status "I am alone!". Each sends out a "Hello World" to the other.
   Upon receiving each other's message, the status changes to what is
   received.
*/

#include "mpi.h"
#include <stdio.h>

int main(int argc, char** argv)
{
  int MyProc, tag=0;
  char msg[12]="Hello World";
  char msg_recpt[12]="I am alone!";
  MPI_Status status;

  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &MyProc);

  printf("Process # %d started \n", MyProc);
  MPI_Barrier(MPI_COMM_WORLD);
```

10

Hello World (2)

```
 if (MyProc == 0)
  {
    printf("Proc #0: %s \n", msg_recpt) ;

    printf("Sending message to Proc #1: %s \n", msg) ;
    MPI_Send(&msg, 12, MPI_CHAR, 1, tag, MPI_COMM_WORLD);

    MPI_Recv(&msg_recpt, 12, MPI_CHAR, 1, tag, MPI_COMM_WORLD, &status);
    printf("Received message from Proc #1: %s \n", msg_recpt) ;
  }
  else
  {
    printf("Proc #1: %s \n", msg_recpt) ;

    MPI_Recv(&msg_recpt, 12, MPI_CHAR, 0, tag, MPI_COMM_WORLD, &status);
    printf("Received message from Proc #0: %s \n", msg_recpt) ;

    printf("Sending message to Proc #0: %s \n", msg) ;
    MPI_Send(&msg, 12, MPI_CHAR, 0, tag, MPI_COMM_WORLD);
  }

MPI_Finalize();
}
```

11

Hello World – any (1)

```
/*
"Hello World" example for "p" number of processors. Initially, all
   processors have status "I am alone!". Each sends out a "Hello World"
   to all others. Upon receiving the messages, each processors's status
   changes to what is received.
*/

#include "mpi.h"
#include <stdio.h>

int main(int argc, char** argv)
{
  int MyProc, size, tag = 0;
  int send_proc = 0, recv_proc = 0;
  char msg[12]="Hello World";
  char msg_recpt[12]="I am alone!";
  MPI_Status status;

  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &MyProc);
  MPI_Comm_size(MPI_COMM_WORLD, &size);

  printf("Process # %d started \n", MyProc);
  printf("Proc #%d: %s \n", MyProc, msg_recpt) ;
  MPI_Barrier(MPI_COMM_WORLD);
```

12

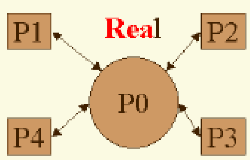
Hello World – any (2)

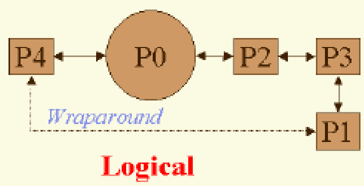
```
 for (send_proc = 0; send_proc < size; send_proc++)
  {
    if (send_proc != MyProc)
    {
      printf("Proc #%d sending message to Proc #%d: %s \n", MyProc,
   send_proc, msg);
      MPI_Send(&msg, 12, MPI_CHAR, send_proc, tag, MPI_COMM_WORLD);
    }
  }

  for (recv_proc = 0; recv_proc < size; recv_proc++)
  {
    if (recv_proc != MyProc)
    {
      MPI_Recv(&msg_recpt, 12, MPI_CHAR, recv_proc, tag, MPI_COMM_WORLD,
   &status);
      printf("Proc #%d received message from Proc #%d: %s \n", MyProc,
   recv_proc, msg_recpt);
    }
   }

  //MPI_Barrier(MPI_COMM_WORLD);
  MPI_Finalize();
}
```

13

Shaping Communicators

- Procs can be cast into different logical configurations
- Using MPI, logical neighbors, coordinates and wrap-around configs can be specified.

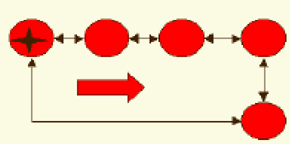




14

Example (Ring Problem)

- On a ring of "P" procs, start a msg from proc_0, send it right and stop when it returns to proc_0
- Required :
  - Cast "P" procs into a ring [MPI_Cart_create(.....)]
  - Determine left & right neighbors [MPI_Cart_shift(.....)]



15

Ring (1)

```
/*
Ring.c -> MPI example from http://www-unix.mcs.anl.gov/mpi

Write a program that takes data from process zero (0 to quit) and sends it
   to all of the other processes by sending it in a ring. That is,
   process i should receive the data and send it to process i+1, until
   the last process is reached.
Assume that the data consists of a single integer. Process zero reads the
   data from the user.
*/

#include <stdio.h>
#include "mpi.h"

int main( argc, argv )
int argc;
char **argv;
{
    int rank, value=1, size;
    MPI_Status status;

    MPI_Init( &argc, &argv );

    MPI_Comm_rank( MPI_COMM_WORLD, &rank );
    MPI_Comm_size( MPI_COMM_WORLD, &size );
```

16

Ring (2)

```
 do {
   if (rank == 0) {
       printf( "\nEnter a number (0 to quit): ");
       scanf( "%d", &value );
       MPI_Send( &value, 1, MPI_INT, rank + 1, 0, MPI_COMM_WORLD );
   }
   else {
       MPI_Recv( &value, 1, MPI_INT, rank - 1, 0, MPI_COMM_WORLD,
              &status );
       if (rank < size - 1)
        MPI_Send( &value, 1, MPI_INT, rank + 1, 0, MPI_COMM_WORLD );
   }
   printf( "Process %d got %d\n", rank, value );
 } while (value != 0);

    MPI_Finalize( );
    return 0;
}
```

17

Broadcasts and Reductions

- Broadcast : An operation where 1 proc sends the same data to all other procs
  - MPI_Bcast(…)
- Reduction : An operation where all procs send their data to 1 proc which "reduces" it to 1 data item
  - MPI_Reduce(… ..)
- Some applications need an 'all reduce" ie all procs need the reduced data item
  - MPI_Allreduce(…)

18

Example (Integer Sum)

- Problem : Sum up all integers between N1 and N2
- Given : "P" processors
- Input : n1 & n2
- Procedure :
  - **Proc_0 reads n1, n2; broadcasts it**
  - **Each proc divides [n1,n2] into equal intervals; chooses one**
  - **Each procs sums over its own interval**
  - **Each proc sends its sum to proc_0 which reduces it to the grand total**

19

Algorithm (Integer Sum)

(1)

Init communicator ;
get comm. Size ;
get MyProc ;
if (MyProc == 0) read (n1, n2);

Broadcast(n1, n2) from proc_0 to all ;
interval = ceil((n2-n1)/P) ;
start = n1 + MyProc*interval
end = start + interval

(2)

for (i = start; i <end; i++)
 sum = sum + I ;
Reduce (sum, grand_total) ;
if(MyProc == 0)
 print(grand_total) ;

20

Integer Sum (1)

```
/*
This program computes the sum of all integers in an interval whose end-
    points (left and right limits) are specified by the user. This data is
    read by the root process and broadcast to all other processors in the
    communicator. Each processor determines its local range of integers
    and computes the partial sums. These partial sums are sent back to the
    root where the grand total is generated and reported to the user.
*/

#include "mpi.h"
#include <stdio.h>

int main(int argc, char **argv)
{
  int MyProc, tag=1, size;
  char msg='A', msg_recpt ;
  MPI_Status *status ;
  int root ;
  int left, right, interval ;
  int number, start, end, sum, GrandTotal;
  int mystart, myend;

  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &MyProc);
  MPI_Comm_size(MPI_COMM_WORLD, &size);
```

21

Integer Sum (2)

```
root = 0;
 if (MyProc == root)    /* Proc root reads the limits in */
 {
   printf("Give the left and right limits of the interval\n");
   scanf("%d %d", &left, &right);
   printf("Proc root reporting : the limits are : %d %d\n", left, right);
 }
 MPI_Bcast(&left, 1, MPI_INT, root, MPI_COMM_WORLD); /*Bcast limits to all*/
 MPI_Bcast(&right, 1, MPI_INT, root, MPI_COMM_WORLD);

 if (((right - left + 1) % size) != 0)
   interval = (right - left + 1) / size + 1 ;  /*Fix local limits of summing*/
 else
   interval = (right - left + 1) / size;
 mystart = left + MyProc*interval ;
 myend = mystart + interval ;
 /* set correct limits if interval is not a multiple of size */
 if (myend > right) myend = right + 1 ;

 sum = root;                       /* Sum locally on each proc */
 if (mystart <= right)
   for (number = mystart; number < myend; number++) sum = sum + number ;
```

22

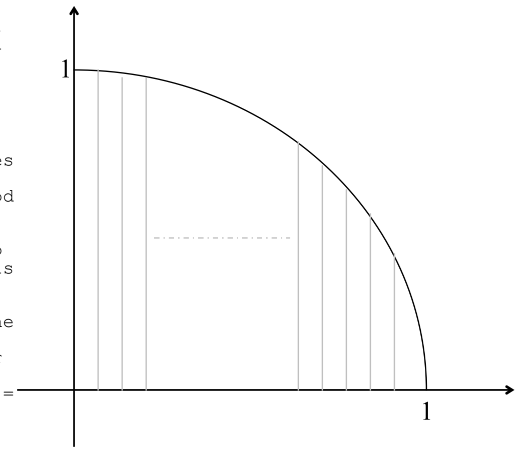
Integer Sum (3)

```
 /* Do reduction on proc root */
  MPI_Reduce(&sum, &GrandTotal, 1, MPI_INT, MPI_SUM, root, MPI_COMM_WORLD) ;
  MPI_Barrier(MPI_COMM_WORLD);

  /* Root reports the results */
  if(MyProc == root)
    printf("Proc root reporting : Grand total = %d \n", GrandTotal);

  MPI_Finalize();
}
```

23

Computing Pi

```
/*
PI calculation -> MPI example from
   http://www-unix.mcs.anl.gov/mpi

This exercise presents a simple
   program to determine the value
   of pi. The algorithm suggested
   here is chosen for its
   simplicity. The method evaluates
   the integral of 4/(1+x*x)
   between -1/2 and 1/2. The method
   is simple: the integral is
   approximated by a sum of n
   intervals; the approximation to
   the integral in each interval is
   (1/n)*4/(1+x*x). The master
   process (rank 0) asks the user
   for the number of intervals; the
   master should then broadcast
   this number to all of the other
   processes. Each process then
   adds up every n'th interval (x =
   -1/2+rank/n, -1/2+rank/n+size/
   n,...). Finally, the sums
   computed by each process are
   added together using a
   reduction.
*/
```



24

Computing Pi (1)

```
#include "mpi.h"
#include <math.h>

int main(argc,argv)
int argc;
char *argv[];
{
    int done = 0, n, myid, numprocs, i;
    double PI25DT = 3.141592653589793238462643;
    double mypi, pi, h, sum, x;

    MPI_Init(&argc,&argv);
    MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
    MPI_Comm_rank(MPI_COMM_WORLD,&myid);
    while (!done)
    {
   if (myid == 0) {
       printf("\nEnter the number of intervals: (0 quits) ");
       scanf("%d",&n);
   }
   MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);
   if (n == 0) break;
```

25

Computing Pi (2)

```
    h   = 1.0 / (double) n;
    sum = 0.0;
    for (i = myid + 1; i <= n; i += numprocs) {
        x = h * ((double)i - 0.5);
        sum += 4.0 / (1.0 + x*x);
    }
    mypi = h * sum;

    MPI_Reduce(&mypi, &pi, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);

    if (myid == 0)
        printf("\nPI is approximately %.16f, Error is %.16f\n",
            pi, fabs(pi - PI25DT));
     }
     MPI_Finalize();
     return 0;
}
```

26

Recapitulation

- ✓ MPI is a library of functions
- ✓ Basic working environment (proc group) called communicator
- ✓ Has functions for point-to-point proc communication, broadcasts, reductions
- ✓ Has the ability to cast the communicator into various logical shapes
- ✓ MPI can be small (~20 functions) or elaborate (~130 functions).
- ✓ Industry standard

32