Introduction to MPI Part 1

Dr. Charles Cavanaugh
9-28-11
Overview

• Definition and rationale
• Important considerations
• Getting started
• Essential functions
• Exercises
What is Message Passing Interface (MPI)?

- A communication library for parallel computing over a distributed system.
- Distributed=loosely coupled computers—they do not share a clock or memory.
- Programmer writes one program in C or Fortran.
- The one program is loaded on one or more processors and called a process.
Major Parallel Computing Models

• Shared memory
  – Much like pthreads (multithreading library)
  – Processes (lightweight processes or threads) all share memory (i.e. globals)
  – Threads have some private memory (stack)
  – Locking is needed and handled through monitors and semaphores
Major Parallel Computing Models

• Message Passing
  – Each process has its own memory
  – Programmer handles any sharing of data by passing messages
  – MPI falls into this category
Parallel Programming Types

• Data Parallel: Single Instruction Multiple Data (SIMD) – instructions are same but work on different data
• Task Parallel: Multiple Instructions Multiple Data (MIMD) – instructions differ as does the data
• Single Program Multiple Data (SPMD): *program* (think source) is the same, but different parts of it execute depending on process rank (equiv. to MIMD)
• *Programming with MPI is typically SPMD.*
SPMD Diagram

Process 0

Code:

MPI_Init...
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
if(rank==0) {
  ...
} else {
  ...
}

Process 1

Code:

MPI_Init...
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
if(rank==0) {
  ...
} else {
  ...
}
## Multiple Data

<table>
<thead>
<tr>
<th>Array</th>
<th>Processor</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
</tr>
</tbody>
</table>
Why Use MPI?

- It allows one to parallelize his/her own programs.
- It is a solid communication library for distributed computing.
Process

• Each *process* is really just the one program, but usually executes differently on each *processor (core)*.
• Each process has a *rank* from 0 .. #CPUs, which is accessible from the running process.
• The programmer generally writes conditional blocks depending on this *rank*.
• Depending on the rank, the process may:
  – work on a different part of the problem space (*slave*),
  – split up a problem space and distribute the work (*master*),
  – combine results and save/display them (*master*), or
  – anything else the programmer wishes to do.
Writing a Program

• MPI has many functions in its library.
  – Point-to-point
  – Collective
• With only syntactic differences, the functions are the same in C and Fortran.
Compiling a Program

• Compiling code only involves:
  – Including a C or Fortran header file
  – Compiling using included special compiler wrapper

• Running code is as easy as using mpirun or placing in a batch script for use with qsub.
Basic MPI Program Structure

Call to Initialize MPI library.
If rank ≠ 0: //Here, the master is 0.
   Read input data (from master or a file).
   Do necessary operations.
   Send result to process 0.
Else:
   Send all/part of input data to slave processes if necessary.
For each process i:
   Receive result from process i.
   Combine result with final result.
Output final result.
Call to Finalize MPI library.
Basic MPI Program in C

```c
#include <stdio.h>
#include <mpi.h>
main(int argc, char **argv) {
    int np, rank;
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &np);
    printf("Hello World on process %d of %d.\n", rank, np);
    MPI_Finalize();
}
```
Compiling & Running Program

```
$ mpicc -o 00basic 00basic.c
$ qsub -I -l nodes=1:ppn=4 -l walltime=00:01:00 -q workq
...
$ mpirun -np 4 00basic
```

script:

```
#!/bin/bash
#PBS -q workq
#PBS -A loni_ccavatrain
#PBS -l nodes=1:ppn=4
#PBS -l walltime=00:00:01
#PBS -o 00basic.out
#PBS -j oe
#PBS -N basic
export WORK_DIR=/work/ccava/mpi
cd $WORK_DIR
export NPROCS=`wc -l $PBS_NODEFILE |gawk '/{print $1}'`
mpirun -machinefile $PBS_NODEFILE -np $NPROCS
/home/ccava/mpi/00basic
```

```
$ qsub script
```
Using Rank to Affect Flow

MPI_Comm_rank(MPI_COMM_WORLD, &rank);
MPI_Comm_size(MPI_COMM_WORLD, &np);
if( rank == 0 ) { /* process 0 */ }
else if( rank == 1 ) { /* process 1 */ }
else if( rank == 2 ) { /* process 2 */ }
else { /* other process */ }
Sending Messages: MPI_Send

int MPI_Send( void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm )
buf - initial address of send buffer (choice)
count - number of elements in send buffer (nonnegative integer)
datatype - datatype of each send buffer element (handle)
dest - rank of destination (integer)
tag - message tag (integer)
comm - communicator (handle)
Receiving Messages: MPI_Recv

```c
int MPI_Recv( void *buf, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI_Status *status )
```

- `buf` - initial address of receive buffer (choice)
- `status` - status object (Status)
- `count` - maximum number of elements in receive buffer (integer)
- `datatype` - datatype of each receive buffer element (handle)
- `source` - rank of source (integer)
- `tag` - message tag (integer)
- `comm` - communicator (handle)
MPI_Send & MPI_Recv Example

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

int main(argc,argv) int argc; char *argv[]; {
    int numtasks, rank, dest, source, rc, count, tag=1;
    char inmsg, outmsg='x';
    MPI_Status Stat;
    MPI_Init(&argc,&argv);
    MPI_Comm_size(MPI_COMM_WORLD, &numtasks);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    if (rank == 0) {
        dest = 1;
        source = 1;
        rc = MPI_Send(&outmsg, 1, MPI_CHAR, dest, tag, MPI_COMM_WORLD);
        rc = MPI_Recv(&inmsg, 1, MPI_CHAR, source, tag, MPI_COMM_WORLD, &Stat);
    } else if (rank == 1) {
        dest = 0;
        source = 0;
        rc = MPI_Recv(&inmsg, 1, MPI_CHAR, source, tag, MPI_COMM_WORLD, &Stat);
        rc = MPI_Send(&outmsg, 1, MPI_CHAR, dest, tag, MPI_COMM_WORLD);
    }
    rc = MPI_Get_count(&Stat, MPI_CHAR, &count);
    printf("Task %d: Received %d char(s) from task %d with tag %d \n", rank, count, Stat.MPI_SOURCE, Stat.MPI_TAG);
    MPI_Finalize();
}
Data Types

MPI_CHAR
MPI_SHORT
MPI_INT
MPI_LONG
MPI_UNSIGNED_CHAR
MPI_UNSIGNED_SHORT
MPI_UNSIGNED_LONG
MPI_UNSIGNED
MPI_FLOAT
MPI_DOUBLE
MPI_LONG_DOUBLE
MPI_BYTE
MPI_PACKED
Distributing Data: MPI_Bcast

`int MPI_Bcast ( void *buffer, int count, MPI_Datatype datatype, int root, MPI_Comm comm )`

- **buffer** - starting address of buffer (choice)
- **count** - number of entries in buffer (integer)
- **datatype** - data type of buffer (handle)
- **root** - rank of broadcast root (integer)
- **comm** - communicator (handle)
Collecting & Calculating with Data: MPI_Reduce

int MPI_Reduce ( void *sendbuf, void *recvbuf, int count, MPI_Datatype datatype, MPI_Op op, int root, MPI_Comm comm )

sendbuf - address of send buffer (choice)
count - number of elements in send buffer (integer)
datatype - data type of elements of send buffer (handle)
op - reduce operation (handle)
root - rank of root process (integer)
comm - communicator (handle)
recvbuf - address of receive buffer (choice, significant only at root )
MPI_Op Values

MPI_MAX maximum
MPI_MIN minimum
MPI_SUM sum
MPI_PROD product
MPI_LAND logical and
MPI_BAND bit-wise and
MPI_LOR logical or
MPI_BOR bit-wise or
MPI_LXOR logical xor
MPI_BXOR bit-wise xor
MPI_MAXLOC max value and location
MPI_MINLOC min value and location

Programmer may define own operation (using MPI_Op_create)
#include "mpi.h"
#include <math.h>

int main(argc,argv) int argc; char *argv[]; {
    int done = 0, n, myid, numprocs, i, rc;
    double PI25DT = 3.141592653589793238462643;
    double mypi, pi, h, sum, x, a;
    MPI_Init(&argc,&argv);
    MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
    MPI_Comm_rank(MPI_COMM_WORLD,&myid);
    while (!done){
        if (myid == 0) {
            printf("Enter the number of intervals: (0 quits) ");
            scanf("%d",&n);
        }
        MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);
        if (n == 0) break;
        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("pi is approximately %.16f, Error is %.16f\n", pi, fabs(pi - PI25DT));
    }
    MPI_Finalize();
}
MPI_BARRIER

Synchronizes all processes by blocking until all reach this function.

int MPI_BARRIER ( MPI_Comm comm )

comm - communicator (handle)
Using MPI_Barrier to Compute Time

double start, end;
...
MPI_Barrier(MPI_COMM_World);
if(rank==0) start = MPI_Wtime(); //seconds
...
MPI_Barrier(MPI_COMM_World);
if(rank==0) end = MPI_Wtime();
...
Exercises