Introduction to MPI Part 2

Dr. Charles Cavanaugh

10-5-11

Overview

- Review of Part 1
- Rationale
- Non-blocking communication
- Advanced collective communication
- User-defined data types
- Exercises

REVIEW

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.*

Parallel Computation via Message Passing

- Each process has its own memory
- Programmer handles any sharing of data by passing messages
- MPI falls into this category

Terminology: *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.*

Terminology: *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.

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.
```

REVIEW

Compiling & Running a 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
```

Advanced Communication Features We'll Cover

- Non-blocking communication
  - Sending and receiving
  - Checking for completion
- Advanced collective communication
  - Broadcast
  - Scatter & gather

Why Use Advanced Communication Features?

- Non-blocking communication
  - Increase performance of code by overlapping computation and communication.
- Advanced collective communication
  - Leverage tested, optimized collective communication routines.

Advanced Communication Features We'll Cover

- Non-blocking communication
  - Sending and receiving
  - Checking for completion
- Advanced collective communication
  - Broadcast
  - Scatter & gather

Non-blocking Send: MPI_Isend

```
int MPI_Isend(void *buf, int count, MPI_Datatype
datatype, int dest, int tag, MPI_Comm comm,
MPI_Request *request)
```

buf	Initial address of send buffer (choice).
count	Number of elements in send buffer (integer).
datatype	Datatype of each send buffer element (handle).
dest	Rank of destination (integer).
tag	Message tag (integer).
comm	Communicator (handle).
request	Communication request (handle).
IERROR	Fortran only: Error status (integer).

Non-blocking Receive: MPI_Irecv

```
int MPI_Irecv(void *buf, int count, MPI_Datatype
datatype, int source, int tag, MPI_Comm comm,
MPI_Request *request)
buf       Initial address of receive buffer (choice).
count     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).
request   Communication request (handle).
IERROR    Fortran only: Error status (integer).
```

Testing for Completion: MPI_Test

```
int MPI_Test(MPI_Request *request, int *flag,
MPI_Status *status)
request   Communication request (handle).
flag      True if operation completed (logical).
status    Status object (status).
IERROR    Fortran only: Error status (integer).
```

Waiting for Completion: MPI_Wait

```
int MPI_Wait(MPI_Request *request, MPI_Status *status)
request   Communication request (handle).
flag      True if operation completed (logical).
status    Status object (status).
IERROR    Fortran only: Error status (integer).
```

Non-blocking Example

Source: http://users.abo.fi/Mats.Aspnas/PP2010/examples/MPI/send-nonblocking-wait.c

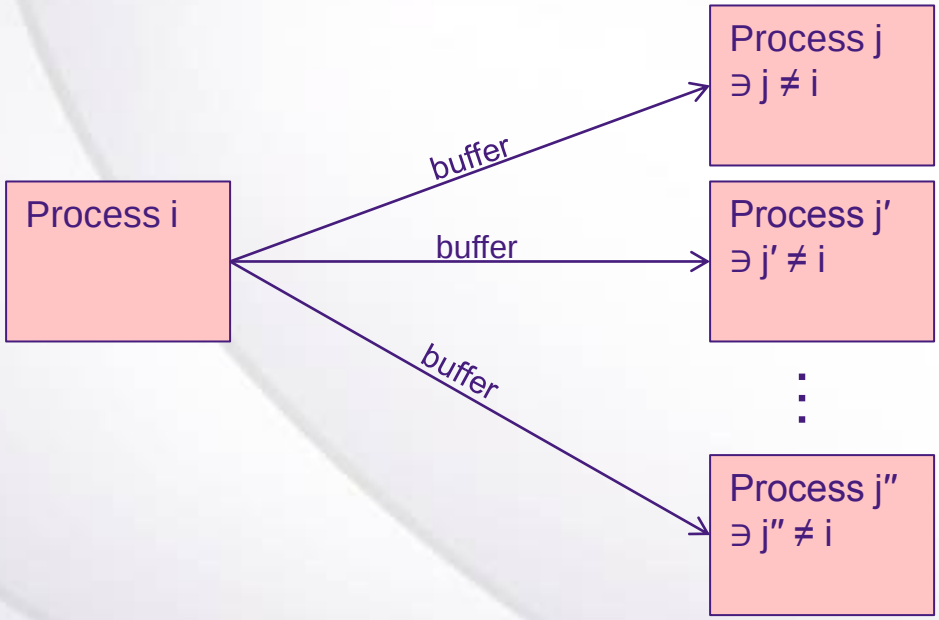
```
#include <stdlib.h>
#include <stdio.h>
#include "mpi.h"
int main(int argc, char* argv[]) {
  int x, y, np, me;
  int tag = 42;
  MPI_Status status;
  MPI_Request send_req, recv_req; /* Request
object for send and receive */
  MPI_Init(&argc, &argv);  /* Initialize MPI */
  MPI_Comm_size(MPI_COMM_WORLD, &np);  /* Get
number of processes */
  MPI_Comm_rank(MPI_COMM_WORLD, &me);  /* Get
own identifier */
  /* Check that we run on exactly two processors
*/
  if (np != 2) {
    if (me == 0) {
      printf("You have to use exactly 2
processors to run this program\n");
    }
    MPI_Finalize();    /* Quit if there is only
one processor */
    exit(0);
  }
```

```
x = me;
  if (me == 0) { /* Process 0 does this */
    printf("Process %d sending to process 1\n",
me);
    MPI_Isend(&x, 1, MPI_INT, 1, tag,
MPI_COMM_WORLD, &send_req);
    printf("Process %d receiving from process
1\n", me);
    MPI_Irecv (&y, 1, MPI_INT, 1, tag,
MPI_COMM_WORLD, &recv_req);
 /* We could do computations here while we are
waiting for communication */
    MPI_Wait(&send_req, &status);
    MPI_Wait(&recv_req, &status);
    printf("Process %d ready\n", me);
  } else {
    MPI_Irecv (&y, 1, MPI_INT, 0, tag,
MPI_COMM_WORLD, &recv_req);
    MPI_Isend (&y, 1, MPI_INT, 0, tag,
MPI_COMM_WORLD, &send_req);
 /* We could do computations here while we are
waiting for communication */
    MPI_Wait(&recv_req, &status);
    MPI_Wait(&send_req, &status);
  }
  MPI_Finalize();
  exit(0);
}
```

Broadcast Identical 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).
IERROR    Fortran only: Error status (integer).
```

Broadcast Identical Data: MPI_Bcast



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)

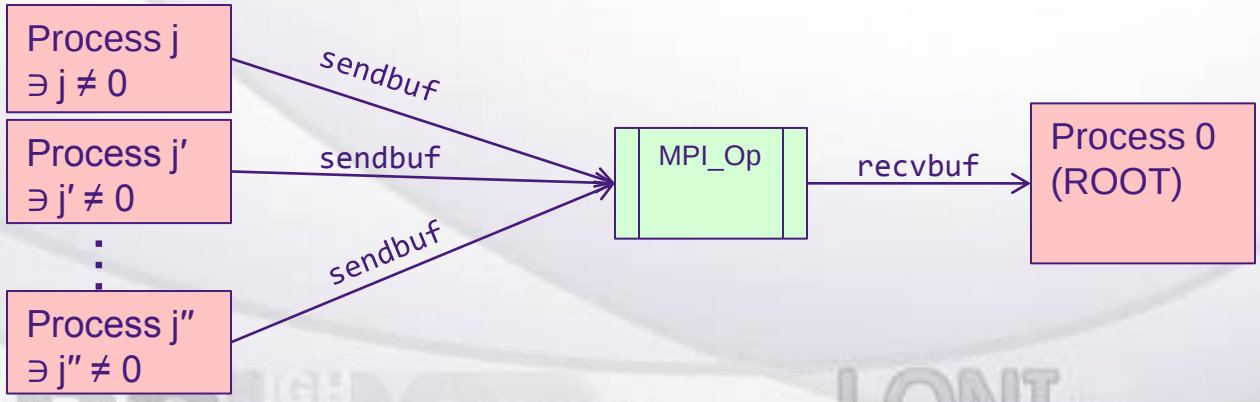
datatype - data type of elements of send buffer (handle)

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)

MPI_Bcast & MPI_Reduce Example

```
#include "mpi.h"
#include <math.h>
int main(argc,argv) int argc; char *argv[]; {
  int done = 0, n, myid, numprocs, i, rc;
  double PI25DT = 3.14159265358979323846264 3;
  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(); }
```

Broadcast Different Data: MPI_Scatter

```
int MPI_Scatter(void *sendbuf, int sendcount,
MPI_Datatype sendtype, void *recvbuf, int recvcount,
MPI_Datatype recvtype, int root, MPI_Comm comm)
```

sendcount Number of elements sent to each process (integer, significant only at root).

sendtype Datatype of send buffer elements (handle, significant only at root).

recvcount Number of elements in receive buffer (integer).

recvtype Datatype of receive buffer elements (handle).

root Rank of sending process (integer).

comm Communicator (handle).

recvbuf Address of receive buffer (choice).

IERROR Fortran only: Error status (integer).

MPI_Scatter

- Inverse of MPI_Gather.
- Equivalent to root executing:
  - MPI_Send(sendbuf + i * sendcount * extent(sendtype), sendcount, sendtype, i, ...) for i = 0..n
- …and other processes executing:
  - MPI_Recv(recvbuf, recvcount, recvtype, i, ...) for i = 0..n

Scatter Example

Source: http://users.abo.fi/Mats.Aspnas/PP2010/examples/MPI/scatter.c

```
#include <stdlib.h>
#include <stdio.h>
#include "mpi.h"
#define MAXPROC 8 /* Max number of procsses */
#define LENGTH 24 /* Lengt of send buffer is
divisible by 2, 4, 6 and 8 */
int main(int argc, char* argv[]) {
  int i, j, np, me;
  const int nametag  = 42; /* Tag value for
sending name */
  const int datatag  = 43; /* Tag value for
sending data */
  const int root = 0;   /* Root process in
scatter */
  MPI_Status status; /* Status object for
receive */
  char myname[MPI_MAX_PROCESSOR_NAME]; /* Local
host name string */
  char
hostname[MAXPROC][MPI_MAX_PROCESSOR_NAME];  /*
Received host names */
  int namelen;
  int x[LENGTH];  /* Send buffer */
  int y[LENGTH];  /* Receive buffer */
  MPI_Init(&argc, &argv); /* Initialize MPI */
  MPI_Comm_size(MPI_COMM_WORLD, &np); /* Get nr
of processes */
```

```
MPI_Comm_rank(MPI_COMM_WORLD, &me); /* Get own
identifier */

  MPI_Get_processor_name(myname, &namelen);  /*
Get host name */
  myname[namelen++] = (char)0;   /* Terminating
null byte */
  /* Check that we have an even number of
processes and at most MAXPROC */
  if (np>MAXPROC || np%2 != 0) {
    if (me == 0) {
      printf("You have to use an even number of
processes (at most %d)\n", MAXPROC);
    }
    MPI_Finalize();
    exit(0);
  }
  if (me == 0) { /* Process 0 does this */

 /* Initialize the array x with values 0 ..
LENGTH-1 */
    for (i=0; i<LENGTH; i++) {
      x[i] = i;
    }
    printf("Process %d on host %s is distributing
array x to all %d processes\n\n",
       me, myname, np);
```

Scatter Example

Source: http://users.abo.fi/Mats.Aspnas/PP2010/examples/MPI/scatter.c

```
/* Scatter the array x to all proceses, place it
in y */
    MPI_Scatter(x, LENGTH/np, MPI_INT, y,
LENGTH/np, MPI_INT, root,
        MPI_COMM_WORLD);
 /* Print out own portion of the scattered array
*/
    printf("Process %d on host %s has elements",
me, myname);
    for (i=0; i<LENGTH/np; i++) {
      printf(" %d", y[i]);
    }
    printf("\n");
 /* Receive messages with hostname and the
scattered data */
 /* from all other processes */
    for (i=1; i<np; i++) {
      MPI_Recv (hostname[i], namelen, MPI_CHAR,
i, nametag, MPI_COMM_WORLD,
        &status);
      MPI_Recv (y, LENGTH/np, MPI_INT, i,
datatag, MPI_COMM_WORLD, &status);
      printf("Process %d on host %s has
elements", i, hostname[i]);
      for (j=0; j<LENGTH/np; j++) {
    printf(" %d", y[j]);
      }
      printf("\n");
  }

    printf("Ready\n");

  } else { /* all other processes do this */
 /* Receive the scattered array from process 0,
place it in array y */
    MPI_Scatter(x, LENGTH/np, MPI_INT, y,
LENGTH/np, MPI_INT, root, \
        MPI_COMM_WORLD);
 /* Send own name back to process 0 */
    MPI_Send (myname, namelen, MPI_CHAR, 0,
nametag, MPI_COMM_WORLD);
 /* Send the received array back to process 0 */
    MPI_Send (y, LENGTH/np, MPI_INT, 0, datatag,
MPI_COMM_WORLD);
  }
  MPI_Finalize();
  exit(0);
}
```

Gather values from group of processes: MPI_Gather

```
int MPI_Gather(void *sendbuf, int sendcount, MPI_Datatype
sendtype, void *recvbuf, int recvcount, MPI_Datatype
recvtype, int root, MPI_Comm comm)
sendbuf   Starting address of send buffer (choice).
sendcount Number of elements in send buffer (integer).
sendtype  Datatype of send buffer elements (handle).
recvcount Number  of elements for any single receive
(integer, significant only at root).
recvtype  Datatype of recvbuffer elements (handle,
significant only  at root).
root      Rank of receiving process (integer).
comm      Communicator (handle).
recvbuf   Address of receive buffer (choice, significant
only at root).
IERROR    Fortran only: Error status (integer).
```

MPI_Gather

- Inverse of MPI_Scatter.
- Equivalent to each process executing:
  - MPI_Send(sendbuf, sendcount, sendtype, root, ...)
- and the root had executed n calls to:
  - MPI_Recv(recfbuf + i * recvcount * extent(recvtype), recvcount, recvtype, i, ...) for i = 0..n

Gather Example

Source: http://users.abo.fi/Mats.Aspnas/PP2010/examples/MPI/gather.c

```
#include <stdlib.h>
#include <unistd.h>
#include <stdio.h>
#include "mpi.h"
#define MAXPROC 8 /* Max number of procsses */
#define NAMELEN 80   /* Max length of machine name */
#define LENGTH 24 /* Lengt of send buffer is divisible by 2, 4, 6 and 8 */
int main(int argc, char* argv[]) {
  int i, j, np, me;
  const int nametag  = 42; /* Tag value for sending name */
  const int datatag  = 43; /* Tag value for sending data */
  const int root = 0;   /* Root process in scatter */
  MPI_Status status; /* Status object for receive */
  char myname[NAMELEN]; /* Local host name string */
  char hostname[MAXPROC][NAMELEN];  /* Received host names */
  int x[LENGTH];  /* Send buffer */
  int y[LENGTH];  /* Receive buffer */
  MPI_Init(&argc, &argv); /* Initialize MPI */
  MPI_Comm_size(MPI_COMM_WORLD, &np); /* Get nr of processes */
```

```
 MPI_Comm_rank(MPI_COMM_WORLD, &me); /* Get own identifier */

  gethostname(myname, NAMELEN); /* Get host name */
  /* Check that we have an even number of processes and at most MAXPROC */
  if (np>MAXPROC || np%2 != 0) {
    if (me == 0) {
      printf("You have to use an even number of processes (at most %d)\n", MAXPROC);
    }
    MPI_Finalize();
    exit(0);
  }
  /* Each process initializes its local array */
  for (i=0; i<LENGTH/np; i++) {
    x[i] = (LENGTH/np)*me+i;
  }
  if (me == 0) { /* Process 0 does this */

    printf("Process %d on host %s is gathering array x from all %d processes\n\n", \
       me, myname, np);
```

Gather Example

Source: http://users.abo.fi/Mats.Aspnas/PP2010/examples/MPI/gather.c

```
 /* Gather the array x from all proceses, place
it in y */
    MPI_Gather(x, LENGTH/np, MPI_INT, y,
LENGTH/np, MPI_INT, root, MPI_COMM_WORLD);
 /* Print out the gathered array */
    printf("Process %d on host %s got
elements\n", me, myname);
    for (i=0; i<LENGTH; i++) {
      printf(" %d", y[i]);
    }
    printf("\n\n");
 /* Print out the local array x on process 0 */
    printf("Process %d on host %s had elements",
me, myname);
    for (i=0; i<LENGTH/np; i++) {
      printf (" %d", x[i]);
    }
    printf ("\n");
 /* Receive messages with hostname and the
original data */
 /* from all other processes */
    for (i=1; i<np; i++) {
      MPI_Recv (&hostname[i], NAMELEN, MPI_CHAR,
i, nametag, MPI_COMM_WORLD, \
        &status);
      MPI_Recv (&y, LENGTH/np, MPI_INT, i,
datatag, MPI_COMM_WORLD, &status);
 printf("Process %d on host %s had elements", i,
hostname[i]);
      for (j=0; j<LENGTH/np; j++) {
    printf(" %d", y[j]);
      }
      printf("\n");
    }

    printf("Ready\n");
  } else { /* all other processes do this */
 /* Receive the scattered array from process 0,
place it in array y */
    MPI_Gather(&x, LENGTH/np, MPI_INT, &y,
LENGTH/np, MPI_INT, root, \
        MPI_COMM_WORLD);
 /* Send own name back to process 0 */
    MPI_Send (&myname, NAMELEN, MPI_CHAR, 0,
nametag, MPI_COMM_WORLD);
 /* Send the received array back to process 0 */
    MPI_Send (&x, LENGTH/np, MPI_INT, 0, datatag,
MPI_COMM_WORLD);
  }
  MPI_Finalize();
  exit(0);
}
```

Exercises