More MPI

Outline

- **Review of last week**
  - 6 Basic MPI Calls
  - Data Types
  - Wildcards
  - Using Status
- **Probing**
- **Asynchronous Communication**
- **Collective Communications**
- **Advanced Topics**
  - "V" operations
  - Derived data types
  - Communicators

Review: Six Basic MPI Calls

- **MPI_Init**
  - Initialize MPI
- **MPI_Comm_Rank**
  - Get the processor rank
- **MPI_Comm_Size**
  - Get the number of processors
- **MPI_Send**
  - Send data to another processor
- **MPI_Recv**
  - Get data from another processor
- **MPI_Finalize**
  - Finish MPI

Simple Send and Receive Program

```
#include <stdio.h>
#include <mpi.h>
int main(int argc,char *argv[])
{
  int myid, numprocs, tag,source,destination,count, buffer;
  MPI_Status status;
  MPI_Init(&argc,&argv);
  MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
  MPI_Comm_rank(MPI_COMM_WORLD,&myid);
  tag=1234;  source=0;  destination=1;  count=1;
  if(myid == source){
    buffer=5678;
    MPI_Send(&buffer,count,MPI_INT,destination,tag,MPI_COMM_WORLD);
    printf("processor %d  sent %d\n",myid,buffer);
  }
  if(myid == destination){
    MPI_Recv(&buffer,count,MPI_INT,source,tag,MPI_COMM_WORLD,&status);
    printf("processor %d  got %d\n",myid,buffer);
  }
  MPI_Finalize();
}
```

MPI Data Types

- **MPI has many different predefined data types**
  - All your favorite C data types are included
- **MPI data types can be used in any communication operation**

MPI Predefined Data Types in C

C MPI Types	
MPI_CHAR	signed char
MPI_SHORT	signed short int
MPI_INT	signed int
MPI_LONG	signed long int
MPI_UNSIGNED_CHAR	unsigned char
MPI_UNSIGNED_SHORT	unsigned short int
MPI_UNSIGNED	unsigned int
MPI_UNSIGNED_LONG	unsigned long int
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_LONG_DOUBLE	long double
MPI_BYTE	-
MPI_PACKED	-

Wildcards

- **Enables programmer to avoid having to specify a tag and/or source.**
- **Example:**

```
MPI_Status status;
int buffer[5];
int error;
error = MPI_Recv(&buffer[0], 5, MPI_INT,
                 MPI_ANY_SOURCE, MPI_ANY_TAG,
                 MPI_COMM_WORLD,&status);
```

- **MPI_ANY_SOURCE and MPI_ANY_TAG are wild cards**
- **status structure is used to get wildcard values**

Status

- **The status parameter returns additional information for some MPI routines**
  - additional error status information
  - additional information with wildcard parameters
- **C declaration : a predefined struct**
  - `MPI_Status status;`
- **Fortran declaration : an array is used instead**
  - `INTEGER STATUS(MPI_STATUS_SIZE)`

Accessing Status Information

- **The tag of a received message**
  - C : status.MPI_TAG
  - Fortran : STATUS(MPI_TAG)
- **The source of a received message**
  - C : status.MPI_SOURCE
  - Fortran : STATUS(MPI_SOURCE)
- **The error code of the MPI call**
  - C : status.MPI_ERROR
  - Fortran : STATUS(MPI_ERROR)
- **Other uses...**

MPI_Probe

- **MPI_Probe allows incoming messages to be checked without actually receiving them.**
  - The user can then decide how to receive the data.
  - Useful when different action needs to be taken depending on the "who, what, and how much" information of the message.

MPI_Probe

- **C**
  - `MPI_Probe(source, tag, comm, &status)`
- **Fortran**
  - `MPI_PROBE(SOURCE, TAG, COMM, STATUS, IERROR)`
- **Parameters**
  - Source: source rank, or MPI_ANY_SOURCE
  - Tag: tag value, or MPI_ANY_TAG
  - Comm: communicator
  - Status: status object

MPI_Probe Example

```
#include <stdio.h>
#include <mpi.h>
#include <math.h>
/* Program shows how to use probe and get_count to find the
   size */
/* of an incomming message */
int main(argc,argv)
int argc;
char *argv[];
{
    int myid, numprocs;
    MPI_Status status;
    int i,mytag,ierr,icount;
    MPI_Init(&argc,&argv);
    MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
    MPI_Comm_rank(MPI_COMM_WORLD,&myid);
/* print out my rank and this run's PE size*/
    printf("Hello from %d\n",myid);
    printf("Numprocs is %d\n",numprocs);
```

MPI_Probe example (cont.)

```
mytag=123;
    i=0;
    icount=0;
    if(myid == 0) {
        i=100;
        icount=1;

        ierr=MPI_Send(&i,icount,MPI_INT,1,mytag,MPI_COMM_WORLD
   );
    }
    if(myid == 1){
        ierr=MPI_Probe(0,mytag,MPI_COMM_WORLD,&status);
        ierr=MPI_Get_count(&status,MPI_INT,&icount);
        printf("getting %d\n",icount);
        ierr =
   MPI_Recv(&i,icount,MPI_INT,0,mytag,MPI_COMM_WORLD,
                          status);
        printf("i= %d\n",i);
    }
    MPI_Finalize();
}
```

MPI_Barrier

- **Blocks the caller until all members in the communicator have called it.**
- **Used as a synchronization tool.**
- **C**
  - `MPI_Barrier(comm )`
- **Fortran**
  - `Call MPI_BARRIER(COMM, IERROR)`
- **Parameter**
  - Comm: communicator (often MPI_COMM_WORLD)

Asynchronous Communication

- **Asynchronous send: send call returns immediately, send actually occurs later**
- **Asynchronous receive: receive call returns immediately. When received data is needed, call a wait subroutine**
- **Asynchronous communication used to overlap communication with computation.**
- **Can help prevent deadlock (but beware!)**

Asynchronous Send with MPI_Isend

- C
  - `MPI_Request request`
  - `MPI_Isend(&buffer, count, datatype, dest,tag, comm, &request)`
- Fortran
  - `Integer REQUEST`
  - `MPI_Isend(BUFFER, COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR)`
- **Request** is a new output parameter
- Don't change data until communication is complete

Asynchronous Receive w/ MPI_Irecv

- **C**
  - `MPI_Request request;`
  - `int MPI_Irecv(&buf, count, datatype, source, tag,`
    `comm, request)`
- **Fortran**
  - `Integer request`
  - `MPI_Irecv(BUFFER, COUNT, DATATYPE, SOURCE, TAG,`
    `COMM, REQUEST,IERROR)`
- **Parameter changes**
  - Request: communication request
  - Status parameter is missing
- **Don't use data until communication is complete**

MPI_Wait Used to Complete Communication

- **Request from Isend or Irecv is input**
  - The completion of a send operation indicates that the sender is now free to update the data in the send buffer
  - The completion of a receive operation indicates that the receive buffer contains the received message
- **MPI_Wait blocks until message specified by "request" completes**

MPI_Wait Usage

- C
  - `MPI_Request request;`
  - `MPI_Status status;`
  - `MPI_Wait(&request, &status)`
- Fortran
  - `Integer request`
  - `Integer status(MPI_STATUS_SIZE)`
  - `MPI_WAIT(REQUEST, STATUS, IERROR)`
- MPI_Wait blocks until message specified by "request" completes

MPI_Test

- Similar to MPI_Wait, but does not block
- Value of flags signifies whether a message has been delivered
- C
  - `int flag`
  - `int MPI_Test(&request,&flag, &status)`
- Fortran
  - `LOGICAL FLAG`
  - `MPI_TEST(REQUEST, FLAG, STATUS, IER)`

Non-Blocking Send Example

```
   call MPI_Isend (buffer,count,datatype,dest,
                   tag,comm, request, ierr)
10 continue

   Do other work ...

   call MPI_Test (request, flag, status, ierr)
   if (.not. flag) goto 10
```

Asynchronous Send and Receive

- **Write a parallel program to send and receive data using MPI_Isend and MPI_Irecv**
  - Initialize MPI
  - Have processor 0 send an integer to processor 1
  - Have processor 1 receive and integer from processor 0
  - Both processors check on message completion
  - Quit MPI

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

/*************************************************************
This is a simple isend/ireceive program in MPI
************************************************************/

int main(argc,argv)
int argc;
char *argv[];
{
    int myid, numprocs;
    int tag,source,destination,count;
    int buffer;
    MPI_Status status;
    MPI_Request request;
```

```
MPI_Init(&argc,&argv);
MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
MPI_Comm_rank(MPI_COMM_WORLD,&myid);
tag=1234;
source=0;
destination=1;
count=1;
request=MPI_REQUEST_NULL;
```

```
  if(myid == source){
      buffer=5678;
      MPI_Isend(&buffer,count,MPI_INT,destination,tag,
                MPI_COMM_WORLD,&request);
    }
  if(myid == destination){
      MPI_Irecv(&buffer,count,MPI_INT,source,tag,
                 MPI_COMM_WORLD,&request);
    }
  MPI_Wait(&request,&status);
  if(myid == source){
      printf("processor %d  sent %d\n",myid,buffer);
    }
  if(myid == destination){
      printf("processor %d  got %d\n",myid,buffer);
    }
    MPI_Finalize();
}
```

Broadcast Operation: MPI_Bcast

- All nodes call MPI_Bcast
- One node (root) sends a message all others receive the message
- C
  - `MPI_Bcast(&buffer, count, datatype, root, communicator);`
- Fortran
  - `call MPI_Bcast(buffer, count, datatype, root, communicator, ierr)`
- Root is node that sends the message

Broadcast Example

- **Write a parallel program to broadcast data using MPI_Bcast**
  - Initialize MPI
  - Have processor 0 broadcast an integer
  - Have all processors print the data
  - Quit MPI

```
/*****************************************************************
This is a simple broadcast program in MPI
*****************************************************************/

int main(argc,argv)
int argc;
char *argv[];
{
    int i,myid, numprocs;
    int source,count;
    int buffer[4];
    MPI_Status status;
    MPI_Request request;

    MPI_Init(&argc,&argv);
    MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
    MPI_Comm_rank(MPI_COMM_WORLD,&myid);
```

```
source=0;
    count=4;
    if(myid == source){
      for(i=0;i<count;i++)
        buffer[i]=i;
    }

   MPI_Bcast(buffer,count,MPI_INT,source,MPI_COMM_WORLD)
   ;
    for(i=0;i<count;i++)
      printf("%d ",buffer[i]);
    printf("\n");
    MPI_Finalize();
}
```

Reduction Operations

- Used to combine partial results from all processors
- Result returned to root processor
- Several types of operations available
- Works on single elements and arrays

MPI_Reduce

- **C**
  - `int MPI_Reduce(&sendbuf, &recvbuf, count, datatype, operation,root, communicator)`
- **Fortran**
  - `call MPI_Reduce(sendbuf, recvbuf, count, datatype, operation,root, communicator, ierr)`
- **Parameters**
  - Like MPI_Bcast, a root is specified.
  - Operation is a type of mathematical operation

Operations for MPI_Reduce

MPI_MAX	Maximum
MPI_MIN	Minimum
MPI_PROD	Product
MPI_SUM	Sum
MPI_LAND	Logical and
MPI_LOR	Logical or
MPI_LXOR	Logical exclusive or
MPI_BAND	Bitwise and
MPI_BOR	Bitwise or
MPI_BXOR	Bitwise exclusive or
MPI_MAXLOC	Maximum value and location
MPI_MINLOC	Minimum value and location

Global Sum with MPI_Reduce

C

```
double sum_partial, sum_global;
sum_partial = ...;
ierr = MPI_Reduce(&sum_partial, &sum_global,
                  1, MPI_DOUBLE_PRECISION,
                  MPI_SUM,root,
                  MPI_COMM_WORLD);
```

Fortran

```
double precision sum_partial, sum_global
sum_partial = ...
call MPI_Reduce(sum_partial, sum_global,
                1, MPI_DOUBLE_PRECISION,
                MPI_SUM,root,
                MPI_COMM_WORLD, ierr)
```

Global Sum Example with MPI_Reduce

- **Write a program to sum data from all processors**

```
#include <mpi.h>
#include <stdio.h>
#include <stdlib.h>

/*
! This program shows how to use MPI_Scatter and
   MPI_Reduce
! Each processor gets different data from the root
   processor
! by way of mpi_scatter.  The data is summed and then
   sent back
! to the root processor using MPI_Reduce.  The root
   processor
! then prints the global sum.
*/
/*  globals */
int numnodes,myid,mpi_err;
#define mpi_root 0
/* end globals  */
```

```
void init_it(int  *argc, char ***argv);

void init_it(int  *argc, char ***argv) {
        mpi_err = MPI_Init(argc,argv);
        mpi_err = MPI_Comm_size( MPI_COMM_WORLD, &numnodes );
        mpi_err = MPI_Comm_rank(MPI_COMM_WORLD, &myid);
}

int main(int argc,char *argv[]){
        int *myray,*send_ray,*back_ray;
        int count;
        int size,mysize,i,k,j,total,gtotal;

        init_it(&argc,&argv);
/* each processor will get count elements from the root */
        count=4;
        myray=(int*)malloc(count*sizeof(int));
```

```
/* create the data to be sent on the root */
        if(myid == mpi_root){
            size=count*numnodes;
                send_ray=(int*)malloc(size*sizeof(int));
                back_ray=(int*)malloc(numnodes*sizeof(int));
                for(i=0;i<size;i++)
                        send_ray[i]=i;
        }
/* send different data to each processor */
        mpi_err = MPI_Scatter(send_ray, count, MPI_INT, myray,
   count,
                                MPI_INT, mpi_root,
   MPI_COMM_WORLD);
/* each processor does a local sum */
        total=0;
        for(i=0;i<count;i++)
            total=total+myray[i];
        printf("myid= %d total= %d\n ",myid,total);
```

```
/* send the local sums back to the root */
    mpi_err = MPI_Reduce(&total, &gtotal, 1, MPI_INT,
   MPI_SUM,
                         mpi_root, MPI_COMM_WORLD);
/* the root prints the global sum */
        if(myid == mpi_root){
          printf("results from all processors= %d \n
   ",gtotal);
        }
    mpi_err = MPI_Finalize();
}
```