MPI Tutorial (part 1)

Patrizio Dazzi
ISTI - CNR
Pisa Research Campus
mail: patrizio.dazzi@isti.cnr.it

Master Degree (Laurea Magistrale) in
Computer Science and Networking
Academic Year 2009-2010

What is MPI ?

- **M P I = Message Passing Interface**
- **Specification for developers of message passing libraries**
  - *hence, it is NOT a library - but rather the specification of what such a library should be.*
    - *specifications have been defined for C/C++ and Fortran.*
- **Two main versions**
  - *MPI-1: final version of draft released in May, 1994*
  - *MPI-2: was finalized in 1996.*
- **MPI implementations are a combination of MPI-1 and MPI-2**

Programming Model

- MPI lends itself to virtually any distributed memory parallel programming model.
- Commonly used to implement behind the scenes some shared memory models, (e.g. Data Parallel), on distributed memory architectures.
- Parallelism is explicit: the programmer is responsible for identifying parallelism and implementing parallel algorithms using MPI constructs.
- The number of tasks dedicated to run a parallel program is static. New tasks can not be dynamically spawned during run time. (MPI-2 addresses this issue).

Program Header and MPI calls structure

- **Header**

```
#include "mpi.h"
```

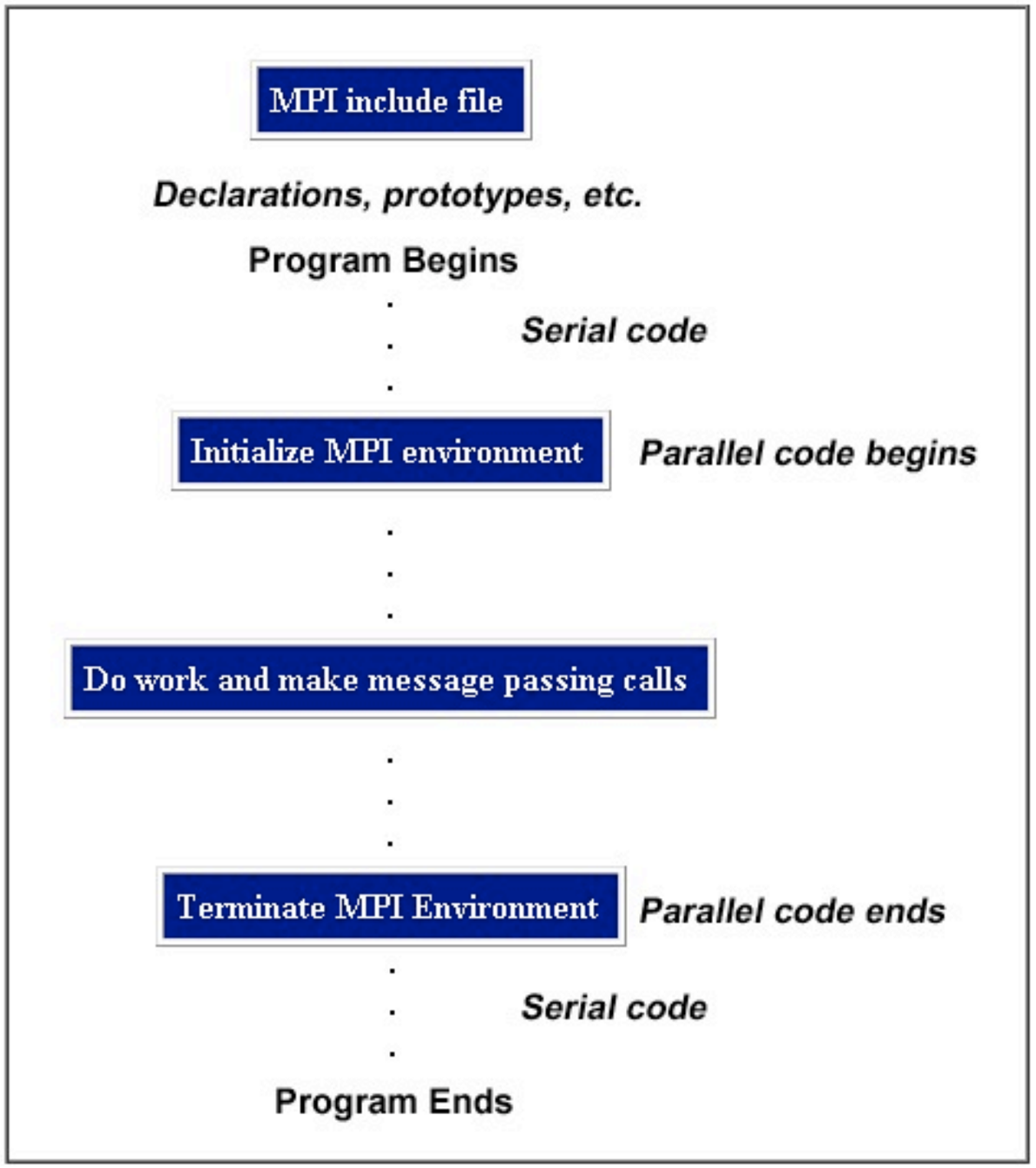
- **Format of MPI Calls:**

  ret_code = MPI_Xxxxx(parameter, ...)

  - *E.g. ret_code = MPI_Send(&buf, count, type, dest, tag, comm);*
  - *ret_code equals to MPI_SUCCESS when successful*

MPI program structure

- **Include file**
- **Environment initialization**
- **MPI Exploitation**
- **Environment termination**



Communicator, Group, Rank (1)

- **Passing Messages means “communicate”**

- **MPI communication are based on three concepts:**
  - *communicator*
  - *group*
  - *rank*

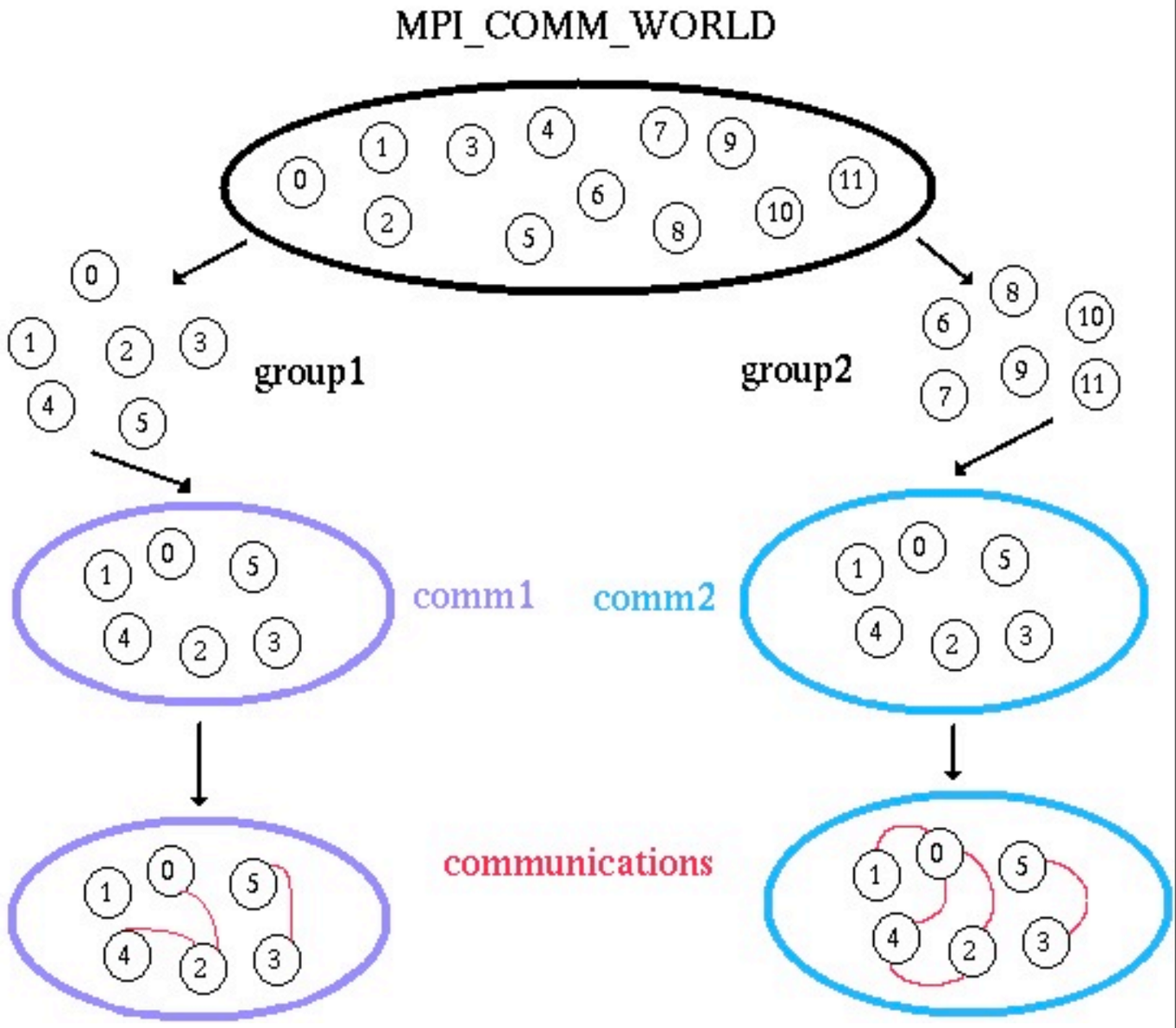
Communicator, Group, Rank (2)

- **“communicators” and “groups” define which collection of processes may communicate with each other.**
  - *a communicator is passed as an argument to several MPI routines*
  - *E.g. MPI_COMM_WORLD is the predefined communicator that includes all the MPI processes*

- **Within a communicator, every MPI process has its own unique identifier: the Rank**
  - *it is assigned by the system when the process initializes.*
  - *rank is sometimes called “task ID”*
  - *ranks are contiguous and begin at zero*
  - *used to specify the source and destination of messages or to control program execution (e.g. if rank=0 do this / if rank=1 do that)*

Communicator vs. Groups

- A group is an ordered set of processes.
- Rank values between zero and N-1.
  N: number of processes in the group.
- A group is associated with a communicator and accessible to the programmer only by a "handle".
- A communicator encompasses a group of processes that may communicate with each other.
- From the programmer's perspective, a group and a communicator are one. Groups routines specifies the processes used to build a communicator.



Environment Management Routines (1)

- **The need of defining a MPI environment**
- **used for several different purposes, such as:**
  - *initializing and terminating the MPI environment*
  - *querying the environment and identity*
  - *etc.*

Environment Management Routines (2)

- **MPI_Init (Prototype: MPI_Init (&argc, &argv))**
  - *Initializes the MPI execution environment. It must be:*
    - called in every MPI program
    - called before any other MPI functions
    - called only once in an MPI program.
- **MPI_Comm_size (Prototype: MPI_Comm_size (comm, &size))**
  - *Determines the number of processes in the group associated with a communicator.*
  - *used within the communicator MPI_COMM_WORLD to determine the total number of application processes.*

Environment Management Routines (3)

- **MPI_Comm_rank (Proto: MPI_Comm_rank (comm, &rank))**
  - *Determines the rank of the calling process within the communicator.*
  - *Each process will be assigned a unique integer rank between 0 and number of processors - 1*
- **MPI_Abort (Prototype: MPI_Abort (comm, errorcode))**
  - *Terminates all MPI processes associated with the communicator.*
  - *In most MPI implementations it terminates ALL processes regardless of the communicator specified.*

Environment Management Routines (4)

- **MPI_Get_processor_name**
  (Prototype: MPI_Get_processor_name (&name, &resultlength))
  - *Returns the processor name and the name length.*
  - *Buffer for name must be at least MPI_MAX_PROCESSOR_NAME characters in size*
  - *What is returned into "name" is implementation dependent*
- **MPI_Initialized (Prototype: MPI_Initialized (&flag))**
  - *Returns true if MPI_Init has been called, false otherwise*
  - *Useful because MPI requires that MPI_Init be called once and only once by each process.*

Environment Management Routines (5)

- **MPI_Wtime (Prototype: MPI_Wtime ())**
  - *Elapsed wall clock time in seconds on the calling processor.*

- **MPI_Wtick (Prototype: MPI_Wtick ())**
  - *Returns the resolution in seconds of MPI_Wtime.*

- **MPI_Finalize (Prototype: MPI_Finalize ())**
  - *Terminates the MPI execution environment.*
  - *should be the last MPI routine called in every MPI program*

Environment Management Routines - Example 1


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

int main(int argc, char** argv){

   int  numtasks, rank, ret_code;

   ret_code = MPI_Init(&argc,&argv);

   if (ret_code != MPI_SUCCESS) {
     printf ("Error starting MPI program. Terminating.\n");
     MPI_Abort(MPI_COMM_WORLD, ret_code);
   }

   MPI_Comm_size(MPI_COMM_WORLD, &numtasks);
   MPI_Comm_rank(MPI_COMM_WORLD, &rank);

   printf ("Number of tasks= %d My rank= %d\n", numtasks, rank);

   /*******  do some work *******/

   MPI_Finalize();

}
```

Environment Management Routines - Example 2


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

int main(argc,argv) {
    int numtasks, rank, dest, source, ret_code, 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) {
       printf (“I am the process with rank 0”);
    }
    else if (rank == 1) {
       printf (“I am the process with rank 1”);
    }

    ret_code = MPI_Get_count(&Stat, MPI_CHAR, &count);
    printf(“The total number of processes is %d”, count);

    MPI_Finalize();
}
```

Communication Routines

- **Point-to-point**
  - *Blocking*
  - *Non-Blocking*

- **Collective communications**
  - *Synchronization*
  - *Data Movement*
  - *Collective Computation*

Point to Point Communication Routines (1)

- **MPI point-to-point operations typically involve message passing between two different MPI tasks.**
  - *One performs a send operation, the other performs a matching receive operation*
- **There are different types of send and receive routines used for different purposes, for instance:**
  - *Synchronous send*
  - *Blocking send / blocking receive*
  - *Non-blocking send / non-blocking receive*
  - *Buffered send*
  - *Combined send/receive*
  - *"Ready" send*

Point to Point Communication Routines (2)

- **Any type of send routine can be paired with any type of receive routine**
- **MPI also provides several routines associated with send - receive operations,**
  - *e.g. those used to wait for a message's arrival or probe to find out if a message has arrived.*
- **When a send is not synchronized with its matching receive, the MPI implementation must be able to manage the data in transit**
  - *The MPI implementation (not the MPI standard) decides what happens to data in these types of cases. Typically, a system buffer area is reserved to hold data in transit.*

Blocking vs. Non-blocking (1)

- **Most of the MPI point-to-point routines can be used in either blocking or non-blocking mode.**
- **Blocking:**
  - *A blocking send routine will only return after it is safe to modify the application send buffer for reuse, i.e. modifications will not affect the data intended for the receive task.*
  - *A blocking send can be synchronous which means there is handshaking occurring with the receive task to confirm a safe send.*
  - *A blocking send can be asynchronous if a system buffer is used to hold the data for eventual delivery to the receive.*
  - *A blocking receive only "returns" after the data has arrived and is ready for use by the program.*

Blocking vs. Non-blocking (2)

- **Non-blocking:**
  - *Non-blocking send and receive routines do not wait for any communication events to complete, such as message copying from user memory to system buffer space or the actual arrival of message.*
  - *Non-blocking operations simply request the MPI library to perform the operation when it is able. The user can not predict when that will happen.*
  - *It is unsafe to modify the application comm. buffer until you know for a fact the requested non-blocking operation was actually performed by the library. There are "wait" routines used to do this.*
  - *Non-blocking communications are primarily used to overlap computation with communication and exploit possible performance gains.*

Order and Fairness

- **Order**
  - *MPI guarantees that messages will not overtake each other.*
  - *If a sender sends two messages in succession to the same destination, both matching the same receive, the receive operation will receive Messages in order.*
  - *If a receiver posts two receives in succession both looking for the same message, messages will be received in order.*
  - *NOTE: Order rules do not apply if there are multiple threads participating in the communication operations.*

Fairness

- **MPI does not guarantee fairness**
  - *it's up to the programmer to prevent "operation starvation"*
  - *Example:*
    - task 0 sends a message to task 2.
    - However, task 1 sends a competing message that matches task 2's receive.
    - Only one of the sends will complete.

MPI Message Passing Routine Arguments (1)

- MPI point-to-point communication routines generally have an argument list that takes one of the following formats:

Blocking sends	`MPI_Send(buffer,count,type,dest,tag,comm)`
Non-blocking sends	`MPI_Isend(buffer,count,type,dest,tag,comm,request)`
Blocking receive	`MPI_Recv(buffer,count,type,source,tag,comm,status)`
Non-blocking receive	`MPI_Irecv(buffer,count,type,source,tag,comm,request)`

MPI Message Passing Routine Arguments (2)

- **Buffer**
  - *Program address space referencing the data that is to be sent or received. In most cases, this is simply the variable name that is be sent/received.*
- **Data Count**
  - *Indicates the number of data elements of a particular type to be sent.*
- **Data Type**
  - *For reasons of portability, MPI predefines its elementary data types.*
  - *But programmers may also create their own data types*

MPI Message Passing Routine Arguments (3)

MPI Datatypes

C Data 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	8 binary digits
MPI_PACKED	data packed or unpacked with MPI_Pack()/ MPI_Unpack

MPI Message Passing Routine Arguments (4)

- **Destination**
  - *indicates the process where a message should be delivered. Specified as the rank of the receiving process.*
- **Source**
  - *indicates the originating process of the message. Specified as the rank of the sending process. This may be set to the wild card MPI_ANY_SOURCE to receive a message from any task.*
- **Tag**
  - *Arbitrary non-negative integer assigned by the programmer to uniquely identify a message. Send and receive operations should match message tags.*
  - *For a receive operation, the wild card MPI_ANY_TAG can be used to receive any message regardless of its tag.*

MPI Message Passing Routine Arguments (5)

- **Communicator**
  - *Indicates the communication context, or set of processes for which the source or destination fields are valid*
- **Status**
  - *For a receive operation, indicates the source of the message and the tag of the message. This argument is a pointer to a predefined structure MPI_Status.*
  - *The actual number of bytes received are obtainable from Status via the MPI_Get_count routine.*
- **Request**
  - *Used by non-blocking send and receive operations. Since non-blocking operations may return before the requested system buffer space is obtained, the system issues a unique "request number". The programmer uses this system assigned "handle" later to determine completion of the non-blocking operation.*

Blocking Message Passing Routines (1)

- **MPI_Send (P: MPI_Send(&buf,count,datatype,dest,tag,comm))**
  - *represents the basic blocking send operation.*
  - *returns only after the application buffer in the sending task is free*
  - *this routine may be implemented differently on different systems.*

- **MPI_Recv (P: MPI_Recv(&buf,count,datatype,source,tag,comm,&status))**
  - *simplest operation for receiving a message*
  - *it blocks until the requested data is available in the application buffer in the receiving task*

Blocking Message Passing Routines (1) - Example


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

int main(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("%d: Received %d char(s) from %d with tag %d\n", rank, count, Stat.MPI_SOURCE,
    Stat.MPI_TAG);
    MPI_Finalize();
}
```

Blocking Message Passing Routines (2)

- **MPI_Ssend (MPI_Ssend (&buf,count,datatype,dest,tag,comm))**
  - *Synchronous blocking send: Send a message and block until the application buffer in the sending task is free and the destination process has started to receive the message.*
- **MPI_Bsend (MPI_Bsend (&buf,count,datatype,dest,tag,comm))**
  - *Buffered blocking send:*
    - permits the programmer to allocate the required amount of buffer space into which data can be copied until it is delivered.
    - Routine returns after the data has been copied from application buffer space to the allocated send buffer.
    - Make sure you have enough buffer space available.
    - Must be used with the MPI_Buffer_attach routine.

Blocking Message Passing Routines (3)

- **MPI_Buffer_attach (MPI_Buffer_attach (&buffer,size))**
  MPI_Buffer_detach (MPI_Buffer_detach (&buffer,size))
  - *Used by programmer to allocate/deallocate message buffer space to be used by the MPI_Bsend routine.*
  - *The size argument is specified in bytes - not in "elements".*
  - *Only one buffer can be attached to a process at a time.*

- **MPI_Rsend (MPI_Rsend(&buf,count,datatype,dest,tag,comm))**
  - *Blocking ready send.*
  - *Should only be used if the programmer is certain that the matching receive has already been posted.*

Blocking Message Passing Routines (2-3) - Example (1)


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

int main(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;
  printf("approaching send operation\n");
  rc = MPI_Ssend(&outmsg, 1, MPI_CHAR, dest, tag, MPI_COMM_WORLD);
  printf("data sent\n");
  rc = MPI_Recv(&inmsg, 1, MPI_CHAR, source, tag, MPI_COMM_WORLD, &Stat);

} else if (rank == 1) {
  dest = 0;
  source = 0;
  sleep(5);
  rc = MPI_Recv(&inmsg, 1, MPI_CHAR, source, tag, MPI_COMM_WORLD, &Stat);
  rc = MPI_Ssend(&outmsg, 1, MPI_CHAR, dest, tag, MPI_COMM_WORLD);
}
rc = MPI_Get_count(&Stat, MPI_CHAR, &count);
printf("%d: Received %d char(s) from %d with tag %d\n", rank, count, Stat.MPI_SOURCE, Stat.MPI_TAG);
MPI_Finalize();
}
```

Blocking Message Passing Routines (2-3) - Example (2)


```c
#include "mpi.h"
#include <stdio.h>
#include <unistd.h>

int main(int argc, char** argv) {
int numtasks, rank, dest, source, rc, count, tag=1; int buffer[20]; 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);
MPI_Buffer_attach (&buffer, 20*sizeof(int));

if (rank == 0) {
  dest = 1;
  source = 1;
  printf("approaching send operation\n");
  rc = MPI_Bsend(&outmsg, 1, MPI_CHAR, dest, tag, MPI_COMM_WORLD);
  printf("data sent\n");
  rc = MPI_Recv(&inmsg, 1, MPI_CHAR, source, tag, MPI_COMM_WORLD, &Stat);

} else if (rank == 1) {
  dest = 0;
  source = 0;
  sleep(5);
  rc = MPI_Recv(&inmsg, 1, MPI_CHAR, source, tag, MPI_COMM_WORLD, &Stat);
  rc = MPI_Ssend(&outmsg, 1, MPI_CHAR, dest, tag, MPI_COMM_WORLD);
}
rc = MPI_Get_count(&Stat, MPI_CHAR, &count);
printf("%d: Received %d char(s) from %d with tag %d\n", rank, count, Stat.MPI_SOURCE, Stat.MPI_TAG);
MPI_Finalize();
}
```

Blocking Message Passing Routines (4)

- **MPI_Sendrecv**
  MPI_Sendrecv (&sendbuf, sendcount, sendtype, dest, sendtag, &recvbuf, recvcount, recvtype, source, recvtag, comm, &status)
  - *Send a message and post a receive before blocking.*
  - *Will block until*
    - the sending application buffer is free and
    - until the receiving application buffer contains the received message.

Blocking Message Passing Routines (4) - Example


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

int main(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_Sendrecv (&outmsg, 1, MPI_CHAR, dest, tag, &inmsg, 1, MPI_CHAR, source, tag, MPI_COMM_WORLD, &Stat);

} else if (rank == 1) {
  dest = 0;
  source = 0;
  rc = MPI_Sendrecv (&outmsg, 1, MPI_CHAR, dest, tag, &inmsg, 1, MPI_CHAR, source, tag, MPI_COMM_WORLD, &Stat);
}

rc = MPI_Get_count(&Stat, MPI_CHAR, &count);
printf("%d: Received %d char(s) from %d with tag %d\n", rank, count, Stat.MPI_SOURCE, Stat.MPI_TAG);
MPI_Finalize();
}
```

Blocking Message Passing Routines (5)

- **MPI_Wait (MPI_Wait (&request, &status))**
  MPI_Waitany (MPI_Waitany (count, &array_of_requests, &index, &status))
  MPI_Waitall (MPI_Waitall (count, &array_of_requests, &array_of_statuses))
  MPI_Waitsome (MPI_Waitsome(incount, &array_of_requests, &outcount, &array_of_offsets, &array_of_statuses))
  - *MPI_Wait blocks until a specified non-blocking send or receive operation has completed.*
  - *For multiple non-blocking operations, the programmer can specify any, all or some completions.*

- **MPI_Probe (MPI_Probe (source,tag,comm,&status))**
  - *Performs a blocking test for a message.*
  - *The "wildcards" MPI_ANY_SOURCE and MPI_ANY_TAG may be used to test for a message from any source or with any tag.*

Non-Blocking Message Passing Routines (1)

- **MPI_Isend (MPI_Isend (&buf,count,datatype,dest,tag,comm,&request))**
  - *Processing continues immediately without waiting for the message to be copied out from the application buffer.*
  - *Returns a request handle for handling the pending message status.*
  - *The program should not modify the buffer until subsequent calls to MPI_Wait or MPI_Test indicate that the Isend has completed.*

Non-Blocking Message Passing Routines (2)

- **MPI_Irecv (MPI_Irecv (&buf,count,datatype,source,tag,comm,&request))**
  - *Processing continues immediately without actually waiting for the message to be received and copied into the the application buffer.*
  - *Returns a request handle for handling the pending message status.*
  - *The program must use calls to MPI_Wait or MPI_Test to determine when the Irecv completes and the requested message is available in the application buffer.*

Non-Blocking Message Passing Routines (3)

- **MPI_Issend (MPI_Issend (&buf,count,datatype,dest,tag,comm,&request))**
  MPI_Ibsend (MPI_Ibsend (&buf,count,datatype,dest,tag,comm,&request))
  MPI_Irsend (MPI_Irsend (&buf,count,datatype,dest,tag,comm,&request))

- **Non-blocking:**

  - *synchronous send. Similar to MPI_Isend(), except MPI_Wait() or MPI_Test() indicates when the destination process has received the message.*

  - *buffered send. Similar to MPI_Bsend() except MPI_Wait() or MPI_Test() indicates when the destination process has received the message. Must be used with the MPI_Buffer_attach routine.*

  - *ready send. Similar to MPI_Rsend() except MPI_Wait() or MPI_Test() indicates when the destination process has received the message.*

    - Remember: Should only be used if the programmer is certain that the matching receive has already been posted.

Non-Blocking Message Passing Routines (4)

- **MPI_Test (MPI_Test (&request, &flag, &status))**
  MPI_Testany (MPI_Testany (count, &array_of_requests, &index, &flag, &status))
  MPI_Testall (MPI_Testall (count, &array_of_requests, &flag, &array_of_statuses))
  MPI_Testsome (MPI_Testsome (incount, &array_of_requests, &outcount, &array_of_offsets, &array_of_statuses))
  - *MPI_Test checks the status of a specified non-blocking send or receive operation.*
  - *"flag" parameter is returned logical true (1) if the operation has completed, and logical false (0) if not.*
  - *For multiple non-blocking operations, the programmer can specify any, all or some completions.*

Non-Blocking Message Passing Routines (5)

- **MPI_Iprobe (MPI_Iprobe (source, tag, comm, &flag, &status))**
  - *Performs a non-blocking test for a message.*
  - *The "wildcards" MPI_ANY_SOURCE and MPI_ANY_TAG may be used to test for a message from any source or with any tag.*
  - *"flag" parameter is returned logical true (1) if a message has arrived, and logical false (0) if not.*
  - *Source and tag will be returned in the status structure as status.MPI_SOURCE and status.MPI_TAG.*

Non-Blocking Message Passing Routines - Example


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

int main(int argc, char** argv){
    int numtasks, rank, next, prev, buf[2], tag1=1, tag2=2;
    MPI_Request reqs[4];
    MPI_Status stats[4];

    MPI_Init(&argc,&argv);
    MPI_Comm_size(MPI_COMM_WORLD, &numtasks);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);

    prev = rank-1;
    next = rank+1;
    if (rank == 0)  prev = numtasks - 1;
    if (rank == (numtasks - 1))  next = 0;

    MPI_Irecv(&buf[0], 1, MPI_INT, prev, tag1, MPI_COMM_WORLD, &reqs[0]);
    MPI_Irecv(&buf[1], 1, MPI_INT, next, tag2, MPI_COMM_WORLD, &reqs[1]);

    MPI_Isend(&rank, 1, MPI_INT, prev, tag2, MPI_COMM_WORLD, &reqs[2]);
    MPI_Isend(&rank, 1, MPI_INT, next, tag1, MPI_COMM_WORLD, &reqs[3]);

    printf("%d: received %d and %d\n", rank, buf[0], buf[1]);

    MPI_Waitall(4, reqs, stats);
    MPI_Finalize();
}
```

Questions ?