Nicola Spallanzani
n.spallanzani@cineca.it
High Performance Computing department

Parallel programming with MPI

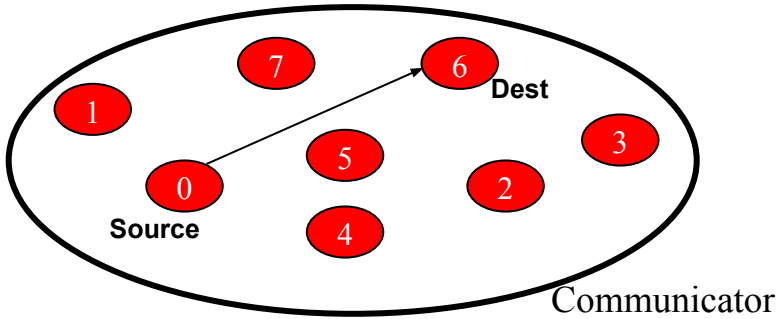
Point-to-Point Communications

Contents

- Send and Receive function calls for point-to-point communications
- Blocking and non-blocking
- How to avoid deadlocks

Point-to-Point Communication

- It is the **basic** communication method provided by MPI library. Communication between 2 processes
- It is conceptually simple: source process A sends a message to destination process B, B receive the message from A.
- Communication take places within a communicator
- Source and Destination are identified by their rank in the communicator



Point-to-Point communication quick example

```
...
CALL MPI_COMM_RANK(MPI_COMM_WORLD, myid, ierr)
IF( myid .EQ. 0 ) THEN
    CALL MPI_SEND(a, 2, MPI_REAL, 1, 10, MPI_COMM_WORLD, ierr)
ELSE IF( myid .EQ. 1 ) THEN
  CALL MPI_RECV(b, 2, MPI_REAL, 0, 10, MPI_COMM_WORLD, status, ierr)
END IF
...
```

Point-to-Point communication quick example

The construction

```
if rank equals i
    send information
else if rank equals j
    receive information
```

is very common in MPI programs. Often one rank (usually rank 0) is selected for particular tasks which can be or should be done by one task only such as reading or writing files, giving messages to the user or for managing the overall logic of the program (e.g. master-slave).

The Message

- Buffers of data are exchanged. They are series of count elements of a particular MPI data type
- In this way MPI knows how many bytes (length) has to send/receive.
- This allows MPI programs to run in heterogeneous environments
- C types are different from Fortran types.

Messages are identified by their envelopes. A message could be exchanged only if the sender and receiver specify the correct envelope

Message Structure

envelope				body		
source	destination	communicator	tag	buffer	count	datatype

Data Types

- MPI Data types
  - Basic types (portability)
  - Derived types (MPI_Type_xxx functions)
- Derived type can be built up from basic types
- User-defined data types allows MPI to use non-contiguous buffers of data
- MPI defines *'handles'* to allow programmers to refer to data types
  - declare the **right type** of the handles as defined in the API of language (C/Fortran)
  - MPI data type and language data type have to be **compatible** (MPI_INTEGER and INTEGER)

Fortran - MPI Intrinsic Datatypes

MPI Data type	Fortran Data type
MPI_INTEGER	INTEGER
MPI_REAL	REAL
MPI_DOUBLE_PRECISION	DOUBLE PRECISION
MPI_COMPLEX	COMPLEX
MPI_DOUBLE_COMPLEX	DOUBLE COMPLEX
MPI_LOGICAL	LOGICAL
MPI_CHARACTER	CHARACTER(1)
MPI_PACKED	
MPI_BYTE	

C - MPI Intrinsic Datatypes

MPI Data type	C Data type
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	

For a communication to succeed ...

1. Sender must specify a valid destination rank.
2. Receiver must specify a valid source rank.
3. The communicator must be the same.
4. Tags must match.
5. Buffers must be large enough.

Completion

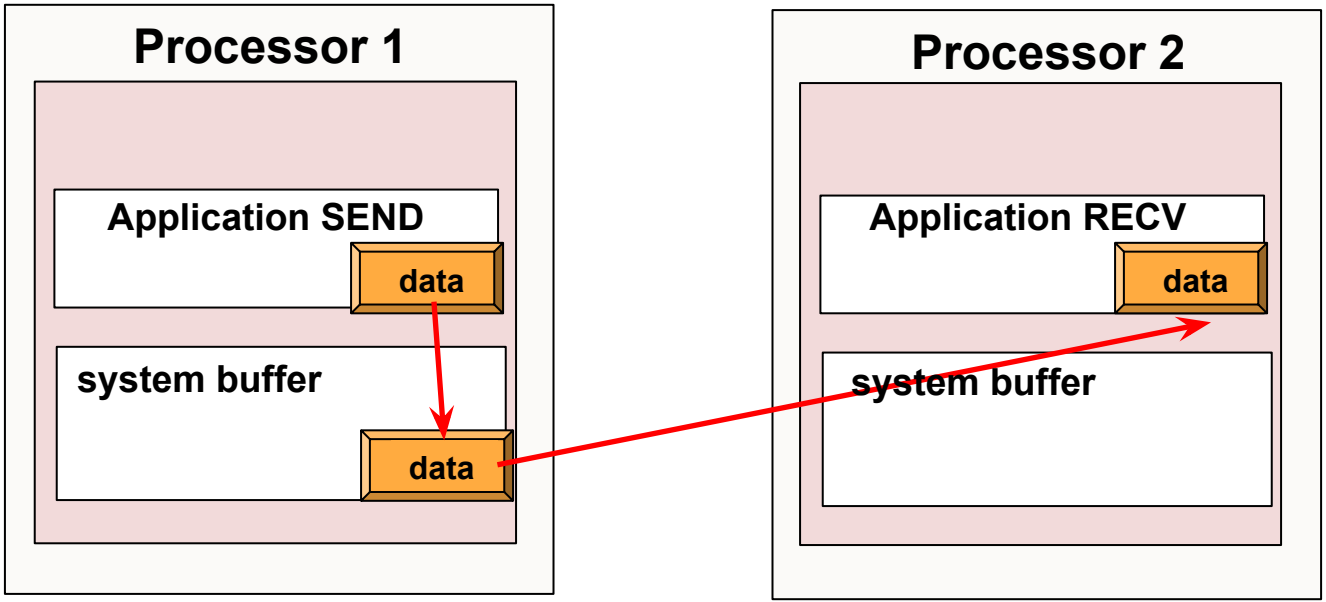
- In a perfect world, every send operation would be **perfectly synchronized** with its matching receive. This is rarely the case. The MPI implementation is able to deal with the exchanging data when the two tasks are out of sync.

- **Completion** of the communication means that memory locations used in the message transfer can be safely accessed
  - Send: variable sent can be reused after completion
  - Receive: variable received can be used after completion

Blocking communications

- Most of the MPI point-to-point routines can be used in either blocking or non-blocking mode.
- Blocking:
  - A blocking send returns after it is safe to modify the sent buffer. Safe does not imply that the data was actually received - it may be stored in a system buffer.
  - A blocking send can be synchronous.
  - A blocking send can be asynchronous if a system buffer is used to hold the data for eventual delivery.
  - A blocking receive only "returns" after the data has arrived and is ready for use by the program.

Blocking Communications



Standard Send and Receive

C:

```
int MPI_Send(void *buf, int count, MPI_Datatype type,
    int dest, int tag, MPI_Comm comm);
```

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

Standard Send and Receive

Basic blocking point-to-point communication routine in MPI.

Fortran:

```
MPI_SEND(buf, count, type, dest, tag, comm, ierr)
MPI_RECV(buf, count, type, source, tag, comm, status, ierr)
```

Message body **Message envelope**

```
buf          send/receive buffer (memory located in sender/receiver memory).
count    (INTEGER)number of contiguous elements of buf to be sent
type     (INTEGER)MPI type of buf (implicitly number of bytes)
dest     (INTEGER)rank of the destination process
tag          (INTEGER)number identifying the message
comm     (INTEGER)communicator of the sender and receiver
status   (INTEGER)array of size MPI_STATUS_SIZE containing
             communication status information (Orig Rank, Tag, Number of elements
    received)
ierr     (INTEGER) error code (if ierr=0 no error occurs)
```

Send and Receive - FORTRAN

```
PROGRAM send_recv

      USE mpi
      implicit none

      INTEGER :: ierr, myid, nproc
      INTEGER :: status(MPI_STATUS_SIZE)
      REAL :: A(2)

      CALL MPI_INIT(ierr)
      CALL MPI_COMM_SIZE(MPI_COMM_WORLD, nproc, ierr)
      CALL MPI_COMM_RANK(MPI_COMM_WORLD, myid, ierr)

      IF( myid .EQ. 0 ) THEN
            A(1) = 3.0
            A(2) = 5.0
            CALL MPI_SEND(A, 2, MPI_REAL, 1, 10, MPI_COMM_WORLD, ierr)
      ELSE IF( myid .EQ. 1 ) THEN
            CALL MPI_RECV(A, 2, MPI_REAL, 0, 10, MPI_COMM_WORLD, status, ierr)
            WRITE(6,*) myid,': a(1)=',a(1),' a(2)=',a(2)
      END IF

      CALL MPI_FINALIZE(ierr)
END PROGRAM
```

Send and Receive - C

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

void main (int argc, char * argv[])
{
  int err, nproc, myid;
  MPI_Status status;
  float a[2];

  err = MPI_Init(&argc, &argv);
  err = MPI_Comm_size(MPI_COMM_WORLD, &nproc);
  err = MPI_Comm_rank(MPI_COMM_WORLD, &myid);

  if( myid == 0 ) {
    a[0] = 3.0, a[1] = 5.0;
    MPI_Send(a, 2, MPI_FLOAT, 1, 10, MPI_COMM_WORLD);
  } else if( myid == 1 ) {
    MPI_Recv(a, 2, MPI_FLOAT, 0, 10, MPI_COMM_WORLD, &status);
    printf("%d: a[0]=%f a[1]=%f\n", myid, a[0], a[1]);
  }

  err = MPI_Finalize();
}
```

Non Blocking communications

- Non-blocking send and receive routines will return almost immediately. They do not wait for any communication events to complete
- Non-blocking operations simply "request" the MPI library to perform the operation when it is possible. The user can not predict when that will happen.
- It is unsafe to modify the application buffer until you know for a fact that 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.
- Another important use is to overlap communication with communication, i.e. avoiding serialization issues (for example, when a loop of communications is involved)

Non-Blocking Send and Receive

C:

```
int MPI_Isend(void *buf, int count, MPI_Datatype type,
    int dest, int tag, MPI_Comm comm, MPI_Request *req);

int MPI_Irecv (void *buf, int count, MPI_Datatype type,
    int source, int tag, MPI_Comm comm, MPI_Request *req);
```

Non-Blocking Send and Receive

FORTRAN:

```
MPI_ISEND(buf, count, type, dest, tag, comm, req, ierr)
MPI_IRECV(buf, count, type, source, tag, comm, req, ierr)
```

buf send/receive buffer (memory located in sender/receiver memory).
count (INTEGER) number of contiguous elements of **buf** to be sent
type (INTEGER) MPI type of **buf**
dest (INTEGER) rank of the destination process
tag (INTEGER) number identifying the message
comm (INTEGER) communicator of the sender and receiver
req (INTEGER) output, identifier of the communications handle
ierr (INTEGER) output, error code (if **ierr=0** no error occurs)

Waiting for Completion

FORTRAN:

```
MPI_WAIT(req, status, ierr)
MPI_WAITALL (count, array_of_requests, array_of_statuses, ierr)
```

A call to this subroutine causes the code to wait until the communication referred to by req is complete.

req (INTEGER): input/output, identifier associated to a communications event (initiated by **MPI_ISEND** or **MPI_IRECV**).

Status (INTEGER): array of size **MPI_STATUS_SIZE**, if **req** was associated to a call to **MPI_IRECV**, **status** contains informations on the received message, otherwise **status** could contain an error code.

ierr (INTEGER): output, error code (if **ierr=0** no error occurs).

C:

```
int MPI_Wait(MPI_Request *req, MPI_Status *status);
Int MPI_Waitall (count,&array_of_requests,&array_of_statuses) ;
```

Testing Completion

FORTRAN:

```
MPI_TEST(req, flag, status, ierr)
MPI_TESTALL (count,array_of_requests,flag,array_of_statuses,ierr)
```

A call to this subroutine sets **flag** to **.true.** if the communication pointed by **req** is complete, sets **flag** to **.false.** otherwise.

Req (INTEGER) input/output, identifier associated to a communications event (initiated by **MPI_ISEND** or **MPI_IRECV**).

Flag (LOGICAL) output, **.true.** if communication **req** has completed **.false.** otherwise

Status (INTEGER)array of size MPI_STATUS_SIZE, if req was associated to a call to **MPI_IRECV**, **status** contains informations on the received message, otherwise **status** could contain an error code.

Ierr (INTEGER) output, error code (if **ierr=0** no error occurs).

C:

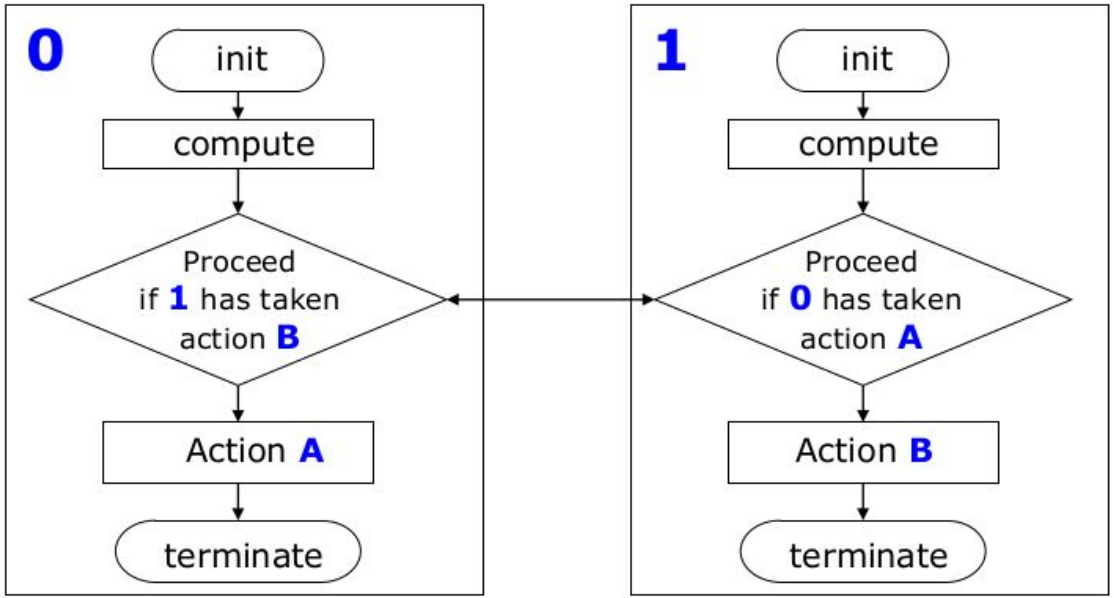
```
int MPI_Test (&request,&flag,&status);
Int MPI_Testall (count,&array_of_requests,&flag,&array_of_statuses);
```

Wildcards

- Both in FORTRAN and C **MPI_RECV** accepts wildcards:
- To receive from any source: MPI_ANY_SOURCE
- To receive with any tag: MPI_ANY_TAG
- Actual source and tag are returned in the receiver's status parameter

DEADLOCK

A Deadlock occurs when 2 (or more) processes are blocked and each one is waiting for the other to make progress.



Simple DEADLOCK

```
PROGRAM deadlock
      USE mpi
      implicit none
      INTEGER :: ierr, myid, nproc
      INTEGER :: status(MPI_STATUS_SIZE)
      REAL :: A(2), B(2)

      CALL MPI_INIT(ierr)
      CALL MPI_COMM_SIZE(MPI_COMM_WORLD, nproc, ierr)
      CALL MPI_COMM_RANK(MPI_COMM_WORLD, myid, ierr)

      IF( myid .EQ. 0 ) THEN
            a(1) = 2.0
            a(2) = 4.0
            CALL MPI_SEND(a, 2, MPI_REAL, 1, 10, MPI_COMM_WORLD, ierr)
            CALL MPI_RECV(b, 2, MPI_REAL, 1, 11, MPI_COMM_WORLD, status, ierr)
      ELSE IF( myid .EQ. 1 ) THEN
            a(1) = 3.0
            a(2) = 5.0
            CALL MPI_SEND(a, 2, MPI_REAL, 0, 11, MPI_COMM_WORLD, ierr)
            CALL MPI_RECV(b, 2, MPI_REAL, 0, 10, MPI_COMM_WORLD, status, ierr)
      END IF
      WRITE(6,*) myid, ': b(1)=', b(1), ' b(2)=', b(2)
      CALL MPI_FINALIZE(ierr)
END PROGRAM
```

Avoiding DEADLOCK

```
PROGRAM avoid_lock
      USE mpi
      implicit none
      INTEGER :: ierr, myid, nproc
      INTEGER :: status(MPI_STATUS_SIZE)
      REAL :: A(2), B(2)

      CALL MPI_INIT(ierr)
      CALL MPI_COMM_SIZE(MPI_COMM_WORLD, nproc, ierr)
      CALL MPI_COMM_RANK(MPI_COMM_WORLD, myid, ierr)

      IF( myid .EQ. 0 ) THEN
            a(1) = 2.0
            a(2) = 4.0
            CALL MPI_RECV(b, 2, MPI_REAL, 1, 11, MPI_COMM_WORLD, status, ierr)
            CALL MPI_SEND(a, 2, MPI_REAL, 1, 10, MPI_COMM_WORLD, ierr)
      ELSE IF( myid .EQ. 1 ) THEN
            a(1) = 3.0
            a(2) = 5.0
            CALL MPI_SEND(a, 2, MPI_REAL, 0, 11, MPI_COMM_WORLD, ierr)
            CALL MPI_RECV(b, 2, MPI_REAL, 0, 10, MPI_COMM_WORLD, status, ierr)
      END IF
      WRITE(6,*) myid, ': b(1)=', b(1), ' b(2)=', b(2)
      CALL MPI_FINALIZE(ierr)
END PROGRAM
```

Avoiding DEADLOCK (2)

```
PROGRAM deadlock
      USE mpi
      implicit none
      INTEGER :: ierr, myid, nproc, req
      INTEGER :: status(MPI_STATUS_SIZE)
      REAL :: A(2), B(2)

      CALL MPI_INIT(ierr)
      CALL MPI_COMM_SIZE(MPI_COMM_WORLD, nproc, ierr)
      CALL MPI_COMM_RANK(MPI_COMM_WORLD, myid, ierr)

      IF( myid .EQ. 0 ) THEN
            a = (/ 2.0, 4.0 /)
            CALL MPI_ISEND(a, 2, MPI_REAL, 1, 10, MPI_COMM_WORLD, req, ierr)
            CALL MPI_RECV(b, 2, MPI_REAL, 1, 11, MPI_COMM_WORLD, status, ierr)
            CALL MPI_WAIT(req, status, ierr)
      ELSE IF( myid .EQ. 1 ) THEN
            a = (/ 3.0, 5.0 /)
            CALL MPI_ISEND(a, 2, MPI_REAL, 0, 11, MPI_COMM_WORLD, req, ierr)
            CALL MPI_RECV(b, 2, MPI_REAL, 0, 10, MPI_COMM_WORLD, status, ierr)
            CALL MPI_WAIT(req, status, ierr)
      END IF
      WRITE(6,*) myid, ': b(1)=', b(1), ' b(2)=', b(2)
      CALL MPI_FINALIZE(ierr)
END PROGRAM
```

SendRecv

- Send a message and post a receive before blocking. Will block until the sending application buffer is free for reuse and until the receiving application buffer contains the received message.
- The easiest way to send and receive data without worrying about deadlocks

FORTRAN:

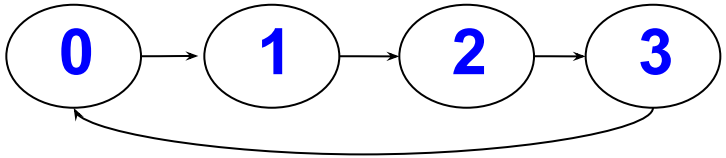
Sender side

```
CALL MPI_SENDRECV(sndbuf, snd_size, snd_type,destid,tag,
rcvbuf, rcv_size, rcv_type, sourceid, tag, comm, status,
ierr)
```

Receiver side

SendRecv example

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

int main(int argc, char *argv[])
{
    int myid, numprocs, left, right;
    int buffer[1], buffer2[1];
    MPI_Status status;

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

    right = (myid + 1) % numprocs;
    left = (myid - 1 + numprocs) % numprocs;

    buffer[0]=myid;
    MPI_Sendrecv(buffer, 1, MPI_INT, right, 123, buffer2, 1, MPI_INT, left, 123,
    MPI_COMM_WORLD, &status);

    printf("I am rank %d and I have received %d\n",myid,buffer2[0]);
    MPI_Finalize();

    return 0;
 }
```



Useful for cyclic communication patterns

SEND and RECV variants

Mode	Completion Condition	Blocking subroutine	Non-blocking subroutine
Standard send	**Message sent (receive state unknown)**	`MPI_SEND`	`MPI_ISEND`
receive	**Completes when a matching message has arrived**	`MPI_RECV`	`MPI_IRECV`
Synchronous send	**Only completes after a matching recv() is posted and the receive operation is started.**	`MPI_SSEND`	`MPI_ISSEND`
Buffered send	**Always completes, irrespective of receiver** **Guarantees the message being buffered**	`MPI_BSEND`	`MPI_IBSEND`
Ready send	**Always completes, irrespective of whether the receive has completed**	`MPI_RSEND`	`MPI_IRSEND`

Final Comments

- MPI is a standard for message-passing and has numerous implementations (OpenMPI, IntelMPI, MPICH, etc)
- MPI uses send and receive calls to manage communications between two processes (point-to-point)
- The calls can be blocking or non-blocking.
- Non-blocking calls can be used to overlap communication with computation (or communication) but wait routines are needed for synchronization.
- Deadlock is a common error and is due to incorrect order of send/receive