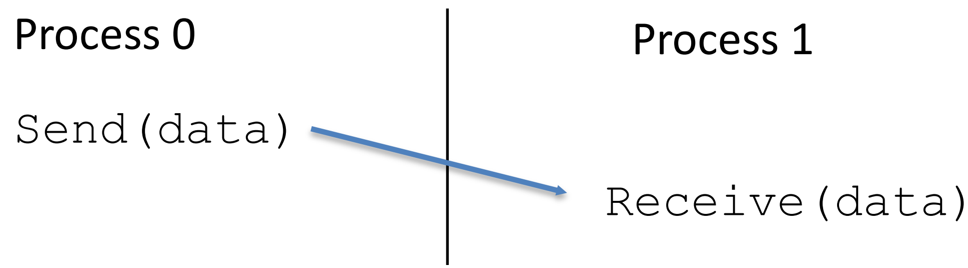
POLITECNICO
MILANO 1863

Message Passing Interface - MPI

Alessandro Margara
alessandro.margara@polimi.it
https://margara.faculty.polimi.it

Point to point communication

Point-to-point communication



- Primitives to send a message
  - From a single sender
  - To a single recipient

- MPI defines
  - How the data is represented (using datatypes)
  - How the message recipient is identified (using its rank inside the communicator)
  - How the actual communication is implemented
    - Blocking vs non-blocking
    - Synchronous vs asynchronous

Point-to-point communication

```
MPI_Send(
    void* data,
    int count,
    MPI_Datatype datatype,
    int destination,
    int tag,
    MPI_Comm communicator);
```

```
MPI_Recv(
    void* data, int count,
    MPI_Datatype datatype,
    int source,
    int tag,
    MPI_Comm communicator,
    MPI_Status* status);
```

Point-to-point communication

```
MPI_Send(
    void* data,
    int count,
    MPI_Datatype datatype,
    int destination,
    int tag,
    MPI_Comm communicator);
```

```
MPI_Recv(
    void* data, int count,
    MPI_Datatype datatype,
    int source,
    int tag,
    MPI_Comm communicator,
    MPI_Status* status);
```

MPI data types

- To enable portability, MPI predefines its elementary data types
- A specific data type exists for each primitive C type
- Programmers may also create their own, structured data types
  - Arrays
  - Structs
  - Compositions of the above

MPI data types

MPI datatype	C equivalent
MPI_SHORT	short int
MPI_INT	int
MPI_LONG	long int
MPI_LONG_LONG	long long int
MPI_UNSIGNED_CHAR	unsigned char
MPI_UNSIGNED_SHORT	unsigned short int
MPI_UNSIGNED	unsigned int
MPI_UNSIGNED_LONG	unsigned long int
MPI_UNSIGNED_LONG_LONG	unsigned long long int
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_LONG_DOUBLE	long double
MPI_BYTE	char

Recipients and message tags

- In send primitives, the recipient is specified through its rank and communicator

- Receive primitives also specify the rank of the sender
  - Can be set to `MPI_ANY_SOURCE` to receive a message from any source

- Message tags can also be used to differentiate messages of different types
  - A receive primitive only receives messages having the specified tag
  - The tag can be set to `MPI_ANY_TAG` to receive any message regardless of its tag

Blocking vs non-blocking calls

- Most MPI p2p procedures can be used in blocking or non-blocking mode
- Blocking
  - A blocking send only returns when it is safe to modify the application buffer for reuse
    - Does not imply that the data is received by the receiving process
    - Data may be sitting in a system buffer on the receiving host
  - A blocking send can be synchronous (with a confirmation of the receiver) or asynchronous if a system buffer is used to hold the data for eventual delivery
  - A blocking receive only returns when the data has arrived and is ready for use by the program

Blocking vs non-blocking calls

- Non-blocking
  - Non-blocking send and receive do not wait for any communication to happen
  - Enables executing further instructions while the communication takes place
  - It is unsafe to modify the application buffer until you know the requested operation is performed
    - There are wait procedures used to do this

Point-to-point calls

Primitives	Semantics
MPI_Send	Blocking (might block or buffer), asynchronous
MPI_Isend	Nonblocking, asynchronous
MPI_Ssend	Blocking, synchronous
MPI_Recv	Blocking
MPI_Irecv	Non-blocking
MPI_Sendrecv	Send a message and post a (blocking) receive before blocking
MPI_Wait	Wait until an asynchronous call is completed

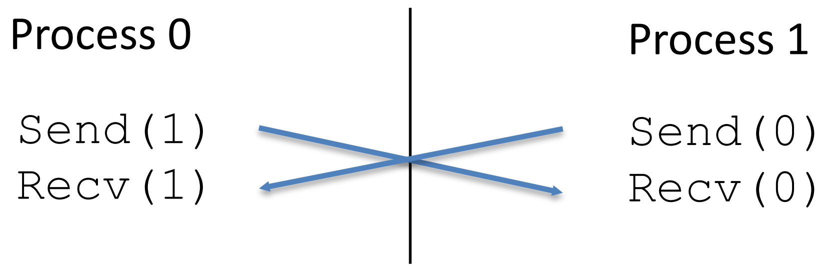
Ordering

- MPI guarantees FIFO order of messages between a sender and a receiver
  - Sender S sends messages M1 and M2 to the same receiver R …
  - … R will receive M1 before M2

- If a receiver performs two receive calls (R1 and R2) both looking for the same message …
- … R1 will receive the message before R2

- The above rules only apply to send/receive operations submitted by the same thread
  - This is a common scenario in MPI programs
    - Mixing MPI and multi-threading is complex and not frequently used
  - No guarantees in the case of multiple threads

Deadlock



- With synchronous send calls, there is the risk of creating deadlocks
- In the above situation, a deadlock might occur even using a standard blocking, asynchronous send
  - With large messages if the system buffer is not big enough
  - The code is unsafe, as its correctness depends on the system implementation

Deadlock

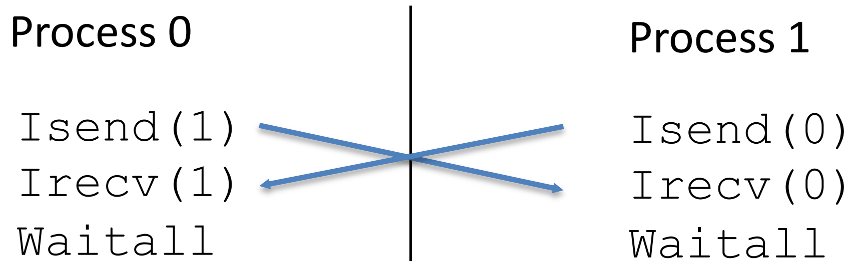
- The above example can be easily generalized to more than two processes
  - Circular dependencies

```
int a[10], b[10], np, myrank;
MPI_Status status;
...
MPI_Comm_size(MPI_COMM_WORLD, &np);
MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
MPI_Ssend(a, 10, MPI_INT, (myrank+1)%np, 1, MPI_COMM_WORLD);
MPI_Recv(b, 10, MPI_INT, (myrank-1)%np, 1, MPI_COMM_WORLD);
...
```

Another example of deadlock

```
int a[10], b[10], myrank;
MPI_Status status;
...
MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
if (myrank == 0) {
      MPI_Ssend(a, 10, MPI_INT, 1, 1, MPI_COMM_WORLD);
      MPI_Ssend(b, 10, MPI_INT, 1, 2, MPI_COMM_WORLD);
} else if (myrank == 1) {
      MPI_Recv(b, 10, MPI_INT, 0, 2, MPI_COMM_WORLD);
      MPI_Recv(a, 10, MPI_INT, 0, 1, MPI_COMM_WORLD);
}
...
```

Non-blocking calls and deadlock



- Non blocking procedures can be used to avoid deadlocks

Example

```
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]);

  /* Do some work here */
  MPI_Waitall(4, reqs, stats);
  MPI_Finalize();
}
```

Dynamic receiving

- How to deal with messages of unknown length?

- Two alternative solutions

1. The process first sends a message with the size of the subsequent message
2. The receiver invokes MPI_Probe to check the availability of messages in its buffer
   - MPI_Probe fills a MPI_Status structure with
     - Sender
     - Tag
     - Message size
   - The receiver can use this information to post an MPI_Recv

Exercise 1

- Modify the deadlock
  - Using normal asynchronous communication
    - Does it work?
    - Is it safe?
  - Using non-blocking calls
  - Using MPI_Sendrecv

Exercise 2

- Write a ping-pong program
  - Process P0 sends a message to process P1 with a number
  - Process P1 replies and increments the number
  - Stop when the number overcomes a given threshold

Exercise 3

- Write a ring communication program
  - A set of processes exchange a token
    - Process P0 sends the token to P1
    - Process P1 sends the token to P2
    - …
  - The program terminates after a given number of iterations over the ring

Exercise 4

- Complete the probe.c example

- Write the code of the receiver
  - Query the availability of a message and get its characteristics
  - Receive the message and print it