MPI

MESSAGE PASSING INTERFACE

NON-BLOCKING FUNCTIONS

- Previous functions are blocking: the following instruction is not executed until they have finished.
- Non-blocking functions: function returns, whether data transfer is finished or not.
  - Requires function to query the status of the data transfer
  - Message buffers are needed (legth of message is limited)
  - Overlapping of communication and computation is possible (reduction of execution time)
- Examples:
  - MPI_Isend
  - MPI_Irecv
  - MPI_Iprobe
  - etc..

NON-BLOCKING FUNCTIONS

- `MPI_Isend`: sends a message but returns before copying into the buffer (buf cannot be modified before data is received).
  - `int MPI_Isend(void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_COMM comm, MPI_Request *request)`
    - `buf`: buffer with data to be sent
    - `count`: size of data to be sent
    - `datatype`: type of data to be sent
    - `dest`: destiny process id (where data are sent to)
    - `tag`: message id
    - `comm`: communicator
    - `request`: to identify the operation in progress.

NON-BLOCKING FUNCTIONS

- `MPI_Irecv`: receives a message but returns before copying into the buffer. Two possibilities: waiting till it ends (MPI_Wait) or checking the communication's status (MPI_Test).
  - `int MPI_Irecv(void *buf, int count, MPI_Datatype datatype, int source, int tag, MPI_COMM comm, MPI_Request *request)`
    - `buf`: buffer where received data will be stored
    - `count`: size of data to be received
    - `datatype`: type of data to be received
    - `source`: source process id (where data come from)
    - `tag`: message id
    - `comm`: communicator
    - `request`: to identify the operation in progress.

NON-BLOCKING FUNCTIONS

- `MPI_Test`: checks if the non-blocking operation has finalized.
  - `int MPI_Test(MPI_Request *request, int *flag, MPI_Status *status)`
    - `request`: to identify the operation we are waiting for.
    - `status`: gathers information about the non-blocking operation that has finalized (pointer)
    - If *flag*=true then the operation has finalized; *request* is freed, *status* is initialized

NON-BLOCKING FUNCTIONS

- `MPI_Wait`: blocks till the non-blocking operation has finished.
  - `int MPI_Wait(MPI_Request *request, MPI_Status *status)`
    - `request`: to identify the operation we are waiting for.
    - `status`: gathers information about the non-blocking operation that has finalized (pointer)

NON-BLOCKING FUNCTIONS

- `MPI_Iprobe`: checks if there are messages to receive.
  - `int MPI_IProbe(int source, int tag, MPI_Comm comm, int *flag, MPI_Status *status)`
    - `source`: source process id (where we check if it has sent data)
    - `tag`: message id (the one that we check if it has been received)
    - `comm`: communicator
    - `status`: gathers information about the message to be received (pointer)
    - If flag=true the message fits (source, tag, comm). Status gives more info.
    - MPI_ANY_SOURCE and MPI_ANY_TAG allow us to wait for messages from any source and/or with any tag.
    - It does not block if there are no messages. If there are messages, they are received with MPI_Recv.

NON-BLOCKING FUNCTIONS

– `MPI_Probe`: checks for specific messages.

- `int MPI_Probe(int source, int tag, MPI_Comm comm, MPI_Status *status)`
  - – `source`: source process id (where we check if it has sent data)
  - – `tag`: message id (the one that we check if it has been received)
  - – `comm`: communicator
  - – `status`: gathers information about the message to be received (pointer)
  - – MPI_ANY_SOURCE and MPI_ANY_TAG allow us to wait for messages from any source and/or with any tag.
  - – It blocks: it finishes when there are messages that fit the given arguments. If there are messages, they are received with MPI_Recv.

EXAMPLE

```
…
int datasize, *buf, source;
…
//Comprobamos si hay algún mensaje pendiente
MPI_Probe(MPI_ANY_SOURCE, 0, comm, &status);
//Obtenemos el tamaño de los datos a recibir y
//reservamos memoria
MPI_Get_count(status, MPI_INT, &datasize);
buf=malloc(datasize*sizeof(int));
//Obtenemos el proceso fuente del mensaje
source=status.MPI_SOURCE;
//Recibimos el mensaje
MPI_Recv(buf,datasize,MPI_INT,source,0,comm,&status);
…
```

NON-BLOCKING FUNCTIONS

– Waits for any specified send or receive to complete:

- `int MPI_Waitany (int count, MPI_Request *array_of_requests, int *index, MPI_Status *status);`
  - `count`: list length
  - `array_of_requests`: array of requests
  - `index`: index of handle for operation that completed
  - `status`: status object
- It is used to wait for the completion of one out of several requests.

NON-BLOCKING FUNCTIONS

– Waits for all given communications to complete:

- `int MPI_Waitall (int count, MPI_Request *array_of_requests, MPI_Status *array_of_statuses);`
  - `count`: list length
  - `array_of_requests`: array of requests
  - `array_of_statuses`: array of status objects
- Blocks until all communication operations associated with active handles in the list complete, and returns the status of all these operations.
  - This includes the case where no handle in the list is active

NON-BLOCKING FUNCTIONS

– Waits for some given communications to complete:

- `int MPI_Waitsome (int incount, MPI_Request *array_of_requests, int *outcount, int *array_of_indices, MPI_Status *array_of_statuses);`
  - `incount`: length of array_of_requests
  - `array_of_requests`: array of requests
  - `outcount`: number of completed requestes
  - `array_of_indices`: array of indices of operations to be completed
  - `array_of_statuses`: array of status objects for operations that completed
- Waits until at least one of the operations associated with active handles in the list have completed.

NON-BLOCKING FUNCTIONS

– Tests for completion of any one previously initiated communication in a list:

- `int MPI_Testany (int count, MPI_Request *array_of_requests, int *index, int *flag, MPI_Status *status);`
  - `count`: list length
  - `array_of_requests`: array of requests
  - `index`: index of operation that completed or MPI_UNDEFINED if none completed
  - `flag`: true if one of the operations is completed
  - `status`: status object
- It tests for completion of either one or none of the operations associated with active handles.

NON-BLOCKING FUNCTIONS

– Tests for the completion of all previously initiated communications in a list:

- `int MPI_Testall (int count, MPI_Request *array_of_requests, int *flag, MPI_Status *array_of_statuses);`
  - `count`: list length
  - `array_of_requests`: array of requests
  - `flag`: true if previously initiated communications are complete
  - `array_of_statuses`: array of status objects
- Returns flag=true if all communications associated with active handles in the array have complete (this includes the case where no handle in the list is active)

NON-BLOCKING FUNCTIONS

– Tests for completion of one or more previously initiated communications in a list:

- `int MPI_Testsome (int incount, MPI_Request *array_of_requests, int *outcount, int *array_of_indices, MPI_Status *array_of_statuses);`
  - `incount`: length of array_of_requests
  - `array_of_requests`: array of requests
  - `outcount`: number of completed requestes
  - `array_of_indices`: array of indices of operations that completed
  - `array_of_statuses`: array of status objects for operations that completed
- Behaves like MPI_Waitsome except that it returns immediately

AVOIDING INTERLOCKS

– An interlock is produced:

- When one or more processes achieves a blocking receiving routine but the message never comes. Process waits indefinitely and there is no error message.
- E.g. 2 processes interchange messages and we do not program it well.

EXAMPLE: IT ALWAYS WORKS

```
…
if (myid == 0){
  MPI_Send(&a,1,MPI_FLOAT,1,tag,MPI_COMM_WORLD);
  MPI_Recv(&b,1,MPI_FLOAT,1,tag,MPI_COMM_WORLD,&status);
}elseif (myid == 1){
  MPI_Recv(&a,1,MPI_FLOAT,0,tag,MPI_COMM_WORLD,&status);
  MPI_Send (&b,1,MPI_FLOAT,0,tag,MPI_COMM_WORLD);
}
…
```

EXAMPLE: IT NEVER WORKS

```
…
if (myid == 0){
 MPI_Recv (&b,1,MPI_FLOAT,1,tag,MPI_COMM_WORLD,&status)
 MPI_Send(&a,1,MPI_FLOAT,1,tag,MPI_COMM_WORLD);
}elseif (myid == 1){
 MPI_Recv(&a,1,MPI_FLOAT,0,tag,MPI_COMM_WORLD,&status);
 MPI_Send(&b,1,MPI_FLOAT,0,tag,MPI_COMM_WORLD);
}
…
```

EXAMPLE: IT MAY WORK

```
…
if (myid == 0){
  MPI_Send(&a,1,MPI_FLOAT,1,tag,MPI_COMM_WORLD);
  MPI_Recv(&b,1,MPI_FLOAT,1,tag,MPI_COMM_WORLD,&status)
}elseif (myid == 1){
  MPI_Send(&b,1,MPI_FLOAT,0,tag,MPI_COMM_WORLD);
  MPI_Recv(&a,1,MPI_FLOAT,0,tag,MPI_COMM_WORLD,&status);
}
…
```

Avoid it!

EXAMPLE: IT MAY WORK

```
…
if (myid == 0){
  MPI_Send(&a,1,MPI_FLOAT,1,tag,MPI_COMM_WORLD);
  MPI_Recv(&b,1,MPI_FLOAT,1,tag,MPI_COMM_WORLD,&status)
}elseif (myid == 1){
  MPI_Send(&b,1,MPI_FLOAT,0,tag,MPI_COMM_WORLD);
  MPI_Recv(&a,1,MPI_FLOAT,0,tag,MPI_COMM_WORLD,&status);
}
…
```

Recv() ALWAYS blocks until the correspoding Send() is carried out

Depending on the platform, Send() MAY block until the corresponding Recv() is carried out

AVOIDING INTERLOCKS

– `MPI_Sendrecv`: sending and receiving data at the same time.

- `int MPI_Sendrecv(void *sendbuf, int sendcount, MPI_Datatype sendtype, int dest, int sendtag, void *recvbuf, int recvcount, MPI_Datatype recvtype, int source, int recvtag, MPI_Comm comm, MPI_Status *status)`

`sendbuf`: buffer of data to be sent

`sendcount`: size of data to be sent // `sendtype`: type of data to be sent

`dest`: destiny process id // `sendtag`: tag of the sending data

`recvbuf`: buffer of data to be received

`recvcount`: size of data to be received // `recvtype`: type of data to be received

`source`: source process id // `recvtag`: tag of the receiving data

`comm`: communicator

`status`: gathers information about the receiving operation (pointer)

EXAMPLE

```
…
tag1=1;
tag2=2;
if (myid == 0){
  MPI_Sendrecv(&a,1, MPI_FLOAT,1,tag1, &b,1,
  MPI_FLOAT,1,tag2, MPI_COMM_WORLD,&status);
elseif (myid == 1){
  MPI_Sendrecv(&b,1, MPI_FLOAT,0,tag2, &a,1,
  MPI_FLOAT,0,tag1, MPI_COMM_WORLD,&status);
}
…
```

Recomended!

AVOIDING INTERLOCKS

– `MPI_Sendrecv_replace`: performs send and receive in one single function call and operates only one single buffer.

- `int MPI_Sendrecv_replace(void *buf, int count, MPI_Datatype type, int dest, int sendtag, int source, int recvtag, MPI_Comm comm, MPI_Status *status)`

  `buf`: buffer of data to be sent/to store the received data

  `count`: size of message (in elements)

  `type`: type of data

  `dest`: destiny process id // `sendtag`: tag of the sending data

  `source`: source process id // `recvtag`: tag of the receiving data

  `comm`: communicator

  `status`: gathers information about the receiving operation (pointer)

EXERCISE

Write a MPI program to communicate N processes in a ring way. i Process must tell i+1 process its machine name.

- Implementation 1: use send() and recv().
- Implementation 2: use Isend() and Irecv().
- Implementation 3: use Sendrecv().
- In implementations 1 and 2, Send()/Isend() and Recv()/Irecv() can be executed in any order?