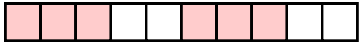
PRACE Course: Intermediate MPI

9-11 November 2022

MPI Point-to-Point Communication

Messages

- A message contains a number of elements of some particular datatype.
- MPI datatypes:
  - Basic datatype.
  - Derived datatypes 
- C types are different from Fortran types.
- Datatype handles are used to describe the type of the data in the memory.

Example: message with 5 integers

2345	**654**	**96574**	**-12**	**7676**

MPI Datatype	C datatype
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	

int arr[5]
count =5
datatype=MPI_INT

2345	654	96574	-12	7676

MPI Datatype	Fortran datatype
MPI_INTEGER	integer
MPI_INTEGERX	integer*X X=1,2,4,8,16
MPI_REAL	real
MPI_REALX	real*X X=4,8,16
MPI_DOUBLE_PRECISION	double precision
MPI_COMPLEX	complex
MPI_COMPLEXY	complex*Y Y=8,16,32
MPI_ LOGICAL	logical
MPI_CHARACTER	character(1)
MPI_BYTE	
MPI_PACKED	

Implementation dependent

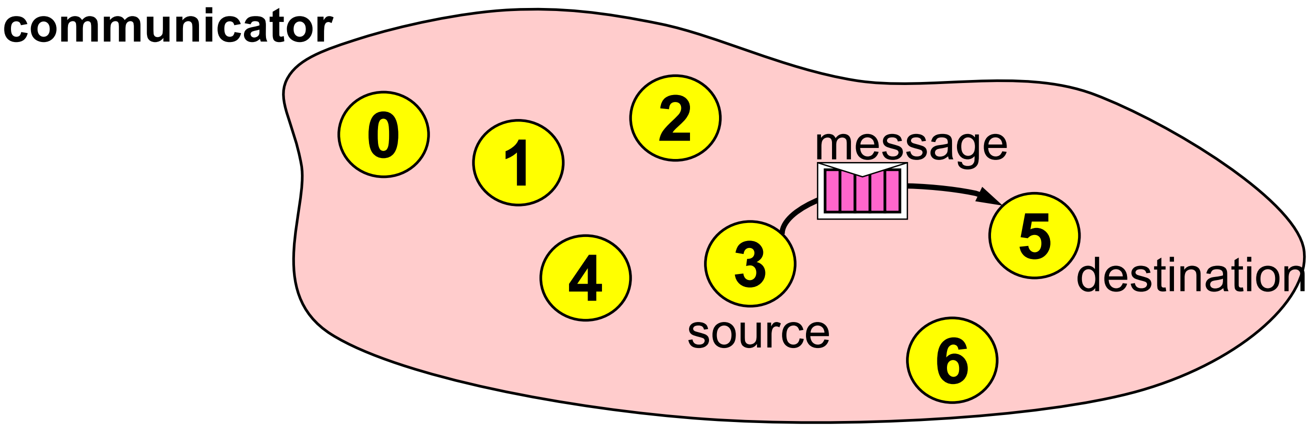
integer :: arr(5)
count=5

datatype=MPI_INTEGER

2345	654	96574	-12	7676

Point-to-Point Communication

- Communication between two processes.
- Source process sends message to destination process.
- Communication takes place within a communicator, e.g., MPI_COMM_WORLD.
- Processes are identified by their ranks in the communicator



Sending a Message

- C:

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

- Fortran:

```
MPI_SEND(buf, count, datatype, dest, tag, comm, ierror)
        TYPE(*), DIMENSION(..) :: buf
        TYPE(MPI_Datatype) :: datatype; TYPE(MPI_Comm) :: comm
        integer :: count, dest, tag; integer,optional :: ierror
```

- buf is the starting point of the message with count elements, each described with datatypeHandle.
- dest is the rank of the destination process within the communicator comm.
- tag is an additional nonnegative integer piggyback information, additionally transferred with the message.

Receiving a Message

- C:
  int MPI_Recv(void *buf, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI_Status *status)

- Fortran:
  MPI_RECV(buf, count, datatype, source, tag, comm, status, ierror)
  TYPE(*), DIMENSION(..) :: buf; TYPE(MPI_Status) :: status
  TYPE(MPI_Datatype) :: datatype; TYPE(MPI_Comm) :: comm
  integer :: count, source, tag; integer,optional :: ierror

- buf/count/datatype describe the receive buffer.
- Receiving the message sent by process with rank source in comm.
- Envelope information is returned in status.
- Only messages with matching tag are received.

Requirements for Point-to-Point Communications

For a communication to succeed:

- Sender must specify a valid destination rank.
- Receiver must specify a valid source rank.
- The communicator must be the same.
- Tags must match.
- Message datatypes must match.

  Message matching rule: receives only if comm, source, and tag match.
- Receiver's buffer must be large enough.

Example - One Ping

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

int main(int argc, char **argv){
   int myRank, ierror, arr[5];
   MPI_Status stat;

   ierror=MPI_Init(&argc,&argv);
   ierror=MPI_Comm_rank(MPI_COMM_WORLD,&myRank);
   if (myRank == 0) {
      arr[0]=2345; arr[1]=654; arr[2]=96574; arr[3]=-12; arr[4]=7676;
      MPI_Send(arr, 5, MPI_INT, 1, 100, MPI_COMM_WORLD);
   } else if (myRank == 1) {
      MPI_Recv(arr, 5, MPI_INT, 0, 100, MPI_COMM_WORLD, &stat);
   }
   ierror=MPI_Finalize();
   return 0;
}
```

Ping: Process 0 sends a message to process 1

Run with two processes

Wildcards

- Receiver can wildcard.
- To receive from any source
  - source = MPI_ANY_SOURCE
- To receive from any tag:
  - tag = MPI_ANY_TAG
- Actual source and tag are returned in the receiver's status parameter.
- Use only when necessary and beneficial. It is much safer to specify the source and tag when you know them

From MPI-4.0

- An MPI application can tell the MPI library that it will never use MPI_ANY_SOURCE and/or MPI_ANY_TAG on this communicator by setting info assertion to true/false.

Communication Envelope

- Envelope information is returned from MPI_RECV in status.
- C:

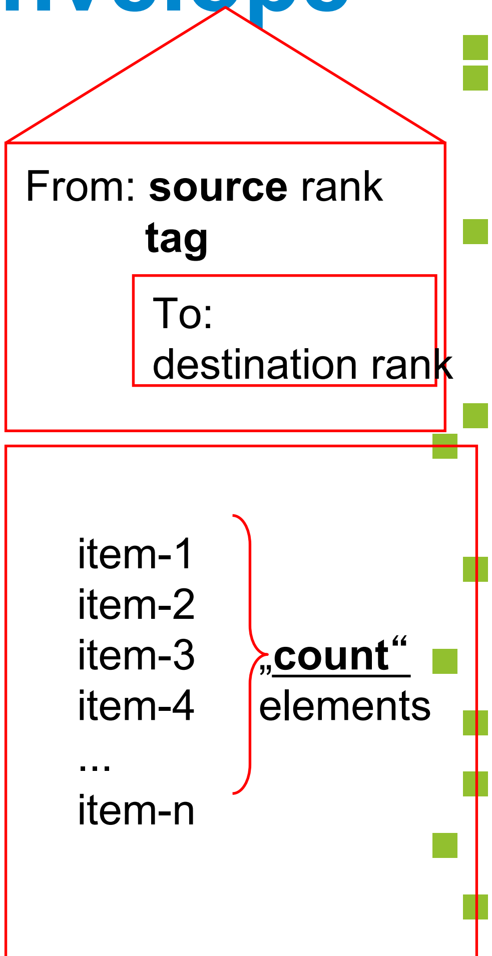
```
MPI_Status status;

status.MPI_SOURCE
status.MPI_TAG
count via MPI_Get_count()
```

- Fortran:

```
TYPE(MPI_Status) :: status

status%MPI_SOURCE
status%MPI_TAG
count via MPI_GET_COUNT()
```



Receive Message Count

- C:
  ```
  int MPI_Get_count(MPI_Status *status, MPI_Datatype
              datatype, int  *count)
  ```

- Fortran:
  ```
  MPI_GET_COUNT(status, datatype, count, ierror)
      TYPE(MPI_Status) :: status
      TYPE(MPI_Datatype) :: datatype
      integer :: count; integer,optional :: ierror
  ```

Communication Modes

- Send communication modes:
  - synchronous send — MPI_**S**send
  - buffered [asynchronous] send — MPI_**B**send
  - standard send — MPI_**Send**
  - ready send — MPI_**R**send

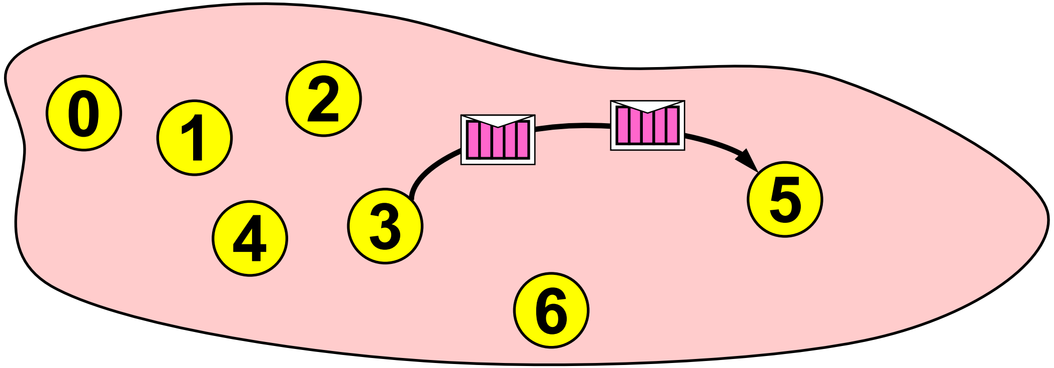
- Receiving all modes — MPI_**Recv**

Communication Modes - Definitions

Sender modes	Definition	Notes
Synchronous send **MPI_Ssend**	Only completes when the receive has started	risk of deadlock risk of serialization risk of waiting —> idle time
Buffered send **MPI_Bsend**	Always completes (unless an error occurs), irrespective of receiver	needs application-defined buffer to be declared with MPI_BUFFER_ATTACH/DETACH
Standard send **MPI_Send**	Standard send. Either synchronous or buffered	Uses an internal buffer
Ready send **MPI_Rsend**	May be started only if the matching receive is already posted!	highly dangerous!
Receive **MPI_Recv**	Completes when the message (data) has arrived	Same routine for all communication modes

Message Order Preservation

- Rule for messages on the same connection, i.e., same communicator, source, and destination rank:
- Messages do not overtake each other.
- This is true even for non-synchronous sends.



- MPI messages are non-overtaking: if one process send two messages to another, then they will be received in the order they were sent

Timing in MPI

- C:
  - double MPI_Wtime(void);
- Fortran:
  - DOUBLE PRECISION MPI_WTIME()
- The elapsed (wall-clock) time between two points:

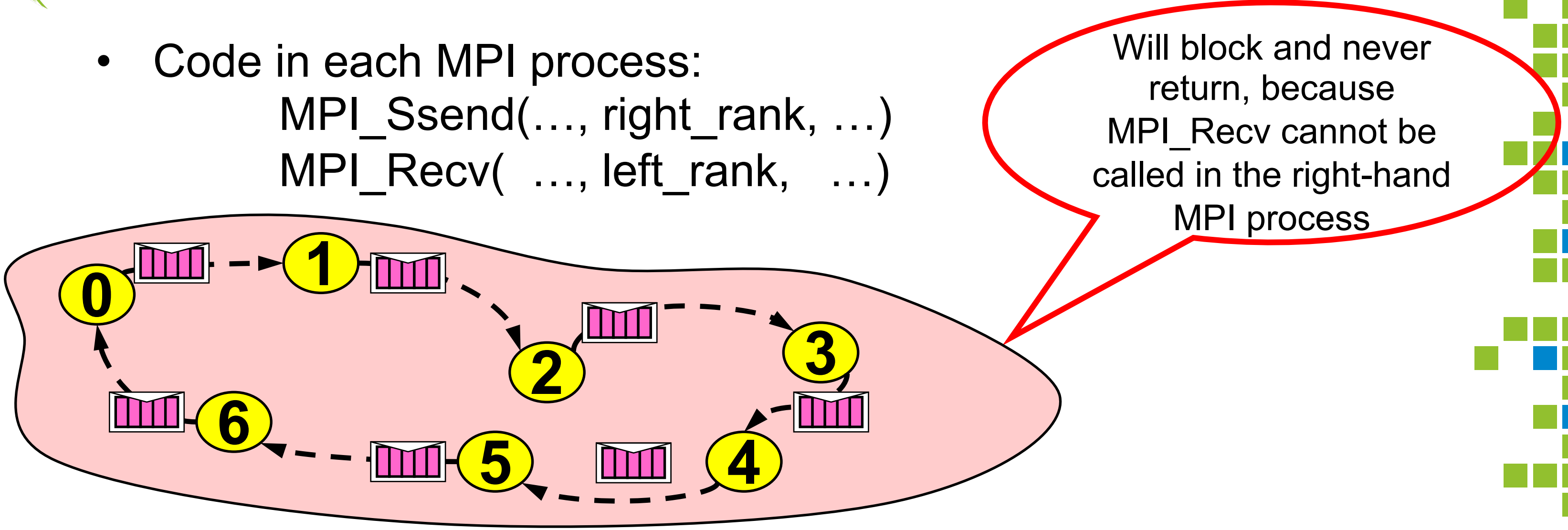
```
double t1, t2;
t1 = MPI_Wtime();
... work to be timed …
t2 = MPI_Wtime();
printf("Elapsed time is %f\n", t2 - t1 );
```

```
real(8) :: t1, t2
t1= MPI_Wtime ()
 … work to be timed …
t2 = MPI_Wtime ()
Print*, 'Elapsed time is', t2-t1
```

- MPI_Wtick(): Returns the number of seconds between processor clock ticks.

Deadlock

- Code in each MPI process:
  MPI_Ssend(…, right_rank, …)
  MPI_Recv(…, left_rank, …)



- Reorganise the communications, i.e. First even processes send odd processes receive, Then odd processes send even processes receive.(Inefficient)
- Use MPI_SendRecv(), which combines send and recv in a single deadlock-free call

Example - Deadlock

- Problem
(Unless MPI_Send/MPI_Recv is buffered)

```
if (myRank == 0) {
    MPI_Send(a, 5, MPI_INT, 1, 100, MPI_COMM_WORLD);
    MPI_Recv(b, 5, MPI_INT, 1, 101, MPI_COMM_WORLD, &status);
} else if (myRank == 1) {
    MPI_Send(b, 5, MPI_INT, 0, 101, MPI_COMM_WORLD);
    MPI_Recv(a, 5, MPI_INT, 0, 100, MPI_COMM_WORLD, &status);
}
```

Solutions:

- Reverse the order of one of the send/receive pairs

```
if (myRank == 0) {
    MPI_Send(a, 5, MPI_INT, 1, 100, MPI_COMM_WORLD);
    MPI_Recv(b, 5, MPI_INT, 1, 101, MPI_COMM_WORLD, &status);
} else if (myRank == 1) {
    MPI_Recv(a, 5, MPI_INT, 0, 100, MPI_COMM_WORLD, &status);
    MPI_Send(b, 5, MPI_INT, 0, 101, MPI_COMM_WORLD);}
```

- Use non-blocking communication

Non-Blocking Communication

Separate communication into three phases:

- Initiate non-blocking communication
  - returns Immediately
  - routine name starting with MPI_I…
- Do some work
  - “latency hiding”
- Wait for non-blocking communication to complete, i.e.
  - The send buffer is read out or
  - The receive buffer is filled in.

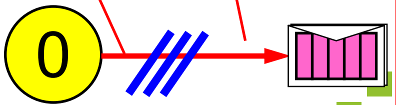
Non-Blocking Examples

- Non-blocking send

MPI_Isend(...)

doing some other work

MPI_Wait(...)

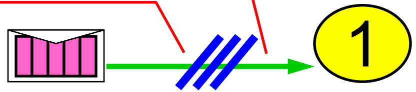


- Non-blocking receive

MPI_Irecv(...)

doing some other work

MPI_Wait(...)



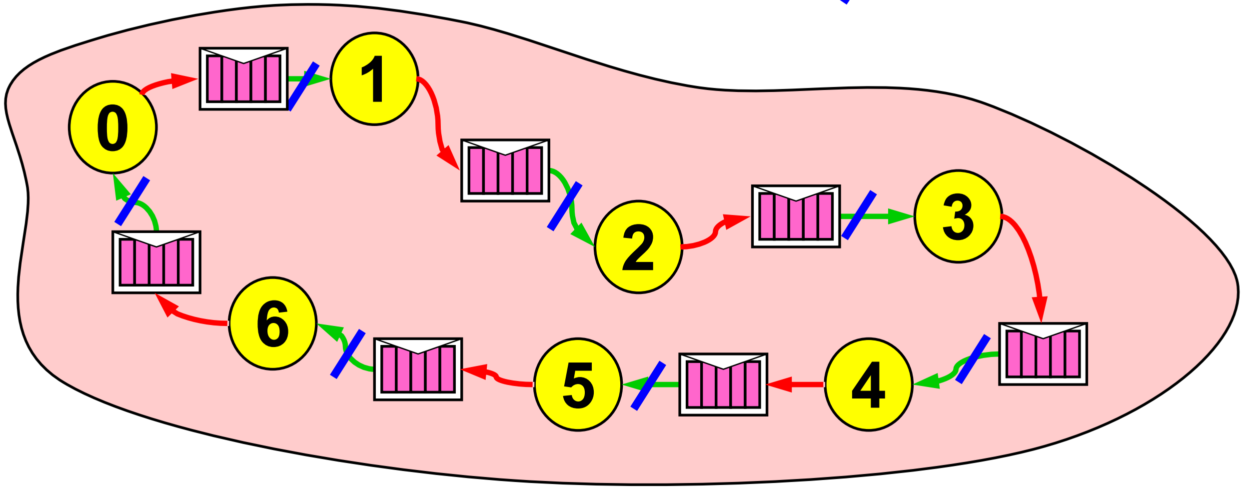
= waiting until operation locally completed

Non-Blocking Send

- Initiate non-blocking send

  → in the ring example: Initiate non-blocking send to the right neighbor
- Do some work:

  → in the ring example: Receiving the message from left neighbor
- Now, the message transfer can be completed
- Wait for non-blocking send to complete



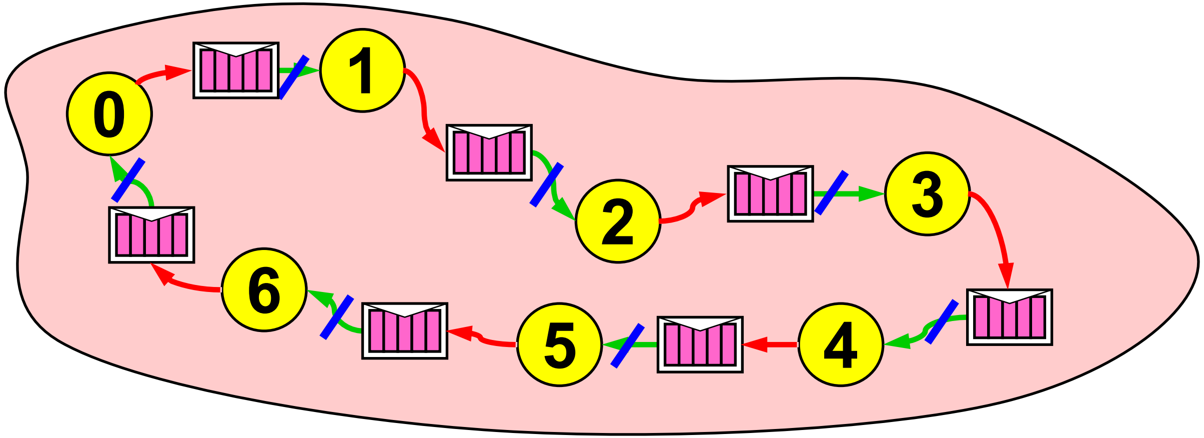
Non-Blocking Send

- C:
  int MPI_Isend(void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm, MPI_Request * request)

- Fortran:
  MPI_ISEND(buf, count, datatype, dest, tag, comm, request, ierror)
  TYPE(*), DIMENSION(..) :: buf
  TYPE(MPI_Datatype) :: datatype
  TYPE(MPI_Comm) :: comm
  TYPE(MPI_Request) :: request
  integer :: count, dest, tag; integer,optional :: ierror

Non-Blocking Receive

- Initiate non-blocking receive
  - in the ring example: Initiate non-blocking receive from left neighbor
- Do some work:
  - in the ring example: Sending the message to the right neighbor
- Now, the message transfer can be completed
- Wait for non-blocking receive to complete



Non-Blocking Receive

- C:
```
int MPI_Irecv(void *buf, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI_Request * request)
```

- Fortran:
```
MPI_IRECV(buf, count, datatype, source, tag, comm, request, ierror)
        TYPE(*), DIMENSION(..) :: buf
        TYPE(MPI_Datatype) :: datatype
        TYPE(MPI_Comm) :: comm
        TYPE(MPI_Request) :: request
        integer :: count, dest, tag; integer,optional :: ierror
```

Completion

- C:
  MPI_Wait(MPI Request *request, MPI_Status *status);
  MPI_Test (MPI Request *request, int *flag, MPI_Status *status);

- Fortran:
  MPI_Wait(request, status, ierror)
  MPI_Test(request, flag, status, ierror)
  TYPE(MPI_Request) :: request; integer :: ierror
  TYPE(MPI_Status) :: status; logical :: flag

- buf must not be used between Isend/Irecv and Wait
- one must
  - wait or
  - loop with TEST until request is completed, i.e., flag == 1 or.true.

Blocking and Non-Blocking

- Send and receive can be blocking or non-blocking.
- A blocking send can be used with a nonblocking receive, and vice-versa.

Send Mode	Blocking Function	Nonblocking Function
Standard	MPI_Send	MPI_Isend
Synchronous	MPI_Ssend	MPI_Issend
Ready	MPI_Rsend	MPI_Irsend
Buffered	MPI_Bsend	MPI_Ibsend

- Issend + Wait is equivalent to blocking call: Ssend
- MPI_Sendrecv: Irecv + Send + Wait

Example - Deadlock

- Problem
(Unless MPI_Send/MPI_Recv is buffered)

```
if (myRank == 0) {
    MPI_Send(a, 5, MPI_INT, 1, 100, MPI_COMM_WORLD);
    MPI_Recv(b, 5, MPI_INT, 1, 101, MPI_COMM_WORLD, &status);
} else if (myRank == 1) {
    MPI_Send(b, 5, MPI_INT, 0, 101, MPI_COMM_WORLD);
    MPI_Recv(a, 5, MPI_INT, 0, 100, MPI_COMM_WORLD, &status);
}
```

Solutions:

- Reverse the order of one of the send/receive pairs

- Use non-blocking
- communication

```
if (myRank == 0) {
    MPI_Isend(a, 5, MPI_INT, 1, 100, MPI_COMM_WORLD,&request);
    MPI_Recv(b, 5, MPI_INT, 1, 101, MPI_COMM_WORLD, &status);
    MPI_Wait(&request, &status);
} else if (myRank == 1) {
    MPI_Recv(a, 5, MPI_INT, 0, 100, MPI_COMM_WORLD, &status);
    MPI_Send(b, 5, MPI_INT, 0, 101, MPI_COMM_WORLD);}
```

Multiple Non-Blocking Communications

You have several request handles:

- Wait or test for completion of **one** message
  - MPI_Waitany / MPI_Testany
- Wait or test for completion of **all** messages
  - MPI_Waitall / MPI_Testall
- Wait or test for completion of **at least one** message
  - MPI_Waitsome / MPI_Testsome