Parallel Processing

Lecture 4 - MPI

Felicja Okulicka-Dłużewska

F.Okulicka@mini.pw.edu.pl

Warsaw University of Technology

Faculty of Mathematics and Information Science

Go to full-screen mode now by hitting CTRL-L

MPI

MPI (Message Passing Interface) is the a library specification for message-passing programs.

- MPI was designed for high performance on both massively parallel machines and on workstation clusters.
- It is widely available, with both free available and vendor-supplied implementations and was developed by a broadly based committee of vendors, implementors, and users.
- By MPI the SPMD (Single Program, Multiple Data) with the message passing is simulated.
- Each processor in a message passing program runs a process - the sub-program written in a conventional sequential language.
- All variables are private - there is no common memory.

MPI

The goals and scope of MPI are:

- To provide source-code portability
- To allow efficient implementation
- To offer a great deal of functionality
- To offer support for heterogeneous parallel architecture

With the Message Passing Model communications are explicit and generally quite visible and under the control of the programmer. With the Data Parallel Model, communications often occur transparently to the programmer, particularly on distributed memory architectures. The programmer may not even be able to know exactly how inter-task communications are being accomplished.

Model of computation in MPI secures:

- portability
- scalability, i.e. the ability of a parallel program's performance to scale is a result of a number of interrelated factors.

Program compilation and running

The compilation is done using the following compilators:

mpicc program-name

mpic++ program-name

mpicf90 program-name

mpicf77 program-name

To start the MPI program, normally mpirun(MPI-1) or mpiexec (MPI-2) is called:

mpiexec -n <numprocs> <program>

mpirun -n <numprocs> <program>

The number of processes <numprocs> are generated. Other arguments to mpiexec may be implementation-dependent.

Using MPI library- MPI headers

MPI headers

Header files must be included:

C	include <mpi.h>
FORTAN	include 'mpif.h'

MPI functions

Functions are called as follows:

C	Error=MPI_xxxx(parameter,...);
FORTAN	CALL MPI_xxxx(parameter,..., IError)

MPI controls its own internal data structures and releases 'handlers' to allow programmers to refer to these. C handles are of defined typedefs whereas Fortran handles are INTEGERs.

Types of MPI Calls

When discussing MPI procedures the following terms are used:

- **local** - if the completion of the procedure depends only on the local executing process. Such an operation does not require an explicit communication with another user process. MPI calls that generate local objects or query the status of local objects are local (for example MPI_Comm_rank(...)),
- **non-local** - if completion of the procedure may require the execution of some MPI procedure on another process. Many MPI communication calls are non-local (for example send: MPI_Send(...)),

Types of MPI Calls

- **blocking** - if return from the procedure indicates the user is allowed to re-use resources specified in the call. Any visible change in the state of the calling process affected by a blocking call occurs before the call returns (for example MPI_Send(...)),
- **nonblocking** - if the procedure may return before the operation initiated by the call completes, and before the user is allowed to re-use resources (such as buffers) specified in the call. A nonblocking call may initiate changes in the state of the calling process that actually take place after the call returned: e.g. a nonblocking call can initiate a receive operation, but the message is actually received after the call returned (for example MPI_ISend(...)),
- **collective** - if all processes in a process group need to invoke the procedure (for example broadcast: MPI_Broad(...)).

MPI programs

MPI program consists of the processes which execute the code eventually dependent on the number of the process called rank. To write simple program without communication between processes we need only four functions :

- MPI_Init,
- MPI_Comm_rank,
- MPI_Comm_size,
- MPI_Finalized.

MPI programs

Communication between processes is done by message passing. The simplest communication is point-to-point. It is supported by the functions of sending and receiving the messages. The most popular are:

- blocking standard send: MPI_Send,
- receive: MPI_Recv.

MPI_Init

The routine for initializing MPI must be called as the first MPI function. It creates the MPI_COMM_WORLD communicator as the environment for communication between the group of processes.

C	int MPI_Init(int *argc, char ***argv) int MPI_Init (&argc,&argv)
C++ (in the MPI:: namespace)	void Op::Init(User_function* function, bool commute) void Init(int& argc, char**& argv)
FORTAN	MPI_INIT(IERROR) INTEGER IERROR

MPI_Comm_rank

To identify different processes the rank is used. Rank is in fact the name of the process. The rank is returned by the routine MPI_Comm_rank.

C	MPI_Comm_rank(MPI_Comm comm, int *rank) MPI_Comm_rank (comm,&rank)
C++ (in the MPI:: namespace)	int Comm::Get_rank() const
FORTAN	MPI_COMM_RANK(COMM, IRANK, IERROR) INTEGER COMM, IRANK, IERROR

The rank of the process is unique inside communicator.

MPI_Comm_size

The number of processes contained within a communicator is obtained by calling MPI_Comm_size.

C	MPI_Comm_size(MPI_Comm comm, int *size)
C++ (in the MPI:: namespace)	int Comm::Get_size() const
FORTAN	MPI_COMM_SIZE(COMM, ISIZE, IERRO INTEGER COMM, ISIZE, IERROR

MPI_Finalize

Exiting MPI routine must be called by all processes.

C	int MPI_Finalize(int *flag) int MPI_Finalize(void)
C++ (in the MPI:: namespace)	bool MPI::Finalize()
FORTAN	MPI_FINALIZE(IERROR) INTEGER IERROR

Example 1

The following simple code creates n processes but only one - with $rank$ equal 0 writes the message .

```
#include <stdio.h>
#include <mpi.h>
int main( int argc, char argv[] )
{
    int rank;
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    if (rank == 0) printf("Starting program ");
    MPI_Finalize();
    return 0;
}
```

Example 2

In the example each process writes it's own message with own rank.

```
/ * "Hello, parallel worlds!" * /
#include <stdio.h>
#include <mpi.h>
int main( int argc, char *argv[] )
{
    int myrank;
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
    printf("Hello, parallel worlds! It's me: processor %d\n", myrank );
    MPI_Finalize();
    return 0;
}
```

Example 2 cont.

You can expect to see something like this when you execute the program on 4 processors:

Hello, parallel worlds! It's me: processor 1!
Hello, parallel worlds! It's me: processor 3!
Hello, parallel worlds! It's me: processor 4!

Hello, parallel worlds! It's me: processor 2!

Example 3

In the example in C++ each process writes it's own message with own rank.

```
#include <iostream>
#include <mpi.h>
using namespace std;
int main (int argc, char *argv[])
{
int rank, size;
 MPI::Init(argc,argv);
rank = MPI::COMM_WORLD.Get_rank();
size = MPI::COMM_WORLD.Get_size();
cout<<"Hello, world!. I am " <<rank<<" of "<< size<<"\n";
 MPI::Finalize();
return 0; }
```

Communication

There are several **types of message passing**

- point-to point communication,
- collective communication,
- broadcasting.

Point-to-point communication

The point-to-point communication is the simplest form of message passing when one process sends a message to another one. There are different types of point-to-point communication:

- **synchronous send** - provide information about the completion of the message,
- **asynchronous send** - only know when the message has left,
- **blocking operations**:
  - relate to when the operation has been completed,
  - only return from the subroutine call when the operation has been completed,

Point-to-point communication

- **non-blocking operations**:
  - return strait away and allow the sub-program to continue to perform other work - at some time later the subprogram can test or wait for the completion of the non-blocking operation,
  - all non-blocking operations should have matching wait operations - some systems cannot free resources until wait has been called,
  - a non-blocking operation immediately followed by a matching wait is equivalent to a blocking operation,
  - non-blocking operations are not the same as sequential subroutine calls as the operation continues after the call has returned.

Collective communication

Collective communication routines are higher level routines involving several processes at a time. It can be built out of point-to-point communications

Collective communications contain the following operations:

- **Barriers** - synchronize processes,
- **Broadcast** - a one-to-many communication,
- **Reduce operations** - Combine data from several processes to produce a single result.

Communication modes

There are different type of **communication modes**:

- **Synchronous send** - only completes when the receive has completed: **MPI_SSEND**,
- **Buffered send** - always completes (unless an error occurs), irrespective of receiver **MPI_BSEND**,
- **Standard send** - aither synchronous or buffered **MPI_SEND**,
- **Ready send** - always completes (unless an error occurs), irrespective or whether the receive has completed **MPI_RSEND**,
- **Receive** - completes when a message has arrived **MPI_RECV**.

Communication - rules

Some rules should be taken under consideration when the communication is designed:

- a sub-program needs to be connected to a message passing system,
- messages need to have addresses to be sent to,
- the receiving process is capable of dealing with messages it is sent,
- MPI_SEND / MPI_RECV form the blocked communication - the processes wait until the communication is completed.

Message order is preserved - messages do not overtake each other.

This is true even for non-synchronous sends.

Messsages in MPI

Messages are packets of data moving between subprograms. The message passing system has the following information:

- sending processor,
- source location,
- data type,
- data length,
- receiving processor(s),
- destination location,
- destination size.

The receiving process is capable of dealing with messages which are sent.

MPI_Send

To communicate a sub-program needs to be connected to a message passing system. Messages need to have addresses to be sent to. Standard MPI_Send in C has the following arguments:

```
int MPI_Send(
        void *buf,                  // buffer
        int count,                  // number of sending data
        MPI_Datatype datatype,      // type of the sending data
        int dest,                   // rank of the destination process
        int tag,                    // the number of the message sendir
        MPI_Comm comm               // the name of the communicator
```

MPI_Send

C	int MPI_Send(void* buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm)
C++ (in the MPI:: namespace)	void Comm::Send(const void* buf, int coun const Datatype& datatype, int dest, int tag const
FORTRAN	MPI_SEND(BUF, COUNT, DATATYPE, DEST, TAG, COMM, IERROR) <type> BUF(*) INTEGER COUNT, DATATYPE, DEST, TAG, COMM, IERROR

Example

The standard send using for sending from buffer msg number $count$ data of type MPI_DOUBLE to process number $dest$ in the communicator MPI_COMM_WORLD:

```
MPI_Send( msg,
          count,
          MPI_DOUBLE,
          dest, tag,
          MPI_COMM_WORLD);
```

The synchronous send have the same parameters as the standard send.

Example

In Fortran the additional parameter IERROR is needed. The call can be like the following:

CALL MPI_SEND(BUF, COUNT, DATATYPE, DEST, TAG, COMM,

IERROR)

MPI_Recv

MPI_Send/MPI_Recv form the blocked communication - the processes wait until the communication is completed.

C	int MPI_Recv(void* buf, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI_Status *status)
C++ (in the MPI:: namespace)	void Comm::Recv(void* buf, int count, const Datatype& datatype, int source, int tag, Status& status) const void Comm::Recv(void* buf, int count, const Datatype& datatype, int source, int tag) const

MPI_Recv

FORTRAN	MPI_RECV(BUF, COUNT, DATATYPE, SOURCE, TAG, COMM, STATUS, IERROR) <type> BUF(*) INTEGER COUNT, DATATYPE, SOURCE, TAG, COMM, STATUS(MPI_STATUS_SIZE), IERROR

MPI_Recv

Arguments:

OUT	buf	initial address of receive buffer (choice)
IN	count	number of elements in receive buffer (integer)
IN	datatype	datatype of each receive buffer element (handle)
IN	source	rank of source (integer)
IN	tag	message tag (integer)
IN	comm	communicator (handle)
OUT	status	status object (Status)

MPI_Recv

- A sub-program needs to be connected to a message passing system.
- Messages need to have addresses to be sent to.
- The receiving process is capable of dealing with messages it is sent.
- MPI_Send/MPI_Recv form the blocked communication - the processes wait until the communication is completed.

Status in MPI_Recv

Communication envelope is returned from MPI_Recv as status.
Information includes:

- source status: MPI_SOURCE
- tag status: MPI_TAG
- count: MPI_GET_count

C	int MPI_Get_status(MPI_Status status, MPI_Datatype datatype, int *count) int MPI_Request_get_status(MPI_Request request, int *flag, MPI_Status *status)
C++	bool MPI::Request::Get_status(MPI::Status& status) const bool MPI::Request::Get_status() const

Success in communication

For a communication to succeed the following conditions should be fulfilled:

- Sender must specify a valid destination rank
- Receiver must specify a valid source rank
- The communicator must be the same
- Tags must match
- Message types must match
- Receiver buffer must be large enough.

Wildcarding

- Receiver can wildcard.
- To receive from any source the MPI_ANY_SOURCE and to receive with any tag: MPI_ANY_TAG should be use.
- Actual source and tag are returned in the receiver's status parameter.

Buffering

A user may specify a buffer to be used for buffering messages sent in buffered mode by calling MPI_Buffer_attach.

Buffering is done by the sender.

C	int MPI_Buffer_attach(void* buffer, int size)
C++ (in the MPI::namespace)	void Attach_buffer(void* buffer, int size)
FORTRAN	MPI_BUFFER_ATTACH(BUFFER, SIZE, IERROR) <type> BUFFER(*) INTEGER SIZE, IERROR

MPI_Buffer_attach

Arguments:

IN	buffer	initial buffer address (choice)
IN	size	buffer size, in bytes (integer)

- MPI_Buffer_attach(...) provides to MPI a buffer in the user's memory to be used for buffering outgoing messages.
- The buffer is used only by messages sent in buffered mode.
- Only one buffer can be attached to a process at a time.

MPI_Buffer_dettach

The routine MPI_Buffer_detach detaches the buffer currently associated with MPI. The call returns the address and the size of the detached buffer.

C	int MPI_Buffer_detach(void* buffer_addr, int* size)
C++ (in the MPI::namespace)	void Dettach_buffer(void* buffer, int size)
FORTRAN	MPI_BUFFER_DETACH(BUFFER, SIZE, IERROR) <type> BUFFER(*) INTEGER SIZE, IERROR

Example

```
/* Example calls to attach and detach buffers */
#define BUFFSIZE 10000
int size
char *buff;
MPI_Buffer_attach( malloc(BUFFSIZE), BUFFSIZE);
/* a buffer of 10000 bytes can now be used by MPI_Bsend */
MPI_Buffer_detach( &buff, &size);
/* Buffer size reduced to zero */
MPI_Buffer_attach( buff, size);

/* Buffer of 10000 bytes available again */
```

MPI_BSEND

Arguments of MPI_BSEND (buf, count, datatype, dest, tag, comm):

IN	buf	initial address of send buffer (choice)
IN	count	number of elements in send buffer (integer)
IN	datatype	datatype of each send buffer element (handle)
IN	dest	rank of destination (integer)
IN	tag	message tag (integer)
IN	comm	communicator (handle)

MPI_BSEND

C	int MPI_Bsend(void* buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm)
C++ (in the MPI:: namespace)	void Comm::Bsend(const void* buf, int count, const Datatype& datatype, int dest, int tag) const
FORTRAN	MPI_BSEND(BUF, COUNT, DATATYPE, DEST, TAG, COMM, IERROR) <type> BUF(*) INTEGER COUNT, DATATYPE, DEST, TAG, COMM, IERROR

Non-blocking communication

To avoid deadlock in non-blocking communication separate communication into three phases:

- Initiate non-blocking communication,
- Do some work,
- Wait for non-blocking communication to complete.

For non-blocking communication the **handlers** are used. A request handle is allocated when the communication is initiated.

Non-blocking synchronous send:
Int MPI_Isend(void* buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm, MPI_handle handle)

MPI_Isend

C	int MPI_Isend(void* buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm, MPI_handle handle)
C++ (in the MPI:: namespace)	Request Comm::Isend(void* buf, int count, const Datatype& datatype, int dest, int tag, int handle) const
FORTRAN	MPI_ISEND(BUF, COUNT, DATATYPE, DEST, TAG, COMM, HANDLE, IERROR) <type> BUF(*) INTEGER COUNT, DATATYPE, DEST, TAG, COMM, HANDLE, IERROR

MPI_Irecv

C	int MPI_Irecv(void* buf, int count, MPI_Datatype datatype, int source, int tag MPI_Comm comm, MPI_Request *reques
C++ (in the MPI:: namespace)	Request Comm::Irecv(void* buf, int count, const Datatype& datatype, int source, int tag) const
FORTRAN	MPI_IRECV(BUF, COUNT, DATATYPE, SOURCE, TAG, COMM, REQUEST, IERROR) <type> BUF(*) INTEGER COUNT, DATATYPE, SOURCE, TAG, COMM, REQUEST, IERROR

MPI_Irecv

Arguments:

OUT	buf	initial address of receive buffer (choice)
IN	count	number of elements in receive buffer (integer)
IN	datatype	datatype of each receive buffer element (handle)
IN	source	rank of source (integer)
IN	tag	message tag (integer)
IN	comm	communicator (handle)
OUT	request	communication request (handle)

The operations return straight away and allow the sub-program to continue to perform other work. At some time later the subprogram can **test** or **wait** for the completion of the non-blocking operation.

Non-blocking modes

Communication modes for non-blocking operations is the same as for the blocking. In addition a prefix of I (for immediate) indicates that the call is nonblocking.

- Synchronous send MPI_ISSEND,
- Buffered send MPI_IBSEND,
- Standard send MPI_ISEND,
- Ready send MPI_IRSEND,
- Receive MPI_IRECV.

Completion

Completion is checked by: int MPI_Wait(...)

C	int MPI_Wait(MPI_Request *request, MPI_Status *status)
C++ (in the MPI:: namespace)	void Request::Wait(Status& status) void Request::Wait()
FORTRAN	MPI_WAIT(REQUEST, STATUS, IERROR) INTEGER REQUEST, STATUS(MPI_STATUS_SIZE), IERROR

Arguments:

INOUT	request	request (handle)
OUT	status	status object (Status)

Example

```
#include <stdio.h>
int main(argc,argv)
int argc;
char argv[];
{ int numtasks, rank, next, prev, buf[2], tag1=1, tag2=2;
MPI_Request reqs[4];
MPI_Status stats[2];
MPI_Init(&argc,&argv);
MPI_Comm_size(MPI_COMM_WORLD, &numtasks);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
prev = rank-1;if (rank == 0) prev = numtasks - 1;
next = rank+1; if (rank == (numtasks - 1)) next = 0;
```

Example

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

Completion

MPI_Test() tests for completionof either one or none of the operations associated with active handles.

Arguments of MPI_Test:

INOUT	request	communication request (handle)
OUT	flag	true if operation completed (logical)
OUT	status	status object (Status)

MPI_Test

C	int MPI_Test(MPI_Request *request, int *flag, MPI_Status *status)
C++ (in the MPI:: namespace)	bool Request::Test(Status& status) bool Request::Test()
FORTRAN	MPI_TEST(REQUEST, FLAG, STATUS, IERROR) LOGICAL FLAG INTEGER REQUEST, STATUS(MPI_STATUS_SIZE), IERROR

Multiple completion

Multiple completions can be tested by the routines:

int MPI_Waitany(int count, MPI_Request *array_of_requests, int *index, MPI_Status *status)
int MPI_Testany(...)

int MPI_Waitall(int count, MPI_Request *array_of_requests, MPI_Status *array_of_statuses)
int MPI_Testall(...)

int MPI_Waitsome(int incount, MPI_Request *array_of_requests, int *outcount, int *array_of_indices, MPI_Status *array_of_statuses)
int MPI_Testsome(...)

PROBE

MPI_IPROBE returns $flag = true$ if there is a message that can be received and that matches the pattern specified by the arguments source, tag, and comm.

C	int MPI_Probe(int source, int tag, MPI_Comm comm, MPI_Status *status)
C++ (in the MPI:: namespace)	void Comm::Probe(int source, int tag, Status$\&$ status) const void Comm::Probe(int source, int tag) cons
FORTRAN	MPI_PROBE(SOURCE, TAG, COMM, STATUS, IERROR) INTEGER SOURCE, TAG, COMM, STATUS(MPI_STATUS_SIZE), IERROR

IPROBE

Arguments:

IN	source	source rank, or MPI_ANY_SOURCE (integer)
IN	tag	tag value, or MPI_ANY_TAG (integer)
IN	comm	communicator (handle)
OUT	status	status object (Status)

MPI_PROBE behaves like MPI_IPROBE except that it is a blocking call that returns only after a matching message has been found.

CANCEL

A call to MPI_CANCEL marks for cancellation a pending, nonblocking communication operation (send or receive).

The cancel call is local.

Arguments:

IN request communication request (handle)

C	int MPI_Cancel(MPI_Request *request)
C++ (in the MPI:: namespace)	void Request::Cancel() const
FORTRAN	MPI_CANCEL(REQUEST, IERROR) INTEGER REQUEST, IERROR

MPI_Test_cancelled

MPI_Test_cancelled returns flag = true if the communication associated with the status object was canceled successfully.

Arguments:

IN	status	status object (Status)
OUT	flag	(logical)

C	int MPI_Test_cancelled(MPI_Status *statu int *flag)
C++ (in the MPI:: namespace)	bool Status::Is_cancelled() const
FORTRAN	MPI_TEST_CANCELLED(STATUS, FLAG IERROR) LOGICAL FLAG INTEGER STATUS(MPI_STATUS_SIZE),

Blocking and non-blocking

Comparing blocking and non-blocking the following facts should be noticed:

- Send and receive can be blocking or non-blocking,
- A blocking send can be used with a non-blocking receive and vice-versa,
- Non-blocking sends can use any mode: synchronous, buffered, standard or ready,
- Synchronous mode affects completion, not initiation.

Persistent communication

Often a communication with the same argument list is repeatedly executed within the inner loop of a parallel computation. In such a situation, it may be possible to optimize the communication by binding the list of the communication arguments to a persistent communication request once and, then, repeatedly using the request to initiate and complete messages. Arguments:

IN	buf	initial address of send buffer (choice)
IN	count	number of elements sent (integer)
IN	datatype	type of each element (handle)
IN	dest	rank of destination (integer)
IN	tag	message tag (integer)
IN	comm	communicator (handle)
OUT	request	communication request (handle)

Persistent communication

C	int MPI_Send_init(void* buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm, MPI_Request *reques
C++ (in the MPI:: namespace)	Prequest Comm::Send_init(const void* buf int count, const Datatype& datatype, int dest, int tag) const
FORTRAN	MPI_SEND_INIT(BUF, COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR) <type> BUF(*) INTEGER REQUEST, COUNT, DATATYPE DEST, TAG, COMM, REQUEST, IERROR

Persistent communication

MPI_Send_init() creates a persistent communication request for a standard mode send operation, and binds to it all the arguments of a send operation.

C	int MPI_Recv_init(void* buf, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI_Request *request) int MPI_Bsend_init(void* buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm, MPI_Request *request) int MPI_Ssend_init(void* buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm, MPI_Request *request) int MPI_Rsend_init(void* buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm, MPI_Request *request)

Persistent communication

Arguments of MPI_SSEND_INIT(buf, count, datatype, dest, tag, comm, request) :

IN	buf	initial address of send buffer (choice)
IN	count	number of elements sent (integer)
IN	datatype	type of each element (handle)
IN	dest	rank of destination (integer)
IN	tag	message tag (integer)
IN	comm	communicator (handle)
OUT	request	communication request (handle)

Persistent communication

The standard send is similar.

C	int MPI_Ssend_init(void* buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm, MPI_Request *request)
C++ (in the MPI:: namespace)	Prequest Comm::Ssend_init(const void* buf, int count, const Datatype& datatype, int dest, int tag) const
FORTRAN	MPI_SSEND_INIT(BUF, COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR) <type> BUF(*) INTEGER COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR

Persistent communication

A persistent communication request is inactive after it was created. A communication (send or receive) that uses a persistent request is initiated by the function MPI_START.

C	int MPI_Start(MPI_Request *request)
C++ (in the MPI:: namespace)	void Prequest::Start()
FORTRAN	MPI_START(REQUEST, IERROR) INTEGER REQUEST, IERROR

Arguments :

INOUT request communication request (handle)

C	int MPI_Startall(int count, MPI_Request *array_of_requests)

Send-receive

The send-receive operations combine in one call the sending of a message to one destination and the receiving of another message, from another process.

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

Multiple communications test or wait for completion of one message or of all messages or of as many messages as possible.

MPI_Request_free

A successful return from a blocking communication operation or from MPI_WAIT or MPI_TEST tells the user that the buffer can be reused and means that the communication is completed by the user, but does not guarantee that the local process has no more work to do.
A successful return from MPI_REQUEST_FREE with a request handle generated by an MPI_ISEND nullifies the handle but provides no assurance of operation completion.

The MPI_ISEND is complete only when it is known by some means that a matching receive has completed.

MPI_Request_free

MPI_FINALIZE guarantees that all local actions required by communications the user has completed will, in fact, occur before it returns.

MPI_FINALIZE guarantees nothing about pending communications that have not been completed (completion is assured only by MPI_WAIT, MPI_TEST, or MPI_REQUEST_FREE combined with some other verification of completion).

MPI_Request_free

Arguments : INOUT request communication request (handle)

C	int MPI_Request_free (MPI_Request *request)
C++ (in the MPI:: namespace)	void Request::Free()
FORTRAN	MPI_REQUEST_FREE(REQUEST, IERROR) INTEGER REQUEST, IERROR

Example

The following is correct.

```
switch(rank) {
  case 0:
    MPI_Isend();
    MPI_Request_free();
    MPI_Barrier(); break;
  case 1:
    MPI_Recv();
    MPI_Barrier(); break;
} ;
MPI_Finalize();

exit();
```

Example

This program is erroneous and its behavior is undefined:

```
switch(rank) {
  case 0:
    MPI_Isend();
    MPI_Request_free();
    break;
  case 1:
    MPI_Recv();
    break;
} ;
MPI_Finalize(); exit();
```

Thank you for your attention!

Any questions are welcome.