Introduction to Message Passing Interface
Lesson 6.

Adriano FESTA

Dipartimento di Ingegneria e Scienze dell'Informazione e Matematica, L'Aquila

DISIM, L'Aquila, 29.04.2019

UNIVERSITÀ
DEGLI STUDI
DE L'AQUILA

adriano.festa@univaq.it

Class outline

- General principles
- Message Passing Interface - MPI
- Point-to-point communication
- Collective communication
- Communicators
- Datatypes
- Topologies
- Inter-communicators
- Profiling

Principles of Message-Passing Programming

- The logical view of a machine supporting the message-passing paradigm consists of p processes, each with its own exclusive address space.
- Each data element must belong to one of the partitions of the space; hence, data must be explicitly partitioned and placed.
- All interactions (read-only or read/write) require cooperation of two processes – the process that has the data and the process that wants to access the data.
- These two constraints, while onerous, make underlying costs very explicit to the programmer.

Principles of Message-Passing Programming

- Message-passing programs are often written using the *asynchronous* or *loosely synchronous* paradigms.
- In the asynchronous paradigm, all concurrent tasks execute asynchronously.
- In the loosely synchronous model, tasks or subsets of tasks synchronize to perform interactions. Between these interactions, tasks execute completely asynchronously.
- Most message-passing programs are written using the *single program multiple data* (SPMD) model.

The Building Blocks: Send and Receive Operations

- The prototypes of these operations are as follows:
  send(void *sendbuf, int nelems, int dest)
  receive(void *recvbuf, int nelems, int source)

Consider the following code segments:

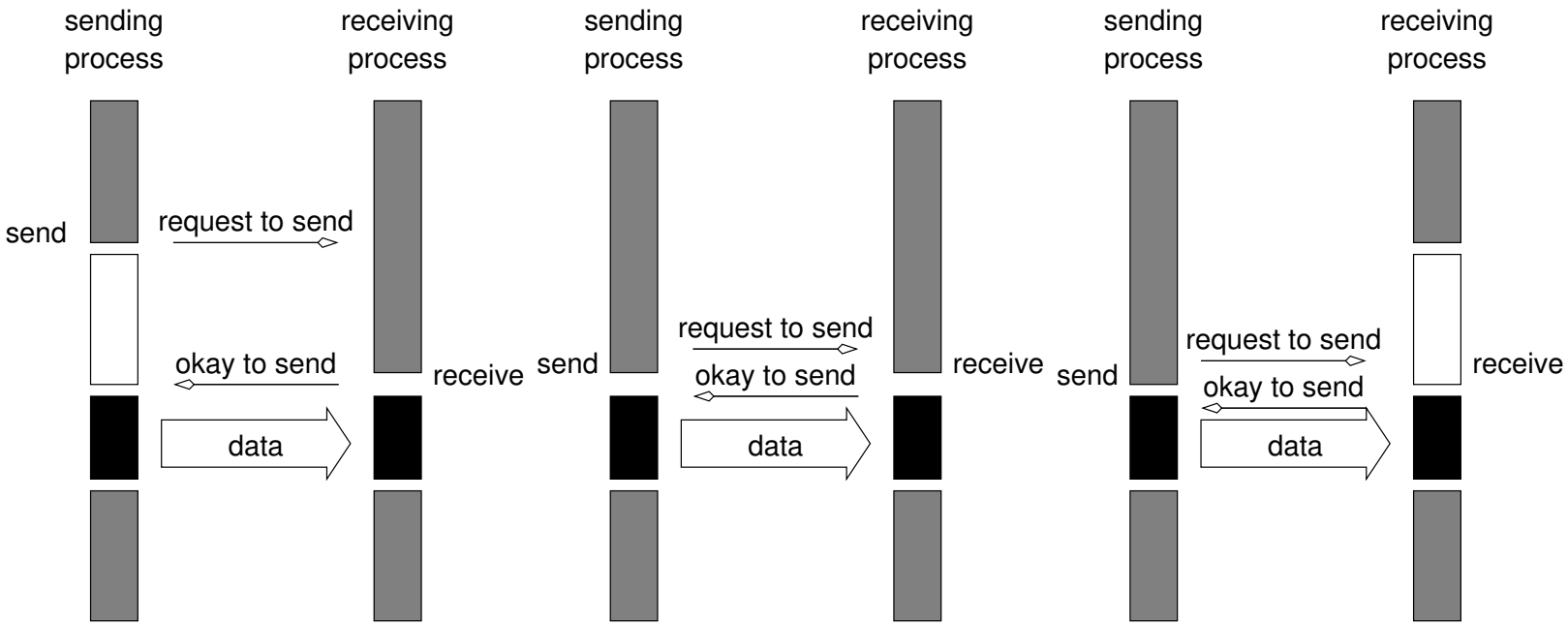
```
a = 100; receive(a, 1, 0) send(a, 1, 1); printf("a = 0;
```

The semantics of the send operation require that the value received by process P1 must be 100 as opposed to 0.
This motivates the design of the send and receive protocols.

Non-Buffered Blocking Message Passing Operations

- A simple method for forcing send/receive semantics is for the send operation to return only when it is safe to do so.
- In the non-buffered blocking send, the operation does not return until the matching receive has been encountered at the receiving process.
- Idling and deadlocks are major issues with non-buffered blocking sends.
- In buffered blocking sends, the sender simply copies the data into the designated buffer and returns after the copy operation has been completed. The data is copied at a buffer at the receiving end as well.
- Buffering alleviates idling at the expense of copying overheads.

Non-Buffered Blocking Message Passing Operations



(a) Sender comes first; idling at sender

(b) Sender and receiver come at about the same time; idling minimized

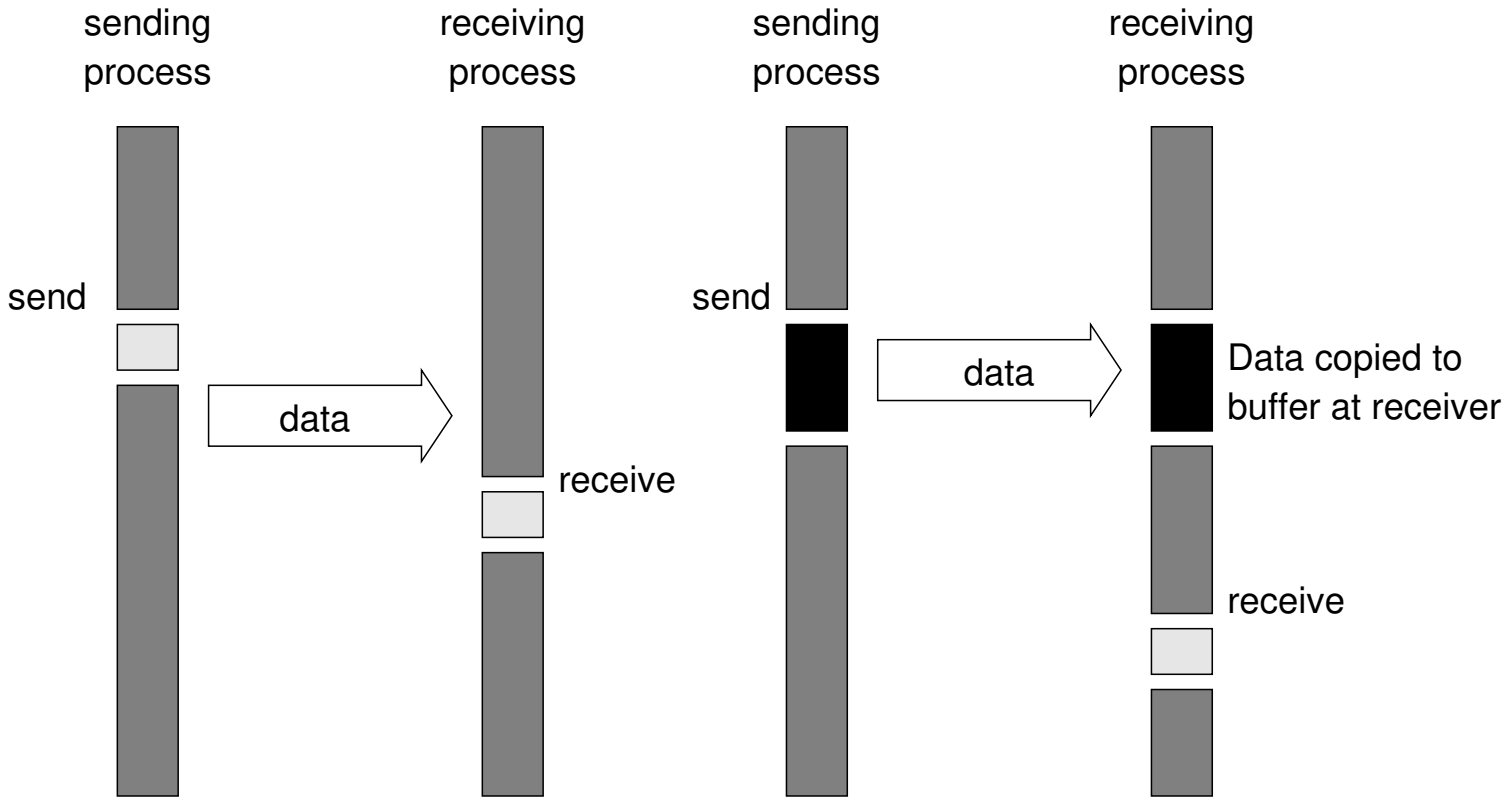
(c) Receiver comes first; idling at receiver

Handshake for a blocking non-buffered send/receive operation. It is easy to see that in cases where sender and receiver do not reach communication point at similar times, there can be considerable idling overheads.

Buffered Blocking Message Passing Operations

- A simple solution to the idling and deadlocking problem outlined above is to rely on buffers at the sending and receiving ends.
- The sender simply copies the data into the designated buffer and returns after the copy operation has been completed.
- The data must be buffered at the receiving end as well.
- Buffering trades off idling overhead for buffer copying overhead.

Buffered Blocking Message Passing Operations



Blocking buffered transfer protocols: (a) in the presence of communication hardware with buffers at send and receive ends; and (b) in the absence of communication hardware, sender interrupts receiver and deposits data in buffer at receiver end.

Buffered Blocking Message Passing Operations

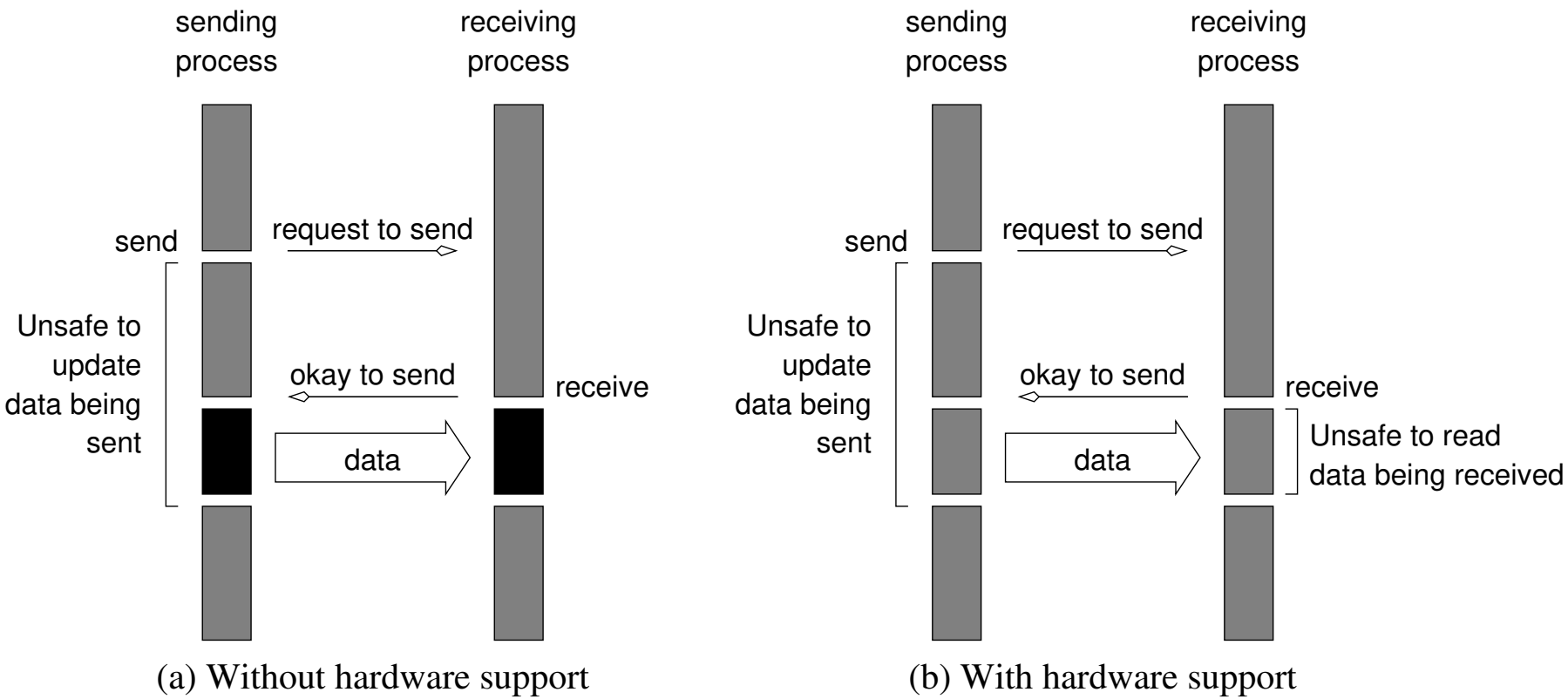
Deadlocks are still possible with buffering since receive operations block.

```
receive(a, 1, 1); receive(a, 1, 0); send(b, 1, 1); send(b, 1, 0);
```

Non-Blocking Message Passing Operations

- The programmer must ensure semantics of the send and receive.
- This class of non-blocking protocols returns from the send or receive operation before it is semantically safe to do so.
- Non-blocking operations are generally accompanied by a `check-status` operation.
- When used correctly, these primitives are capable of overlapping communication overheads with useful computations.
- Message passing libraries typically provide both blocking and non-blocking primitives.

Non-Blocking Message Passing Operations

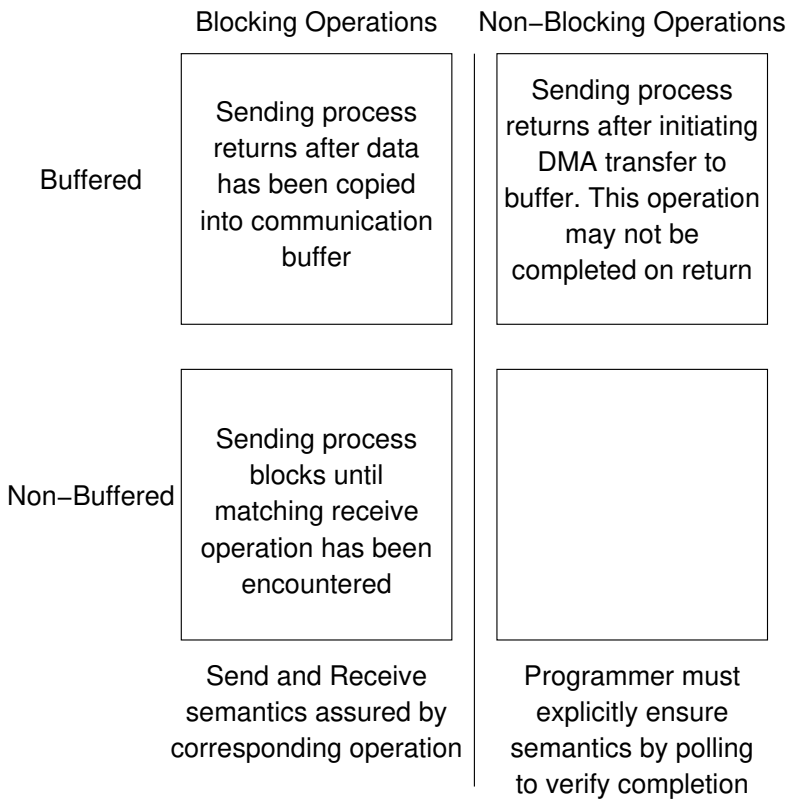


(a) Without hardware support

(b) With hardware support

Non-blocking non-buffered send and receive operations (a) in absence of communication hardware; (b) in presence of communication hardware.

Send and Receive Protocols



Space of possible protocols for send and receive operations.

Message Passing Interface (MPI)

- A message-passing library specification
- For parallel computers, clusters, and heterogeneous networks
- Designed to aid the development of portable parallel software libraries
- Designed to provide access to advanced parallel hardware for end users, library writers, tool developers
- MPI-1 standard widely accepted by vendors and programmers
- MPI implementations available on most modern platforms
- Huge number of MPI applications deployed
- Several tools exist to trace and tune MPI applications
- MPI provides rich set of functionality to support library writers, tools developers and application programmers

MPI Salient Features

- Point-to-point communication
- Collective communication on process groups
- Communicators and groups for safe communication
- User defined datatypes
- Virtual topologies
- Support for profiling

A First MPI Program (Fortran)

C version

```
#include <stdio.h>
#include <mpi.h>
main( int argc, char **argv )
{
MPI_Init ( &argc, &argv );
printf ( \Hello World!\n" );
MPI_Finalize ( );
}
```

Fortran version

```
program mpi_example_one
use mpi
integer ierr
call MPI_INIT( ierr )
print *, 'Hello world!'
call MPI_FINALIZE( ierr )
end program
```

Starting the MPI Environment

MPI_INIT( )
Initializes MPI environment. This function must be called and must be the first MPI function called in a program.

Syntax

- int *MPI_Init*(*&argc*, *&argv*); C
- *MPI_INIT* (IERROR) INTEGER IERROR; Fortran

Exiting the MPI Environment

MPI_FINALIZE()
Cleans up all MPI state. Once this routine has been called, no MPI routine (even *MPI_INIT*) may be called.

Syntax

- int *MPI_Finalize*(); C
- *MPI_FINALIZE* (IERROR) INTEGER IERROR; Fortran

Finding Out About the Parallel Environment

The main questions in a parallel program are:

- How many processes are there?
- Who am I?
- "How many" is answered with the function call *MPI_COMM_SIZE()*
- "Who am I" is answered with the function call *MPI_COMM_RANK()*
- The rank is a number between zero and (size - 1)

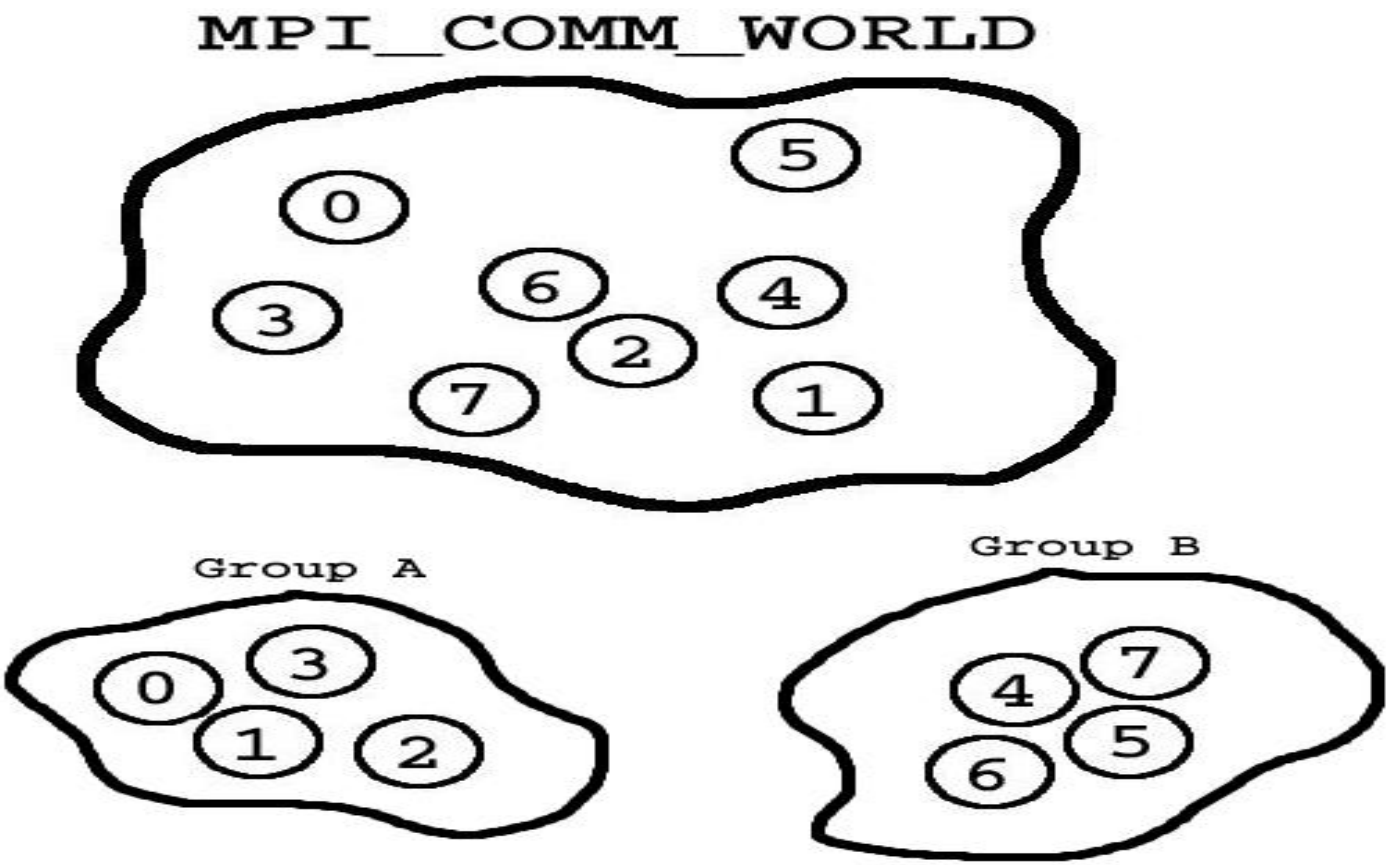
Example 1 (Fortran)

```
program mpirank
use mpi
integer::rank, size, ierr
call MPI_INIT( ierr )
call MPI_COMM_RANK( MPI_COMM_WORLD, rank, ierr )
call MPI_COMM_SIZE ( MPI_COMM_WORLD, size, ierr )
print *, 'Process up rank and size ',rank, size
call MPI_FINALIZE( ierr )
end program
```

Communicator

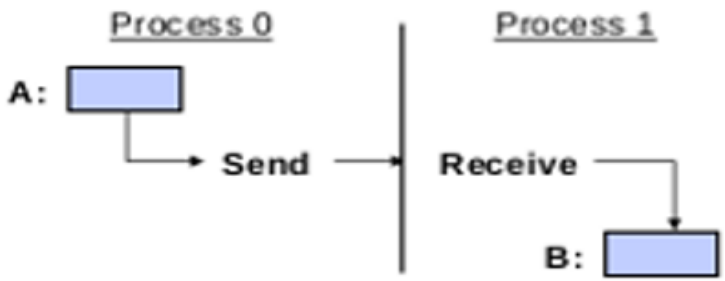
- Communication in MPI takes place with respect to communicators
- An MPI process can query a communicator for information about the group, with *MPI_COMM_SIZE* and *MPI_COMM_RANK*.
- *MPI_COMM_RANK*(*comm*, *rank*), *MPI_COMM_RANK* returns in rank the rank of the calling process in the group associated with the communicator comm.
- *MPI_COMM_SIZE* (comm, size),*MPI_COMM_SIZE* returns in size the number of processes in the group associated with the communicator comm.

Communicator



Point to Point communications

Basic message passing process



Questions

- To whom is data sent?
- Where is the data?
- What type of data is sent?
- How much data are sent?
- How does the receiver identify it?

Message Organization in MPI

Message is divided into data and envelope

Data

- buffer
- count
- datatype

Envelope

- process identifier (source/destination rank)
- message tag
- communicator

MPI and Fortran data type

MPI Fortran Datatypes

MPI FORTRAN	FORTRAN datatypes
MPI_INTEGER	INTEGER
MPI_REAL	REAL
MPI_REAL8	REAL*8
MPI_DOUBLE_PRECISION	DOUBLE PRECISION
MPI_COMPLEX	COMPLEX
MPI_LOGICAL	LOGICAL
MPI_CHARACTER	CHARACTER
MPI_BYTE	
MPI_PACKED	

MPI and C data type

MPI C Datatypes

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_LONG	unsigned long_int
MPI_UNSIGNED	unsigned int
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_LONG_DOUBLE	long double
MPI_BYTE	
MPI_PACKED	

Blocking and Non blocking

- Blocking operation: An MPI communication operation is blocking, if return of control to the calling process indicates that all resources, such as buffers, specified in the call can be reused, e.g., for other operations. In particular, all state transitions initiated by a blocking operation are completed before control returns to the calling process.
- Non blocking operation: An MPI communication operation is non blocking, if the corresponding call may return before all effects of the operation are completed and before the resources used by the call can be reused. Thus, a call of a non-blocking operation only starts the operation. The operation itself is com- pleted not before all state transitions caused are completed and the resources specified can be reused.

Synchronous and Asynchronous communication

The terms blocking and non-blocking describe the behavior of operations from the local view of the executing process, without taking the effects on other processes into account. But it is also useful to consider the effect of communication operations from a global viewpoint. In this context, it is reasonable to distinguish between synchronous and asynchronous communications:

- Synchronous communication: The communication between a sending process and a receiving process is performed such that the communication operation does not complete before both processes have started their communication operation. This means in particular that the completion of a synchronous send indicates not only that the send buffer can be reused, but also that the receiving process has started the execution of the corresponding receive operation.
- Asynchronous communication: Using asynchronous communication, the sender can execute its communication operation without any coordination with the receiving process.

Communication modes with MPI

- There are four communication modes provided by MPI: standard, synchronous, buffered and ready. The modes refer to four different types of send. It is not meaningful to talk of communication mode in the context of a receive. "Completion" of a send means by definition that the send buffer can safely be re-used. The standard, synchronous and buffered sends differ only in one respect: how completion of the send depends on the receipt of the message.
- Synchronous send : only completes when the receive has completed.
- Buffered send : always completes (unless an error occurs), irrespective of whether the receive has completed.
- Standard send: either synchronous or buffered.
- Ready send: always completes (unless an error occurs), irrespective of whether the receive has completed.
- Receive: completes when a message has arrived.

Standard send

The standard send completes once the message has been sent, which may or may not imply that the message has arrived at its destination. The message may instead lie "in the communications network" for some time. A program using standard sends should therefore obey various rules:

- It should not assume that the send will complete before the receive begins. For example, two processes should not use blocking standard sends to exchange messages, since this may on occasion cause deadlock.
- It should not assume that the send will complete after the receive begins. For example, the sender should not send further messages whose correct interpretation depends on the assumption that a previous message arrived elsewhere; it is possible to imagine scenarios (necessarily with more than two processes) where the ordering of messages is non-deterministic under standard mode.

Standard send

- In summary, a standard send may be implemented as a synchronous send, or it may be implemented as a buffered send, and the user should not assume either case.
- Processes should be eager readers, i.e. guarantee to eventually receive all messages sent to them, else the network may overload.
- If a program breaks these rules, unpredictable behaviour can result: programs may run successfully on one implementation of MPI but not on others, or may run successfully on some occasions and "hang" on other occasions in a non-deterministic way.

MPI Basic Send

The standard send has the following form
MPI_SEND(*buf*, *count*, *datatype*, *dest*, *tag*, *comm*) where

- buf is the address of the data to be sent.
- count is the number of elements of the MPI datatype which buf contains.
- datatype is the MPI datatype.
- dest is the destination process for the message. This is specified by the rank of the destination process within the group associated with the communicator comm.
- tag is a marker used by the sender to distinguish between different types of messages. Tags are used by the programmer to distinguish between different sorts of message.
- comm is the communicator shared by the sending and receiving processes. Only processes which have the same communicator can communicate.
- IERROR contains the return value of the Fortran version of the synchronous send.

MPI Synchronous Send

- If the sending process needs to know that the message has been received by the receiving process, then both processes may use synchronous communication.
- What actually happens during a synchronous communication is something like this: the receiving process sends back an acknowledgement (a procedure known as a `hand- shake' between the processes).
- This acknowledgement must be received by the sender before the send is considered complete.
- The standard send has the following form
  MPI_SSEND(*buf*, *count*, *datatype*, *dest*, *tag*, *comm*)

MPI Synchronous Send

- If a process executing a blocking synchronous send is "ahead" of the process executing the matching receive, then it will be idle until the receiving process catches up.
- Similarly, if the sending process is executing a non-blocking synchronous send, the completion test will not succeed until the receiving process catches up. Synchronous mode can therefore be slower than standard mode.
- Synchronous mode is however a safer method of communication because the communication network can never become overloaded with undeliverable messages.
- It has the advantage over standard mode of being more predictable: a synchronous send always synchronises the sender and receiver, whereas a standard send may or may not do so. This makes the behaviour of a program more deterministic. Debugging is also easier because messages cannot lie undelivered and "invisible" in the network.

MPI Buffered Send

- Buffered send guarantees to complete immediately, copying the message to a system buffer for later transmission if necessary. The advantage over standard send is predictability the sender and receiver are guaranteed not to be synchronised and if the network overloads, the behaviour is defined, namely an error will occur.
- Therefore a parallel program using buffered sends need only take heed of the rule on pages 17-18. The disadvantage of buffered send is that the programmer cannot assume any pre-allocated buffer space and must explicitly attach enough buffer space for the program with calls to *MPI_BUFFER_ATTACH*.
- Non-blocking buffered send has no advantage over blocking buffered send.
- To use buffered mode, the user must attach buffer space: *MPI_BUFFER_ATTACH*(*buffer*, *size*)

MPI Buffered Send

This specifies the array buffer of size bytes to be used as buffer space by buffered mode. Of course buffer must point to an existing array which will not be used by the programmer. Only one buffer can be attached per process at a time.
Buffer space is detached with: *MPI_BUFFER_DETACH*(*buffer*, *size*) Any communications already using the buffer are allowed to complete before the buffer is detached by MPI.
Often buffered sends and non-blocking communication are alternatives and each has pros and cons:

- buffered sends require extra buffer space to be allocated and attached by the user;
- buffered sends require copying of data into and out of system buffers while non-blocking communication does not;
- non-blocking communication requires more MPI calls to perform the same number of communications.

MPI Ready Send

A ready send, like buffered send, completes immediately. The communication is guaranteed to succeed normally if a matching receive is already posted. However, unlike all other sends, if no matching receive has been posted, the outcome is undefined. The sending process simply throws the message out onto the communication network and hopes that the receiving process is waiting to catch it. If the receiving process is ready for the message, it will be received, else the message may be silently dropped, an error may occur, etc.
The idea is that by avoiding the necessity for handshaking and buffering between the sender and the receiver, performance may be improved. Use of ready mode is only safe if the logical control flow of the parallel program permits it.
The ready send has a similar form to the standard send:
MPI_RSEND(*buf*, *count*, *datatype*, *dest*, *tag*, *comm*) Non-blocking ready send has no advantage over blocking ready send

MPI Basic Recv

The format of the standard blocking receive is:
MPI_RECV(*buf*, *count*, *datatype*, *source*, *tag*, *comm*, *status*) where

- buf is the address where the data should be placed once received (the receive buffer). For the communication to succeed, the receive buffer must be large enough to hold the message without truncation — if it is not, behaviour is undefined. The buffer may however be longer than the data received.
- count is the number of elements of a certain MPI datatype which buf can contain. The number of data elements actually received may be less than this.
- datatype is the MPI datatype for the message. This must match the MPI datatype specified in the send routine.
- source is the rank of the source of the message in the group associated with the communicator comm. Instead of prescribing the source, messages can be received from one of a number of sources by specifying a wildcard, *MPI_ANY_SOURCE*, for this argument.

MPI Basic Recv

- tag is used by the receiving process to prescribe that it should receive only a message with a certain tag. Instead of prescribing the tag, the wildcard *MPI_ANY_TAG* can be specified for this argument.
- comm is the communicator specified by both the sending and receiving process. There is no wildcard option for this argument.
- If the receiving process has specified wildcards for both or either of source or tag, then the corresponding information from the message that was actually received may be required. This information is returned in status, and can be queried using routines described later.
- IERROR contains the return value of the Fortran version of the standard receive. Completion of a receive means by definition that a message arrived i.e. the data has been received.

MPI communications notes

The word "blocking" means that the routines described above only return once the com- munication has completed. This is a non-local condition i.e. it might depend on the state of other processes. The ability to select a message by source is a powerful feature. For example, a source process might wish to receive messages back from worker proc- esses in strict order. Tags are another powerful feature. A tag is an integer labelling different types of message, such as "initial data", "client-server request", "results from worker". Note the difference between this and the programmer sending an inte- ger label of his or her own as part of the message — in the latter case, by the time the label is known, the message itself has already been read. The point of tags is that the receiver can select which messages it wants to receive, on the basis of the tag. Point-to-point communications in MPI are led by the sending process "pushing" mes- sages out to other processes — a process cannot "fetch" a message, it can only receive a message if it has been sent.

MPI communications notes

When a point-to-point communication call is made, it is termed posting a send or posting a receive, in analogy perhaps to a bulletin board. Because of the selection allowed in receive calls, it makes sense to talk of a send matching a receive. MPI can be thought of as an agency — processes post sends and receives to MPI and MPI matches them up.

Non Blocking Communication

The communications described so far are all blocking communications. This means that they do not return until the communication has completed (in the sense that the buffer can be used or re-used). Using blocking communications, a first attempt at a parallel algorithm for the one-dimensional smoothing might look like this:

```
for(iterations)
update all cells;

send  boundary values to neighbours;

receive halo values from neighbours;
```

This produces a situation akin to that shown in where each process sends a message to another process and then posts a receive. Assume the messages have been sent using a standard send. Depending on implementation details a standard send may not be able to complete until the receive has started. Since every process is sending and none is yet receiving, deadlock can occur and none of the communications ever complete.

Non Blocking Communication

There is a solution to the deadlock based on "red-black" communication in which "odd" processes choose to send whilst "even" processes receive, followed by a reversal of roles 1 — but deadlock is not the only problem with this algorithm. Communication is not a major user of CPU cycles, but is usually relatively slow because of the communication network and the dependency on the process at the other end of the communication. With blocking communication, the process is waiting idly while each communication is taking place. Furthermore, the problem is exacerbated because the communications in each direction are required to take place one after the other. The point to notice is that the non-boundary cells could theoretically be updated during the time when the boundary/halo values are in transit. This is known as latency hiding because the latency of the communications is overlapped with useful work. This requires a decoupling of the completion of each send from the receipt by the neighbour. Non-blocking communication is one method of achieving this.

Non Blocking Communication

In non-blocking communication the processes call an MPI routine to set up a communication cation (send or receive), but the routine returns before the communication has completed. The communication can then continue in the background and the process can carry on with other work, returning at a later point in the program to check that the communication has completed successfully. The communication is therefore divided into two operations: the initiation and the completion test. Non-blocking communication is analogous to a form of delegation the user makes a request to MPI for communication and checks that its request completed satisfactorily only when it needs to know in order to proceed.

The solution now looks like:

```
for(iterations) update boundary cells; initiate sending of boundary
values to neighbours; initiate receipt of halo values from neighbours;
update non-boundary cells; wait for completion of sending of boundary
values; wait for completion of receipt of halo values;
```

Note also that deadlock cannot occur and that communication in each direction can occur simultaneously. Completion tests are made when the halo data is required for the next iteration (in the case of a receive) or the boundary values are about to be updated again (in the case of a send).

Non Blocking Communication

The non-blocking routines have identical arguments to their blocking counterparts except for an extra argument in the non-blocking routines. This argument, request, is very important as it provides a handle which is used to test when the communication has completed.

- Standard send *MPI_ISEND*
- Synchronous send *MPI_ISSEND*
- Buffered send *MPI_BSEND*
- Ready send *MPI_RSEND*
- Receive *MPI_IRECV*

Non Blocking Communication

The sending process initiates the send using the following routine (in synchronous mode): *MPI_ISSEND*(*buf*, *count*, *datatype*, *dest*, *tag*, *comm*, *request*) It then continues with other computations which do not alter the send buffer. Before the sending process can update the send buffer it must check that the send has com- pleted using the routines described in Testing communications for completion (next pages).

Testing communications for completion

When using non-blocking communication it is essential to ensure that the communication has completed before making use of the result of the communication or reusing the communication buffer.
Completion tests come in two types:

- WAIT type these routines block until the communication has completed. They are useful when the data from the communication is required for the computa- tions or the communication buffer is about to be re-used. Therefore a non-blocking communication immediately followed by a WAIT-type test is equivalent to the corresponding blocking communication.
- TEST type these routines return a TRUE or FALSE value depending on whether or not the communication has completed. They do not block and are useful in situations where we want to know if the communication has completed but do not yet need the result or to re-use the communication buffer i.e. the process can usefully perform some other task in the meantime.

Testing a non-blocking communication for completion

The WAIT-type test is:
MPI_WAIT(*request*, *status*) This routine blocks until the communication specified by the handle request has completed. The request handle will have been returned by an earlier call to a non-blocking communication routine.
The TEST-type test is: *MPI_TEST*(*request*, *flag*, *status*) In this case the communication specified by the handle request is simply queried to see if the communication has completed and the result of the query (TRUE or FALSE) is returned immediately in flag.

Multiple Communications

In this case the routines test for the completion of all of the specified communications. The blocking test is as follows:

- *MPI_WAITALL*(*count*, *array_of_requests*, *array_of_statuses*) This routine blocks until all the communications specified by the request handles, array of requests, have completed. The statuses of the communications are returned in the array array of statuses and each can be queried in the usual way for the source and tag if required. There is also a TEST-type version which tests each request handle without blocking.
- *MPI_TESTALL*(*count*, *array_of_requests*, *flag*, *array_of_statuses*) If all the communications have completed, flag is set to TRUE, and information about each of the communications is returned in array ofstatuses. Otherwise flag is set to FALSE and array of statuses is undefined.

Completion of any of a number of communications

It is often convenient to be able to query a number of communications at a time to find out if any of them have completed . This can be done in MPI as follows:

- *MPI_WAITANY*(*count*, *array_of_requests*, *index*, *status*) *MPI_WAITANY* blocks until one or more of the communications associated with the array of request handles, array_of_requests, has completed. The index of the completed communication in the array of requests handles is returned in index, and its status is returned in status. Should more than one communication have completed, the choice of which is returned is arbitrary. It is also possible to query if any of the communications have completed without blocking.
- *MPI_TESTANY*(*count*, *array_of_requests*, *index*, *flag*, *status*) The result of the test (TRUE or FALSE) is returned immediately in flag. Otherwise behaviour is as for *MPI_WAITANY*.