

Message Passing with MPI

PPCES 2018

Joachim Protze / Marc-André Hermanns
IT Center / JARA-HPC

Slides by Hristo Iliev

■ Motivation

■ Part 1

- Concepts
- Point-to-point communication
- Non-blocking operations

■ Part 2

- Collective operations
- Communicators
- User datatypes

■ Part 3

- Hybrid parallelisation
- Common parallel patterns

■ Motivation

■ Part 1

- Concepts
- Point-to-point communication
- Non-blocking operations

■ Part 2

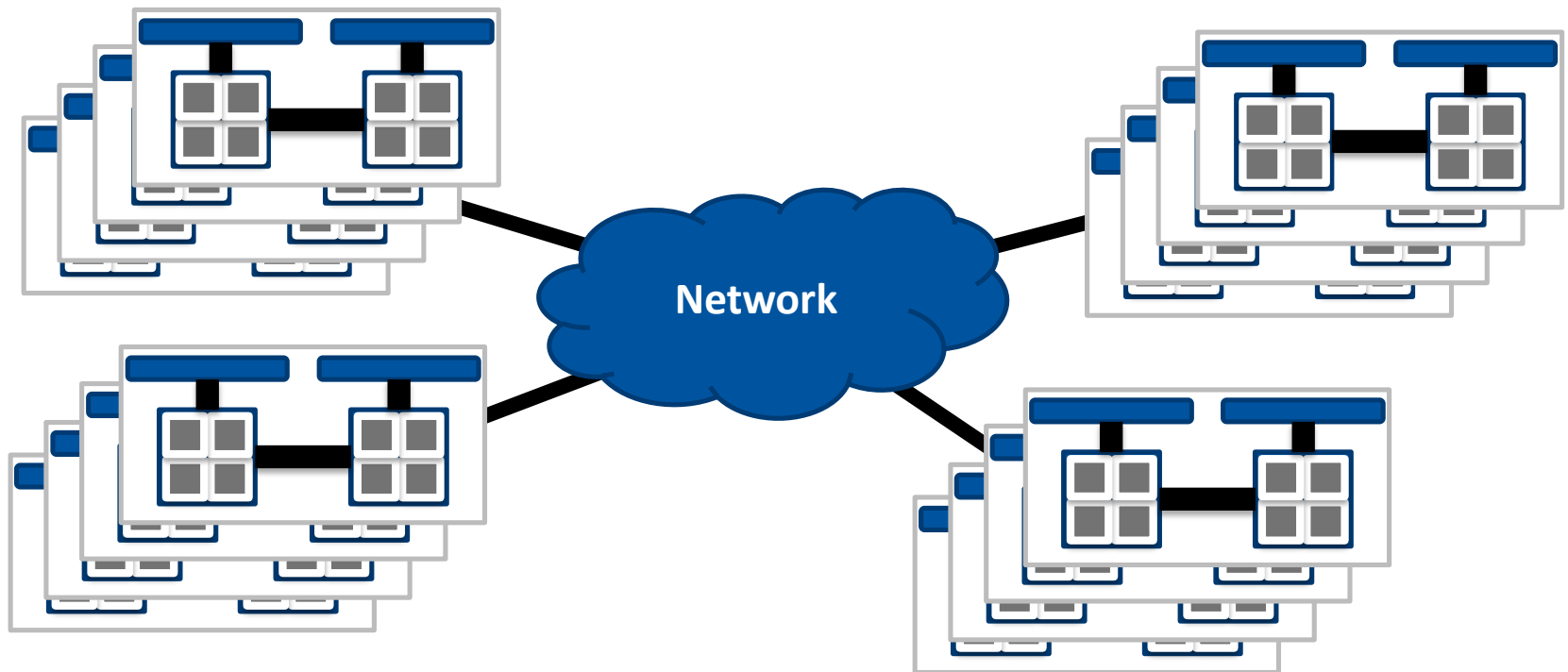
- Collective operations
- Communicators
- User datatypes

■ Part 3

- Hybrid parallelisation
- Common parallel patterns

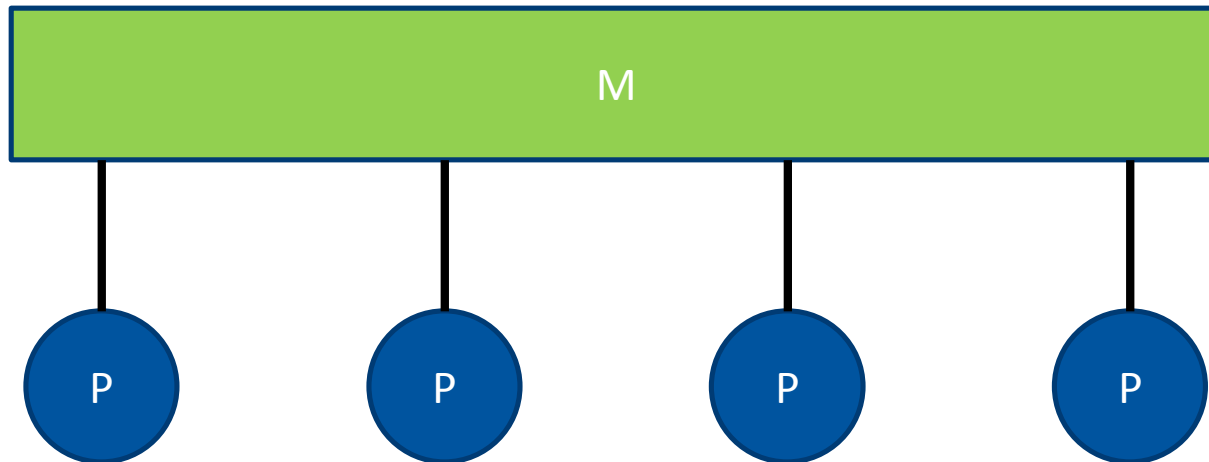
■ Clusters

- HPC market is at large dominated by distributed memory *multicomputers*: *clusters* and specialised *supercomputers*
- Nodes have no direct access to other nodes' memory and run a separate copy of the (possibly stripped down) OS



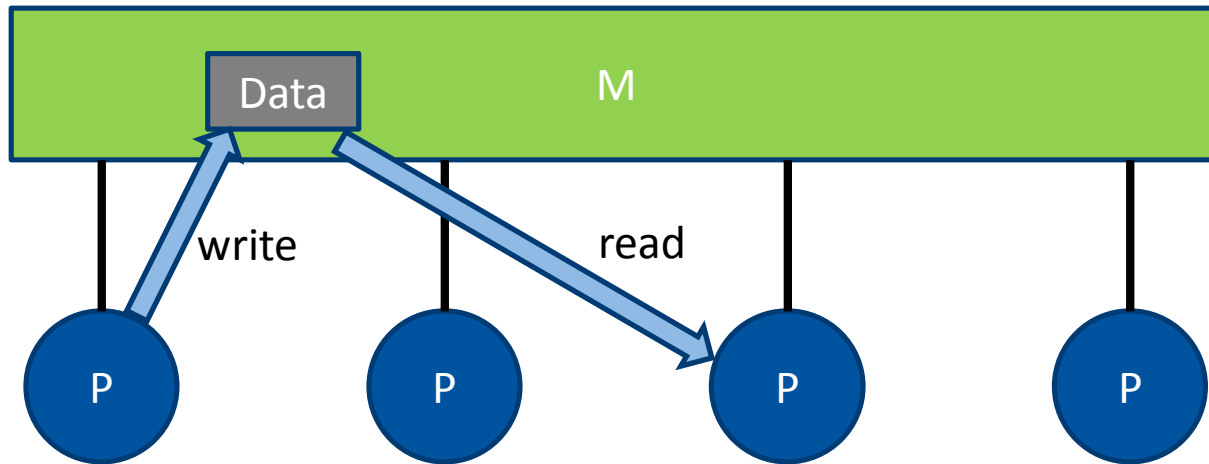
■ Shared Memory

→ All processing elements (P) have direct access to the main memory block (M)



■ Shared Memory

→ All processing elements (P) have direct access to the main memory block (M)



→ Data exchange is achieved through read/write operations on shared variables located in the global address space

■ Shared Memory – Pros

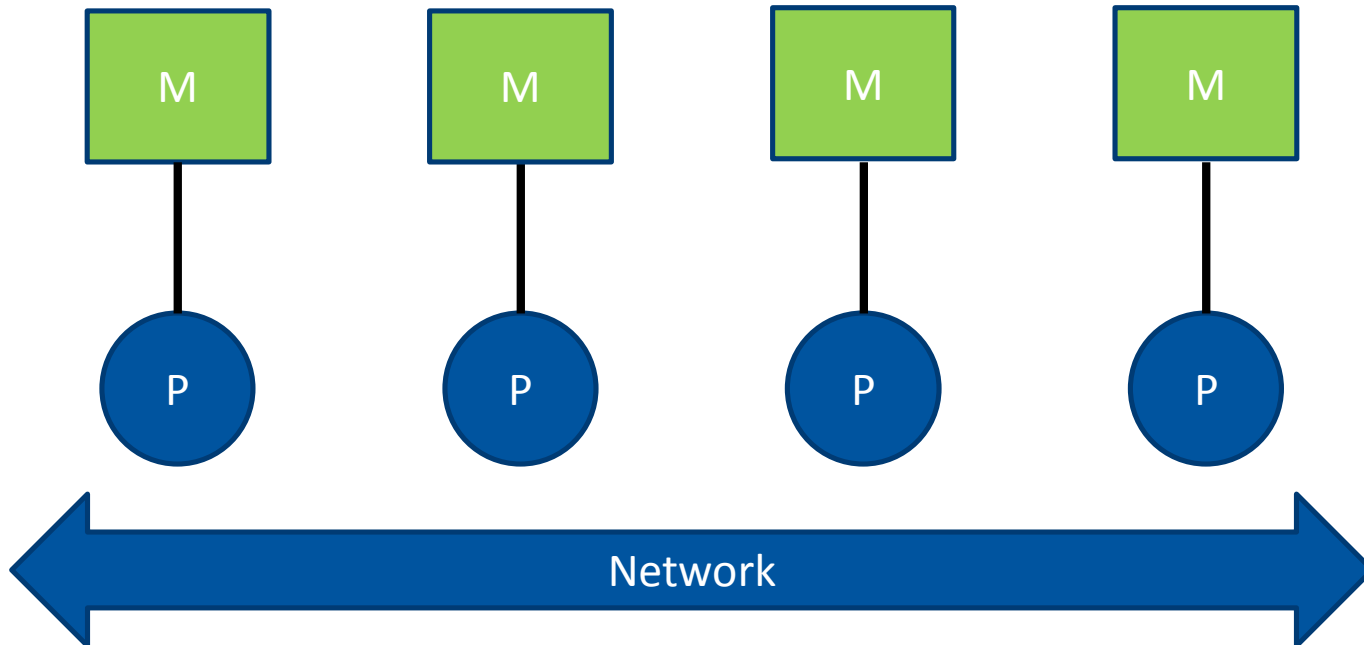
- All processing elements (P) have direct access to the main memory (M)
 - Single System Image
 - One single OS instance – easier to install and manage
 - Execution streams typically implemented as a set of OS entities that share a single (virtual) address space – *threads* within a single *process*
- Data exchange is achieved through the means of read/write operations in the global address space
 - Easy to conceptualise and program:
thread 1: **a = produce_value();** thread 2: **compute(a);**

■ Shared Memory – Cons

- Requires complex hardware
 - Memory usually divided into regions (NUMA) to reduce complexity
- Processing elements typically have caches
 - Maintaining cache coherence is very expensive
 - Non-cache-coherent systems are harder to program
- Data races
 - Synchronisation needed to enforce access order – barriers, locks, etc.

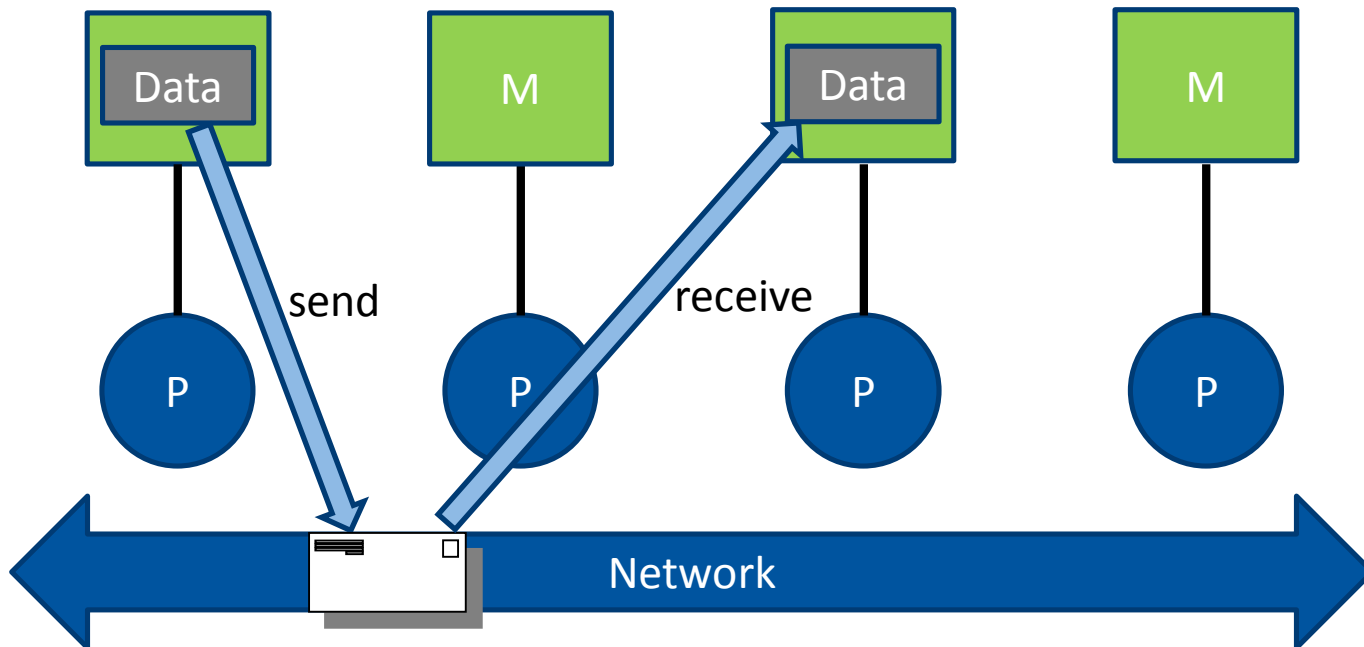
■ Distributed Memory

→ Each processing element (P) has its separate main memory block (M)



■ Distributed Memory

→ Each processing element (P) has its separate main memory block (M)



→ Data exchange is achieved through message passing over the network

■ Distributed Memory

- Each processing element (P) has its separate main memory block (M)
- Data exchange is achieved through message passing over the network
- Message passing could be either explicit (MPI) or implicit (PGAS)
- Programs typically implemented as a set of OS entities with own (virtual) address spaces – *processes*
- No shared variables
 - No data races
 - Explicit synchronisation mostly unneeded
 - Results as side effect of the send-receive semantics

- **A process is a running in-memory instance of an executable file**
 - Executable code, e.g., binary machine instructions
 - One or more threads of execution sharing memory address space
 - Memory: data, heap, stack, processor state (CPU registers and flags)
 - Operating system context (e.g. signals, I/O handles, etc.)
 - PID
- **Isolation and protection**
 - A process cannot interoperate with other processes or access their context (even on the same node) without the help of the operating system
 - No direct inter-process data exchange (isolated/virtual address spaces)
 - No direct inter-process synchronisation

■ Interaction with other processes

- Shared memory segments
 - Restricted to the same node
- File system
 - Slow; shared file system required for internode data access
- Networking (e.g. sockets, named pipes, etc.)
 - Coordination and addressing issues
- Special libraries (middleware) make IPC transparent and more portable
 - **MPI**, PVM – tightly coupled
 - Globus Toolkit (GRID infrastructure) – loosely coupled
 - BOINC (SETI@home, Einstein@home, *@home) – decoupled

- **Abstractions make programming and understanding easier**

- **Single Program Multiple Data**

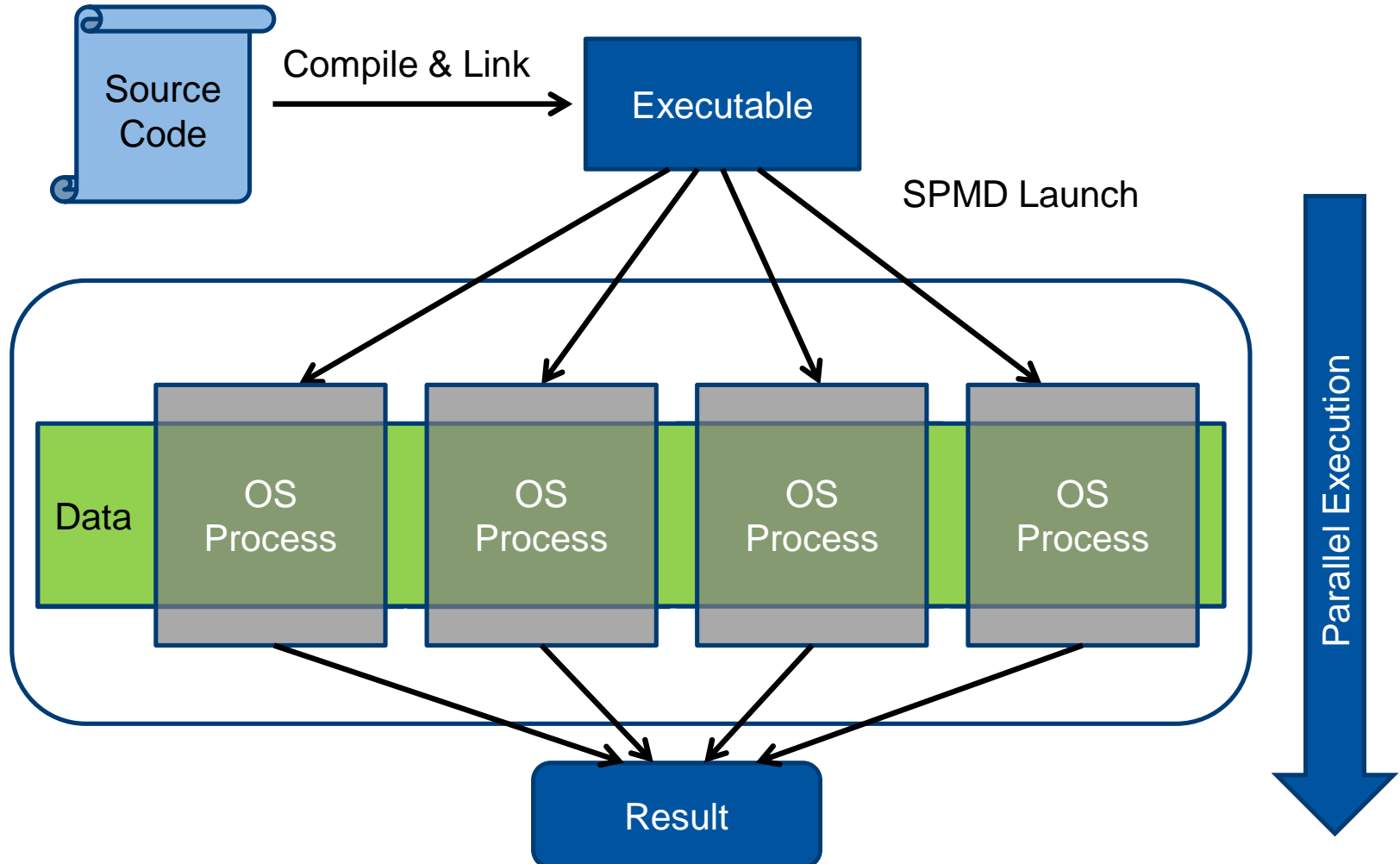
- Multiple instruction flows (instances) from a Single Program working on Multiple (different parts of) Data

- Instances could be threads (OpenMP) and/or processes (MPI)

- Each instance receives a unique ID – can be used for flow control

```
if (myID == specificID)
{
    do_something();
}
else
{
    do_something_different();
}
```

■ SPMD Program Lifecycle – multiple processes (e.g. MPI)



- **Provide dynamic identification of all peers**

- Who else is also working on this problem?

- **Provide robust mechanisms to exchange data**

- Whom to send data to / From whom to receive the data?

- How much data?

- What kind of data?

- Has the data arrived?

- **Provide synchronisation mechanisms**

- Have all processes reached same point in the program execution flow?

- **Provide methods to launch and control a set of processes**

- How do we start multiple processes and get them to work together?

- **Portability**

■ Sockets API is straightforward but there are some major issues:

- How to obtain the set of communicating partners?
- Where and how can these partners be reached?
 - Write your own registry server or use broadcast/multicast groups
 - **Worst case: AF_INET sockets with FQDN and TCP port number**
e.g. linuxbmc0064.rz.rwth-aachen.de:24892
- How to coordinate the processes in the parallel job?
 - Does the user have to start each process in his parallel job by hand?
 - Executable distribution and remote launch
 - Integration with DRMs (batch queuing systems)
- Redirection of standard I/O and handling of signals

■ Define the problem

→ problem space

→ methods

■ Solution:

→ input domain

→ algorithm

→ output domain

■ Decomposition

→ Split the input domain into (preferably non-interacting) regions

■ Motivation

■ Part 1

- Concepts
- Point-to-point communication
- Non-blocking operations

■ Part 2

- Collective operations
- Communicators
- User datatypes

■ Part 3

- Hybrid parallelisation
- Common parallel patterns

■ Message Passing Interface

- The de-facto standard API for explicit message passing nowadays
- A moderately large standard (v3.1 is a 868 pages long)
- Maintained by the non-profit Message Passing Interface Forum

<http://www.mpi-forum.org/>

■ Many concrete implementations of the MPI standard

- Open MPI, MPICH, Intel MPI, MVAPICH, MS-MPI, etc.

■ MPI is used to express the explicit interaction (communication) in programs for computers with distributed memory

■ MPI provides source level portability of parallel applications between different implementations and hardware platforms

- **A language-independent specification (LIS) of a set of communication and I/O operations**

- Standard bindings for C and Fortran

- Concrete function prototypes / interfaces

- Non-standard bindings for other languages exist:

- C++ [Boost.MPI](#)

- Java [Open MPI](#), [MPJ Express](#)

- Python [mpi4py](#)

- **Unlike OpenMP and PGAS languages, MPI does not extend the base language, but provides a set of library functions (+ specialised runtime) and makes use of existing compilers**

- **Version 1.0 (1994): FORTRAN 77 and C bindings**
- **Version 1.1 (1995): Minor corrections and clarifications**
- **Version 1.2 (1997): Further corrections and clarifications**
- **Version 2.0 (1997): MPI-2 – Major extensions**
 - One-sided communication
 - Parallel I/O
 - Dynamic process creation
 - Fortran 90 and C++ bindings
 - Language interoperability
- **Version 2.1 (2008): Merger of MPI-1 and MPI-2**
- **Version 2.2 (2009): Minor corrections and clarifications**
 - C++ bindings deprecated
- **Version 3.0 (2012): Major enhancements**
 - Non-blocking collective operations
 - Modern Fortran 2008 bindings
 - C++ deleted from the standard
- **Version 3.1 (2015): Corrections and clarifications**
 - Portable operation with address variables
 - Non-blocking collective I/O

- **The MPI Forum document archive (free standards for everyone!)**

- <http://www.mpi-forum.org/docs/>

- **The MPI home page at Argonne National Lab**

- <http://www-unix.mcs.anl.gov/mpi/>

- <http://www.mcs.anl.gov/research/projects/mpi/www/>

- **Open MPI (default MPI implementation on the RWTH cluster)**

- <http://www.open-mpi.org/>

- **Manual pages**

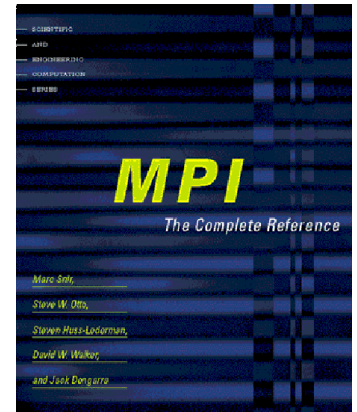
- man MPI

- man MPI_Xxx_yyy_zzz (for all MPI calls)

■ MPI: The Complete Reference Vol. 1 The MPI Core

by Marc Snir, Steve Otto, Steven Huss-Lederman,
David Walker, Jack Dongarra

2nd edition, The MIT Press, 1998



■ MPI: The Complete Reference Vol. 2 The MPI Extensions

by William Gropp, Steven Huss-Lederman,
Andrew Lumsdain, Ewing Lusk, Bill Nitzberg,
William Saphir, Marc Snir

2nd edition, The MIT Press, 1998



■ Using MPI

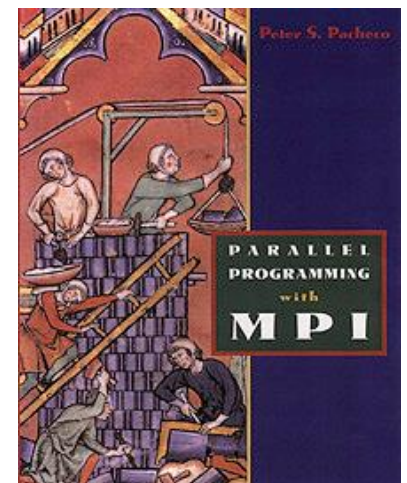
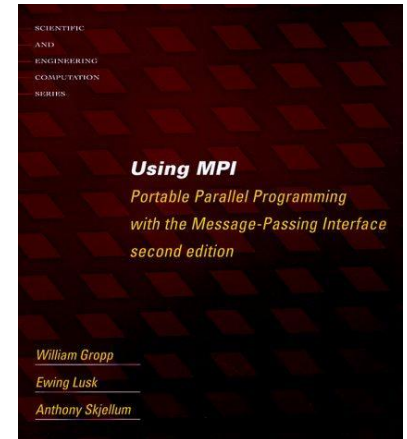
by William Gropp, Ewing Lusk, Anthony Skjellum
The MIT Press, Cambridge/London, 1999

■ Using MPI-2

by William Gropp, Ewing Lusk, Rajeev Thakur
The MIT Press, Cambridge/London, 2000

■ Parallel Programming with MPI

by Peter Pacheco
Morgan Kaufmann Publishers, 1996



■ MPI Basics

→ Start-up, initialisation, finalisation, and shutdown

■ Point-to-Point Communication

→ Send and receive

→ Basic MPI data types

→ Message envelope

→ Combined send and receive

→ Send modes

→ Non-blocking operations

→ Common pitfalls

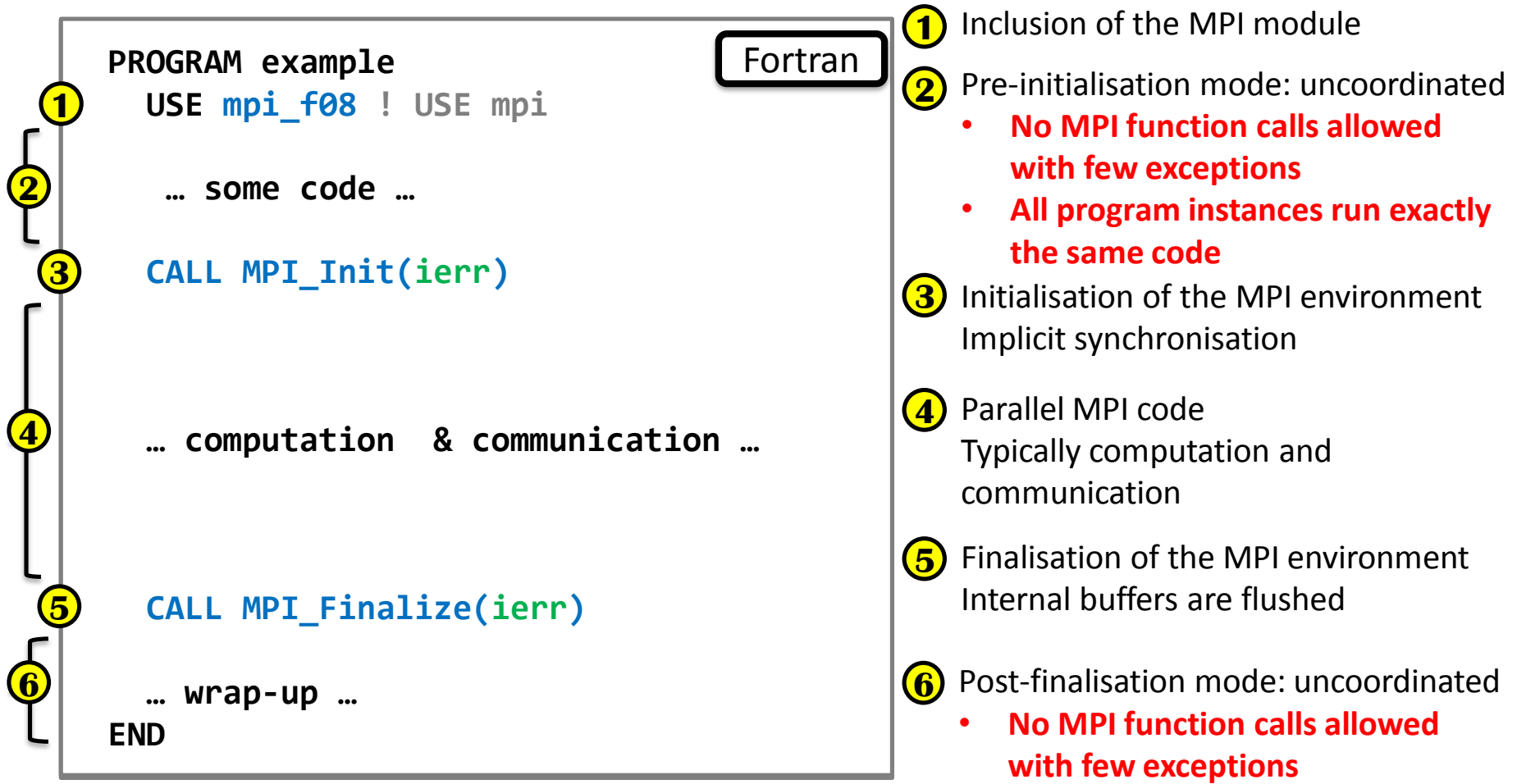
■ Start-up, initialisation, finalisation, and shutdown – C

```
1 #include <mpi.h>
2
3 int main(int argc, char **argv)
4 {
5     ... some code ...
6     MPI_Init(&argc, &argv);
7
8     ... computation & communication ...
9
10    MPI_Finalize();
11    ... wrap-up ...
12    return 0;
13 }
```

C

- 1 Inclusion of the MPI header file
- 2 Pre-initialisation mode: uncoordinated
 - **No MPI function calls allowed with few exceptions**
 - **All program instances run exactly the same code**
- 3 Initialisation of the MPI environment
Implicit synchronisation
- 4 Parallel MPI code
Typically computation and communication
- 5 Finalisation of the MPI environment
Internal buffers are flushed
- 6 Post-finalisation mode: uncoordinated
 - **No MPI function calls allowed with few exceptions**

■ Start-up, initialisation, finalisation, and shutdown – Fortran



- How many processes are there in total?
- Who am I?

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

C

```
int main(int argc, char **argv)  
{
```

```
    ... some code ...
```

```
    int ierr = MPI_Init(&argc, &argv),  
              numberOfProcs, rank;
```

```
    ... other code ...
```

```
    ierr = MPI_Comm_size(MPI_COMM_WORLD,  
                          &numberOfProcs);
```

```
    ierr = MPI_Comm_rank(MPI_COMM_WORLD,  
                          &rank);
```

```
    ... computation & communication ...
```

```
    ierr = MPI_Finalize();
```

```
    ... wrap-up ...
```

```
    return 0;
```

```
}
```

①

②

① Obtains the number of processes (ranks) in the MPI program

Example: if the job was started with 4 processes, then **numberOfProcs** will be set to 4 by the call

② Obtains the identity of the calling process within the MPI program

NB: MPI processes are numbered starting from 0

Example: if there are 4 processes in the job, then **rank** receive value of 0 in the first process, 1 in the second process, and so on

- How many processes are there in total?
- Who am I?

Fortran

PROGRAM example

```
USE mpi_f08 ! USE mpi
```

```
INTEGER :: rank, numberOfProcs, ierr
```

```
... some code ...
```

```
CALL MPI_Init(ierr)
```

```
... other code ...
```

```
CALL MPI_Comm_size(MPI_COMM_WORLD,&  
    numberOfProcs, ierr)
```

```
CALL MPI_Comm_rank(MPI_COMM_WORLD,&  
    rank, ierr)
```

```
... computation & communication ...
```

```
CALL MPI_Finalize(ierr)
```

```
... wrap-up ...
```

```
END PROGRAM example
```

1

2

1

Obtains the number of processes (ranks) in the MPI program

Example: if the job was started with 4 processes, then **numberOfProcs** will be set to 4 by the call

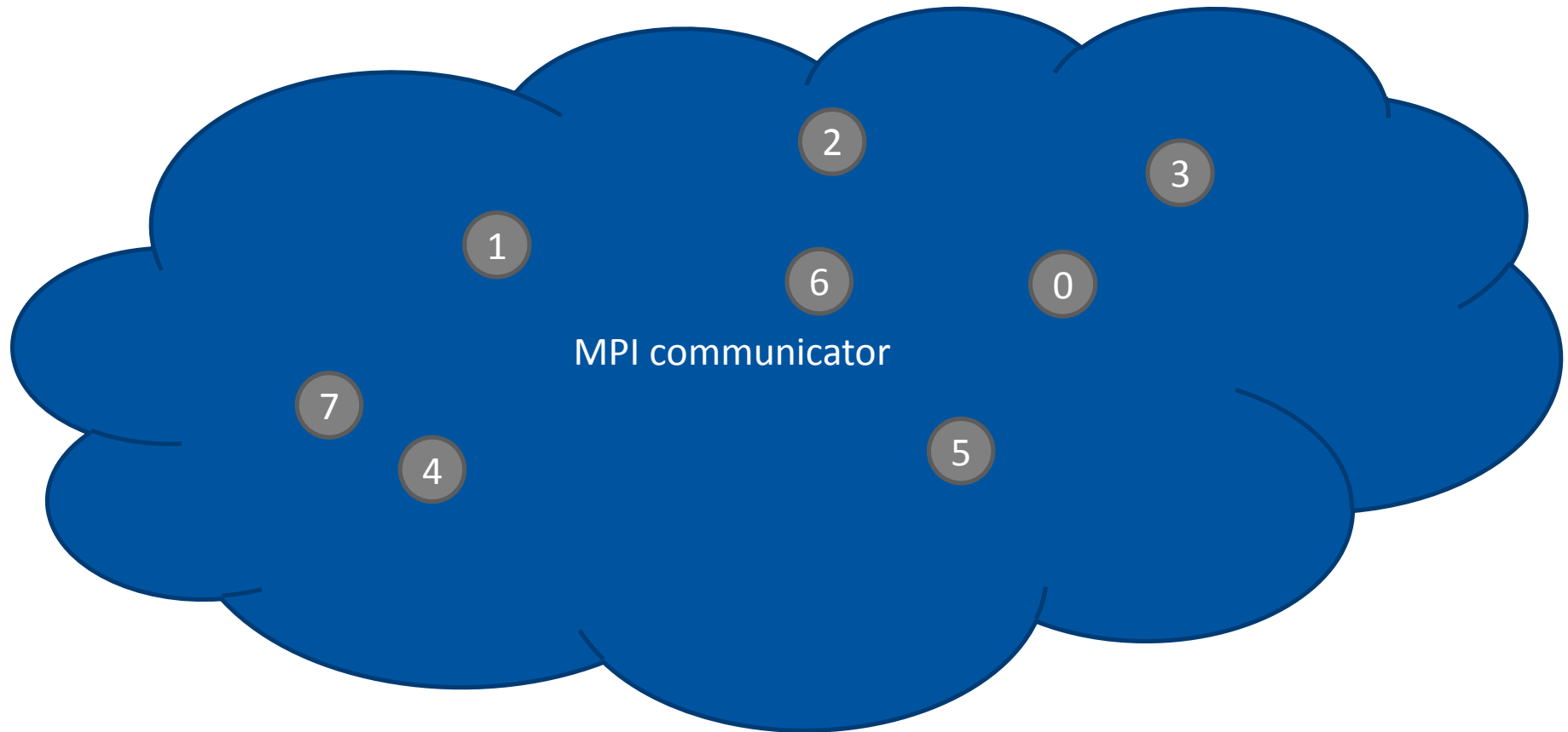
2

Obtains the identity of the calling process within the MPI program

NB: MPI processes are numbered starting from 0

Example: if there are 4 processes in the job, then **rank** receive value of 0 in the first process, 1 in the second process, and so on

- The processes in any MPI program are initially indistinguishable
- MPI_Init assigns each process a unique identity – rank



- **The processes in any MPI program are initially indistinguishable**
- **MPI_Init assigns each process a unique identity – rank**
 - Without personality, the started MPI processes cannot do coordinated parallel work in the pre-initialisation mode
 - Ranks range from 0 up to the total number of processes minus 1
- **Ranks are associated with the so-called communicators**
 - Logical contexts where communication takes place
 - Represent groups of MPI processes with some additional information
 - The most important one is the world communicator **MPI_COMM_WORLD**
 - Contains all processes launched *initially* as part of the MPI program
 - Ranks are always provided in MPI calls in combination with the corresponding communicator

■ Initialisation:

```
C:      ierr = MPI_Init(&argc, &argv);  
Fortran: CALL MPI_Init(ierr)
```

- Initialises the MPI library and makes the process member of the world communicator
- [C] Modern MPI implementations allow both arguments to be NULL, otherwise they *must* point to the arguments of **main()**
- **May not be called more than once for the duration of the program execution**
- Error code as return value in [C] and additional parameter in [F]

■ Finalisation:

```
C:      ierr = MPI_Finalize();  
Fortran: CALL MPI_Finalize(ierr)
```

- Cleans up the MPI library and prepares the process for termination
- **Must be called once before the process terminates**
- Having other code after the finalisation call is not recommended

■ Number of processes in the MPI program:

```
C:      ierr = MPI_Comm_size(MPI_COMM_WORLD, &size);  
Fortran: CALL MPI_Comm_size(MPI_COMM_WORLD, size, ierr)
```

- Obtains the number of processes initially started in the MPI program (the size of the world communicator)
- **size** is an integer variable
- **MPI_COMM_WORLD** is a predefined constant *MPI handle* that represents the world communicator

■ Process identification:

```
C:      ierr = MPI_Comm_rank(MPI_COMM_WORLD, &rank);  
Fortran: CALL MPI_Comm_rank(MPI_COMM_WORLD, rank, ierr)
```

- Determines the rank (unique ID) of the process within the world communicator
- **rank** is an integer variable; receives value between 0 and #processes - 1

- **Most MPI calls in C return an integer error code:**
 - `int MPI_Comm_size(...)`

- **Most MPI calls are Fortran subroutines with an extra **INTEGER** output argument (always last one in the list) for the error code:**
 - `SUBROUTINE MPI_Comm_size (... , ierr)`
 - The Fortran 2008 interface makes the error argument optional

- **Fortran note**
 - The **USE mpi** interface does not provide interfaces for all MPI routines
 - **USE mpi_f08** when possible

- **Error codes indicate the success of the operation:**
 - Failure is indicated by error codes other than **MPI_SUCCESS**
 - C: `if (MPI_SUCCESS != MPI_Init(NULL, NULL)) ...`
 - Fortran: `CALL MPI_Init(ierr)`
`IF (ierr /= MPI_SUCCESS) ...`
- **An MPI error handler is called first before the call returns**
 - **The default error handler for non-I/O calls aborts the entire MPI program!**
 - Error checking in simple programs is redundant
- **NB: Actual MPI error code values are implementation specific**

- **MPI objects (e.g. communicators) are referenced via handles**
 - Opaque process-local values
 - Cannot be passed from one process to another
- **C (mpi.h)**
 - typedef'd handle types: MPI_Comm, MPI_Datatype, MPI_File, etc.
- **Fortran (USE mpi)**
 - All handles are INTEGER values
 - Easy to pass the wrong handle type
- **Fortran 2008 (USE mpi_f08)**
 - Wrapped INTEGER values: TYPE(MPI_Comm), TYPE(MPI_File), etc.
 - The INTEGER handle is still available: comm%MPI_VAL

Provide dynamic identification of all peers

→ Who am I and who else is also working on this problem?

■ Provide robust mechanisms to exchange data

→ Whom to send data to / From whom to receive the data?

→ How much data?

→ What kind of data?

→ Has the data arrived?

■ Provide synchronisation mechanisms

→ Have all processes reached same point in the program execution flow?

■ Provide methods to launch and control a set of processes

→ How do we start multiple processes and get them to work together?

■ Portability

■ Motivation

■ Part 1

- Concepts
- Point-to-point communication
- Non-blocking operations

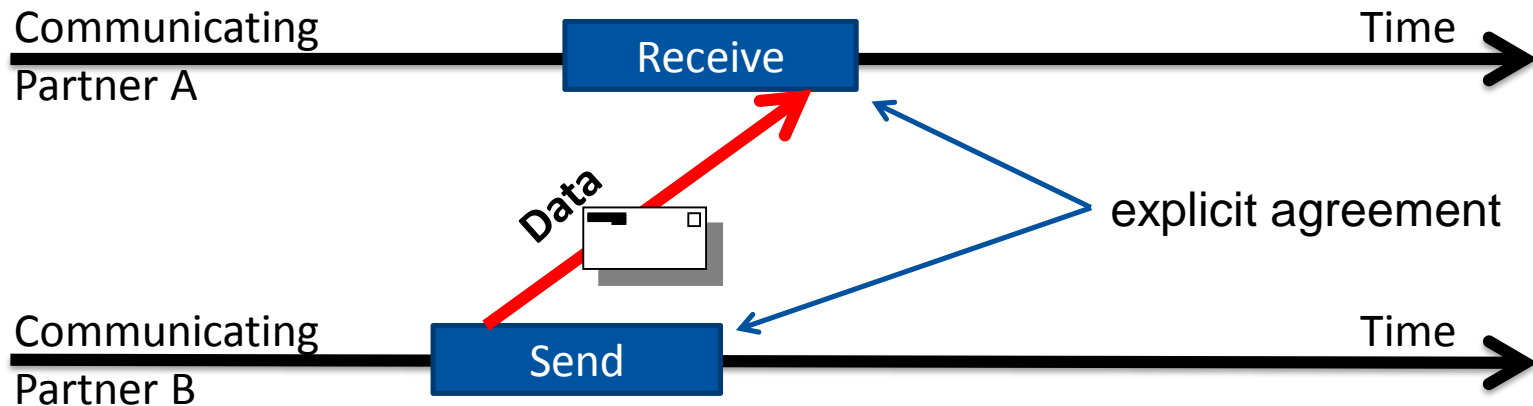
■ Part 2

- Collective operations
- Communicators
- User datatypes

■ Part 3

- Hybrid parallelisation
- Common parallel patterns

- The goal is to enable communication between processes that share no memory space



- **Explicit message passing requires:**
 - Send and receive primitives (operations)
 - Known addresses of both the sender and the receiver
 - Specification of what has to be sent/received

■ Sending a message:

What?

```
MPI_Send (void *data, int count, MPI_Datatype type,  
int dest, int tag, MPI_Comm comm)
```

To whom?

C

- **data:** location in memory of the data to be sent
- **count:** number of data elements to be sent (MPI is array-oriented)
- **type:** Handle of the *MPI datatype* of the buffer content
- **dest:** rank of the receiver
- **tag:** additional identification of the message
ranges from 0 to UB (impl. dependant but not less than 32767)
- **comm:** communication context (communicator handle)

```
MPI_Send (data, count, type, dest, tag, comm, ierr)
```

Fortran

■ Receiving a message:

What?

```
MPI_Recv (void *data, int count, MPI_Datatype type,  
int source, int tag, MPI_Comm comm, MPI_Status *status)
```

C

- **data:** location of the receive buffer
- **count:** size of the receive buffer in data elements
- **type:** Handle of the MPI datatype of the data elements
- **source:** rank of the sender or **MPI_ANY_SOURCE** (wildcard)
- **tag:** message tag or **MPI_ANY_TAG** (wildcard)
- **comm:** communication context
- **status:** status of the receive operation or **MPI_STATUS_IGNORE**

From whom?

```
MPI_Recv (data, count, type, src, tag, comm, status, ierr)
```

Fortran

- **MPI is a library – it cannot infer the type of elements in the supplied buffer at run time and that’s why it has to be told what it is**
- **MPI datatypes tell MPI how to:**
 - read binary values from the send buffer
 - write binary values into the receive buffer
 - correctly apply value alignments
 - convert between machine representations in heterogeneous environments
- **MPI datatype **must** match the language type(s) in the data buffer**
- **MPI datatype **must** match between sender and receiver**
- **MPI datatypes are handles and cannot be used to declare variables**

- MPI provides many predefined datatypes for each language binding:

→ C

MPI data type	C data type
MPI_CHAR	char
MPI_SHORT	short
MPI_INT	int
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_UNSIGNED_INT	unsigned int
...	...
MPI_BYTE	-

8 binary digits
no conversion
used for binary data

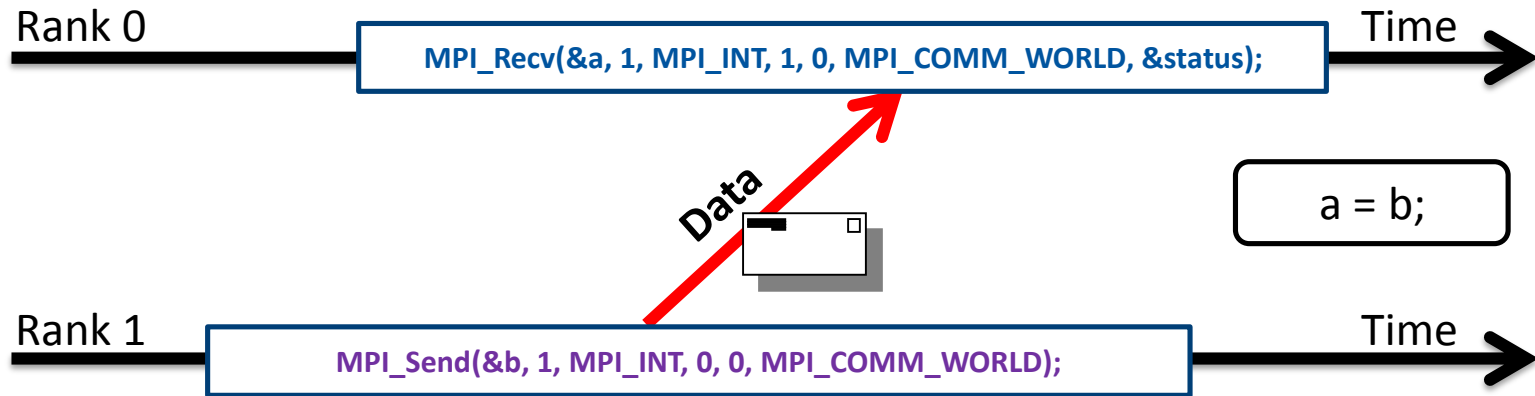
- MPI provides many predefined datatypes for each language binding:

- C

- Fortran

MPI data type	Fortran data type
MPI_INTEGER	INTEGER
MPI_REAL	REAL
MPI_REAL8	REAL(KIND=8)
MPI_DOUBLE_PRECISION	DOUBLE PRECISION
MPI_COMPLEX	COMPLEX
MPI_LOGICAL	LOGICAL
MPI_CHARACTER	CHARACTER(1)
...	...
MPI_BYTE	-

- Message passing in MPI is explicit:

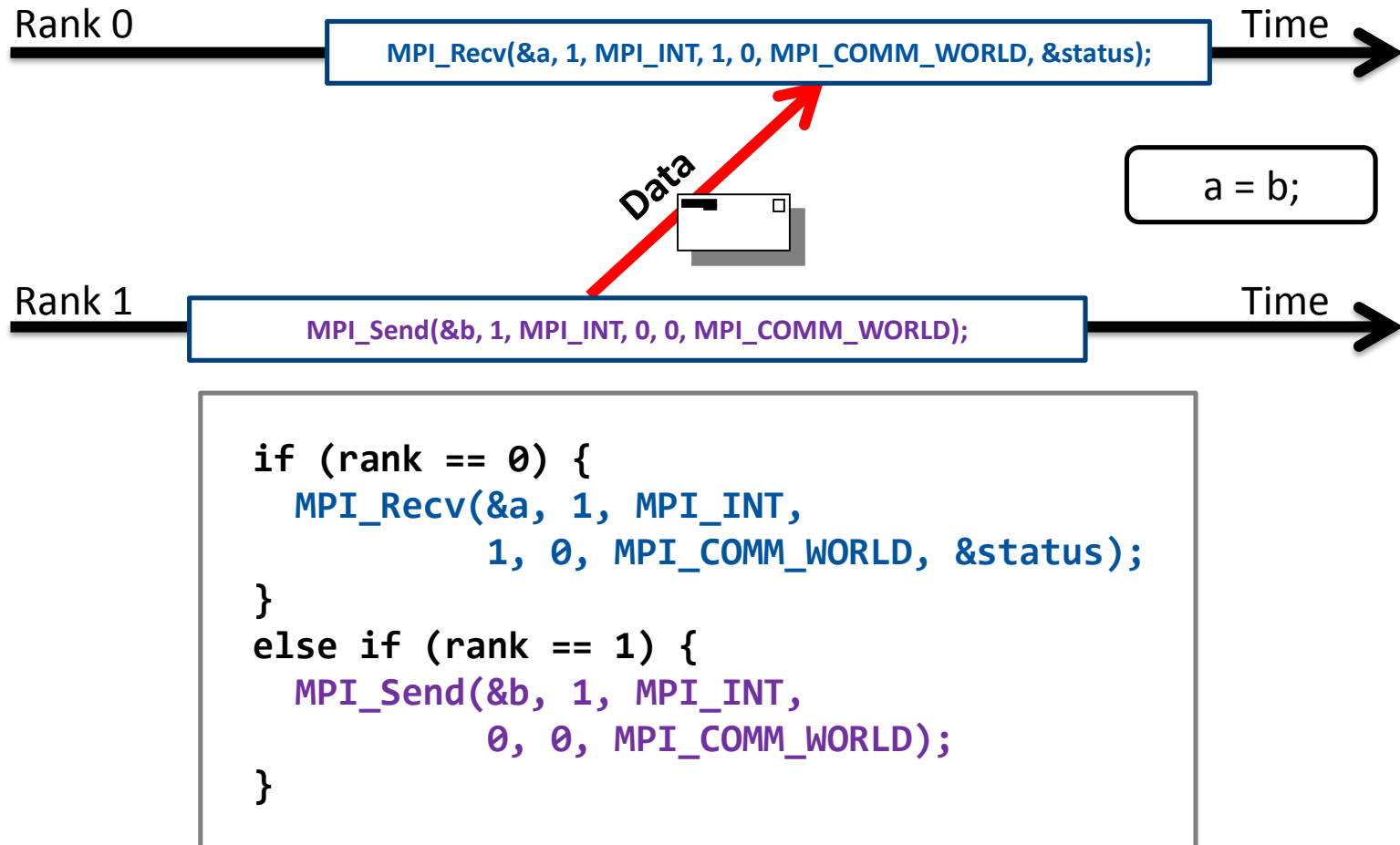


- The value of variable *b* in rank 1 is copied into variable *a* in rank 0

- For now, assume that *comm* is always `MPI_COMM_WORLD`

→ We will talk about other communicators later on

■ Message passing in MPI is explicit:



Provide dynamic identification of all peers

→ Who am I and who else is also working on this problem?

Provide robust mechanisms to exchange data

→ Whom to send data to / From whom to receive the data?

→ How much data?

→ What kind of data?

→ ~~Has the data arrived?~~ (only the receiver knows)

■ Provide synchronisation mechanisms

→ Have all processes reached same point in the program execution flow?

■ Provide methods to launch and control a set of processes

→ How do we start multiple processes and get them to work together?

Portability

Complete MPI Example



C

```
#include <mpi.h>

int main(int argc, char **argv)
{
    int nprocs, rank, data;
    MPI_Status status;
    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD,
                  &nprocs);
    MPI_Comm_rank(MPI_COMM_WORLD,
                  &rank);
    if (rank == 0)
        MPI_Recv(&data, 1, MPI_INT, 1, 0,
                 MPI_COMM_WORLD, &status);
    else if (rank == 1)
        MPI_Send(&data, 1, MPI_INT, 0, 0,
                 MPI_COMM_WORLD);
    MPI_Finalize();
    return 0;
}
```

1

2

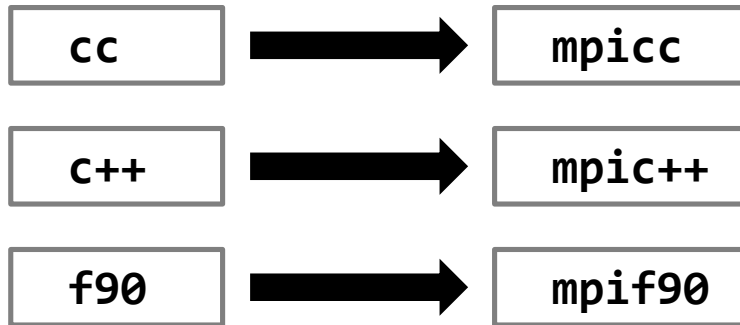
3

4

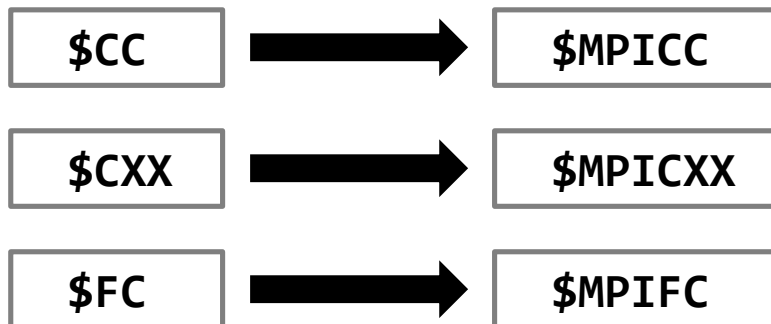
5

- 1 Initialise the MPI library
- 2 Identify current process
- 3 Behave differently based on the rank
- 4 Communicate
- 5 Clean up the MPI library

- MPI is a typical library with C header files, Fortran modules, etc.
- Some MPI vendors provide convenience compiler wrappers:



- On RWTH Compute Cluster (depending on the loaded modules):



- MPI is a typical library with C header files, Fortran modules, etc.
- Some MPI vendors provide convenience compiler wrappers:

```
cluster:~[1]$ $MPICC --show
icc
-I/opt/MPI/openmpi-1.6.5/linux/intel/include
-I/opt/MPI/openmpi-1.6.5/linux/intel/include/openmpi
-fexceptions
-pthread
-I/opt/MPI/openmpi-1.6.5/linux/intel/lib
-Wl,-rpath,/opt/MPI/openmpi-1.6.5/linux/intel/lib
-I/opt/MPI/openmpi-1.6.5/linux/intel/lib
-L/opt/MPI/openmpi-1.6.5/linux/intel/lib
-lmpi
-ldl
-Wl,--export-dynamic
-lns1
-lutil
```

- **Most MPI implementations provide a special launcher program:**

```
mpiexec -n nprocs ... program <arg1> <arg2> <arg3> ...
```

→ launches **nprocs** instances of **program** with command-line arguments **arg1**, **arg2**, ... and provides the MPI library with enough information in order to establish network connections between the processes

- **The standard specifies the mpiexec program but does not require it:**

→ IBM BG/Q: **runjob --np 1024 ...**

→ SLURM resource manager: **srun ...**

- **On RWTH Compute Cluster:**

→ interactive jobs

```
$MPIEXEC -n nprocs ... program <arg1> <arg2> <arg3> ...
```

→ batch jobs

```
$MPIEXEC $FLAGS_MPI_BATCH ... program <arg1> <arg2> <arg3> ...
```

■ Most MPI implementations provide a special launcher program:

```
mpiexec -n nprocs ... program <arg1> <arg2> <arg3> ...
```

- launches **nprocs** instances of **program** with command-line arguments **arg1**, **arg2**, ... and provides the MPI library with enough information in order to establish network connections between the processes
- Sometimes called **mpirun**

■ The launcher often performs more than simply launching processes:

- Helps MPI processes find each other and establish the world communicator
- Redirects the standard output of all ranks to the terminal
- Redirects the terminal input to the standard input of rank 0
- Forwards received signals (Unix-specific)

Provide dynamic identification of all peers

→ Who am I and who else is also working on this problem?

Provide robust mechanisms to exchange data

→ Whom to send data to / From whom to receive the data?

→ How much data?

→ What kind of data?

→ ~~Has the data arrived?~~ (only the receiver knows)

■ Provide synchronisation mechanisms

→ Have all processes reached same point in the program execution flow?

Provide methods to launch and control a set of processes

→ How do we start multiple processes and get them to work together?

Portability

■ Compile and Run a Simple MPI Program

```
cluster:~[1]$ vim hello.c  
  
cluster:~[2]$ $MPICC -o hello.exe hello.c  
  
cluster:~[3]$ $MPIEXEC -n 4 hello.exe  
Hello world from rank 2 of 4  
Hello world from rank 0 of 4  
Hello world from rank 1 of 4  
Hello world from rank 3 of 4
```

- Reception of MPI messages is done by matching their envelope
- Send operation

```
MPI_Send (void *data, int count, MPI_Datatype type,  
int dest, int tag, MPI_Comm comm)
```

- Message Envelope:

	Sender	Receiver
Source	Implicit	Explicit, wildcard possible (MPI_ANY_SOURCE)
Destination	Explicit	Implicit
Tag	Explicit	Explicit, wildcard possible (MPI_ANY_TAG)
Communicator	Explicit	Explicit

Message Envelope

- Receive operation

```
MPI_Recv (void *data, int count, MPI_Datatype type,  
int source, int tag, MPI_Comm comm, MPI_Status *status)
```


- Reception of MPI messages is also dependent on the data.
- Recall:

```
MPI_Send (void *data, int count, MPI_Datatype type,  
          int dest, int tag, MPI_Comm comm)
```

```
MPI_Recv (void *data, int count, MPI_Datatype type,  
          int source, int tag, MPI_Comm comm, MPI_Status *status)
```

- The standard expects datatypes at both ends to match
 - Not enforced by most implementations
- Matching sends and receives must always come in pairs
- **NB: messages do not aggregate**

Rank 0:

```
MPI_Send(myArr,1,MPI_INT,1,0,MPI_COMM_WORLD)  
... some code ...  
MPI_Send(myArr,1,MPI_INT,1,0,MPI_COMM_WORLD)
```

Rank 1:

```
MPI_Recv(myArr,2,MPI_INT,0,0,MPI_COMM_WORLD,&stat)  
... some code ...
```



Unmatched

- **The receive buffer must be able to fit the entire message**
 - send count \leq receive count **OK** (but check status)
 - send count $>$ receive count **ERROR** (message truncation)
- **The MPI status object holds information about the received message**
- **C: `MPI_Status` status;**
 - `status.MPI_SOURCE` message source rank
 - `status.MPI_TAG` message tag
 - `status.MPI_ERROR` receive status code

- **The receive buffer must be able to fit the entire message**
 - send count \leq receive count **OK** (but check status)
 - send count $>$ receive count **ERROR** (message truncation)
- **The MPI status object holds information about the received message**
- **Fortran: `INTEGER, DIMENSION(MPI_STATUS_SIZE) :: status`**
 - `status(MPI_SOURCE)` message source rank
 - `status(MPI_TAG)` message tag
 - `status(MPI_ERROR)` receive status code

- **The receive buffer must be able to fit the entire message**
 - send count \leq receive count **OK** (but check status)
 - send count $>$ receive count **ERROR** (message truncation)
- **The MPI status object holds information about the received message**
- **Fortran 2008:**
 - **TYPE(MPI_Status) :: status**
 - **status%MPI_SOURCE** message source rank
 - **status%MPI_TAG** message tag
 - **status%MPI_ERROR** receive status code

■ Blocks until a matching message appears:

```
MPI_Probe (int source, int tag, MPI_Comm comm, MPI_Status *status)
```

- Message is not received, one must call **MPI_Recv** to receive it
- Information about the message is stored in the status field

```
MPI_Probe(MPI_ANY_SOURCE, MPI_ANY_TAG, comm, &status);
```

- Checks for any message in the given communicator

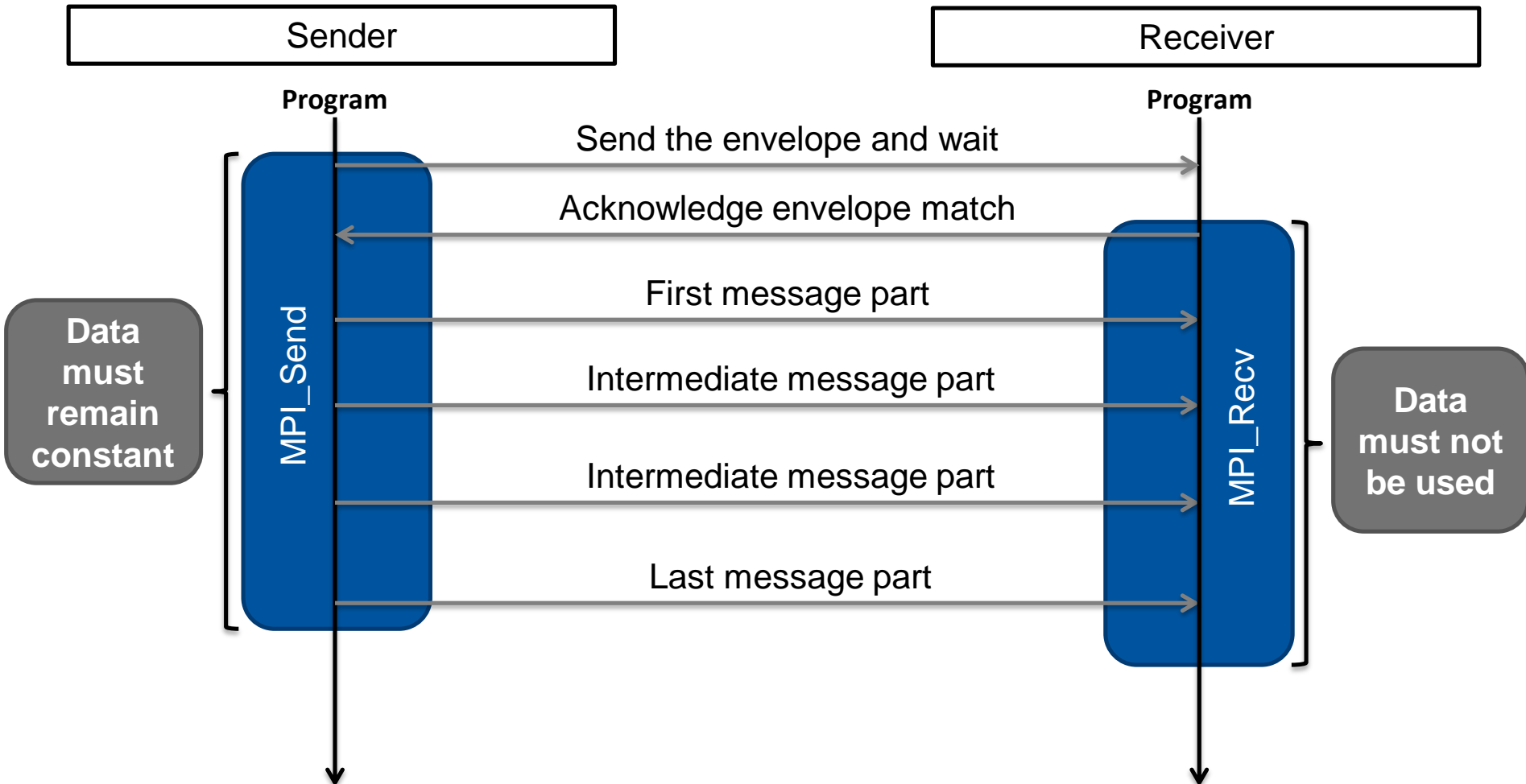
■ Message size inquiry:

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

- Calculates how many integral **datatype** elements can be formed from the data in the message referenced by **status**
- If the number is not integral, **count** is set to **MPI_UNDEFINED**
- Can be used with the status from **MPI_Recv** too

- **MPI operations complete once the message buffer is no longer in use by the MPI library and is thus free for reuse**
- **Send operations complete:**
 - once the message is constructed *and*
 - placed completely onto the network *or*
 - buffered completely (by MPI, the OS, the network, ...)
- **Receive operations complete:**
 - once the entire message has arrived and has been placed into the buffer
- **Blocking MPI calls only return once the operation has completed**
 - **MPI_Send** and **MPI_Recv** are blocking

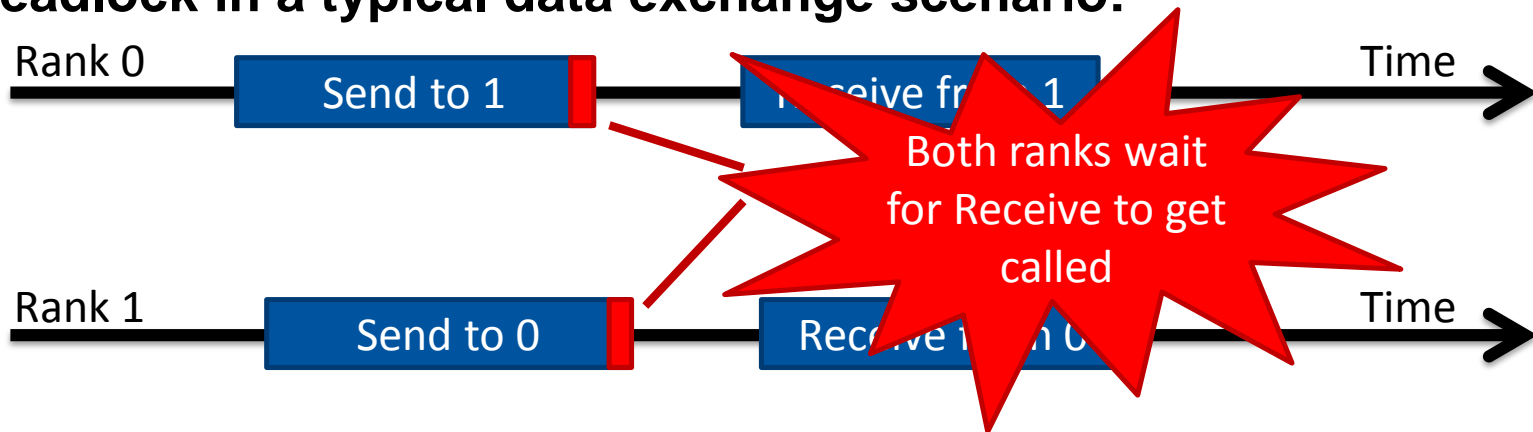
■ Blocking send (w/o buffering) and receive calls:



■ Both MPI_Send and MPI_Recv calls are blocking:

- The receive operation only returns after a matching message has arrived
- The send operation ***might*** be buffered (*implementation-specific!!!*) and therefore return before the message is actually placed onto the network
- Larger messages are usually sent only when both the send and the receive operations are active (synchronously)
- **Never rely on any implementation-specific behaviour!!!**

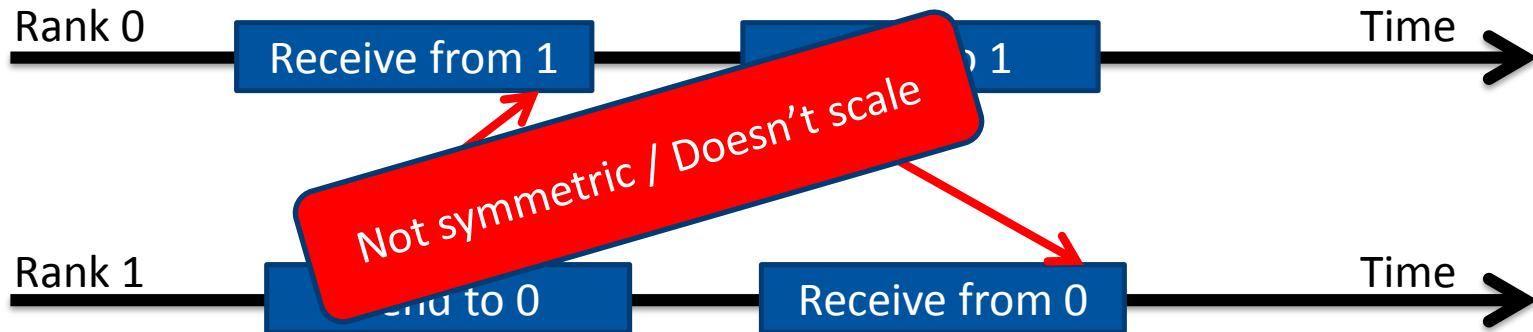
■ Deadlock in a typical data exchange scenario:



■ Both MPI_Send and MPI_Recv calls are blocking:

- The receive operation only returns after a matching message has arrived
- The send operation ***might*** be buffered (*implementation-specific!!!*) and therefore return before the message is actually placed onto the network
- Larger messages are usually sent only when both the send and the receive operations are active (synchronously)
- **Never rely on any implementation-specific behaviour!!!**

■ Deadlock prevention in a typical data exchange scenario:



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

- Combines message send and receive into a single call

	Send	Receive
Data	senddata	recvdata
Count	sendcount	recvcount
Type	sendtype	recvtype
Destination	dest	-
Source	-	source
Tag	sendtag	recvtag
Communicator	comm	comm
Receive status	-	status

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


- Sends one message and receives one message (in any order) without deadlocking (unless unmatched)
- **Send and receive buffers must not overlap!**

```
MPI_Sendrecv_replace (void *data, int count, MPI_Datatype datatype,  
                      int dest, int sendtag, int source, int recvtag,  
                      MPI_Comm comm, MPI_Status *status)
```

- First sends a message to *dest*, then receives a message from *source*, using the same memory location, elements count and datatype for both operations
- Usually slower than `MPI_Sendrecv` and might use more memory

- **Order is preserved in a given communicator for point-to-point operations between any pair of processes**
 - Messages within some communicator to the same rank are non-overtaking
 - Probe/receive returns the earliest matching message
- **Order is not guaranteed for**
 - messages sent within different communicators
 - messages arriving from different senders

```
MPI_Status status;  
  
MPI_Probe(MPI_ANY_SOURCE, MPI_COMM_WORLD, &status);  
... allocate ... message size ...  
MPI_Recv(buf, ..., MPI_INT, MPI_ANY_SOURCE, 0,  
         MPI_COMM_WORLD, &status);
```



- **Order is preserved in a given communicator for point-to-point operations between any pair of processes**
 - Messages within some communicator to the same rank are non-overtaking
 - Probe/receive returns the earliest matching message
- **Order is not guaranteed for**
 - messages sent within different communicators
 - messages arriving from different senders

```
MPI_Status status;  
  
MPI_Probe(MPI_ANY_SOURCE, 0, MPI_COMM_WORLD, &status);  
... allocate buffer based on message size ...  
MPI_Recv(buffer, size, MPI_INT, status.MPI_SOURCE, 0,  
         MPI_COMM_WORLD, &status);
```

Also applies to sequences of wildcard receives

■ Motivation

■ Part 1

- Concepts
- Point-to-point communication
- Non-blocking operations

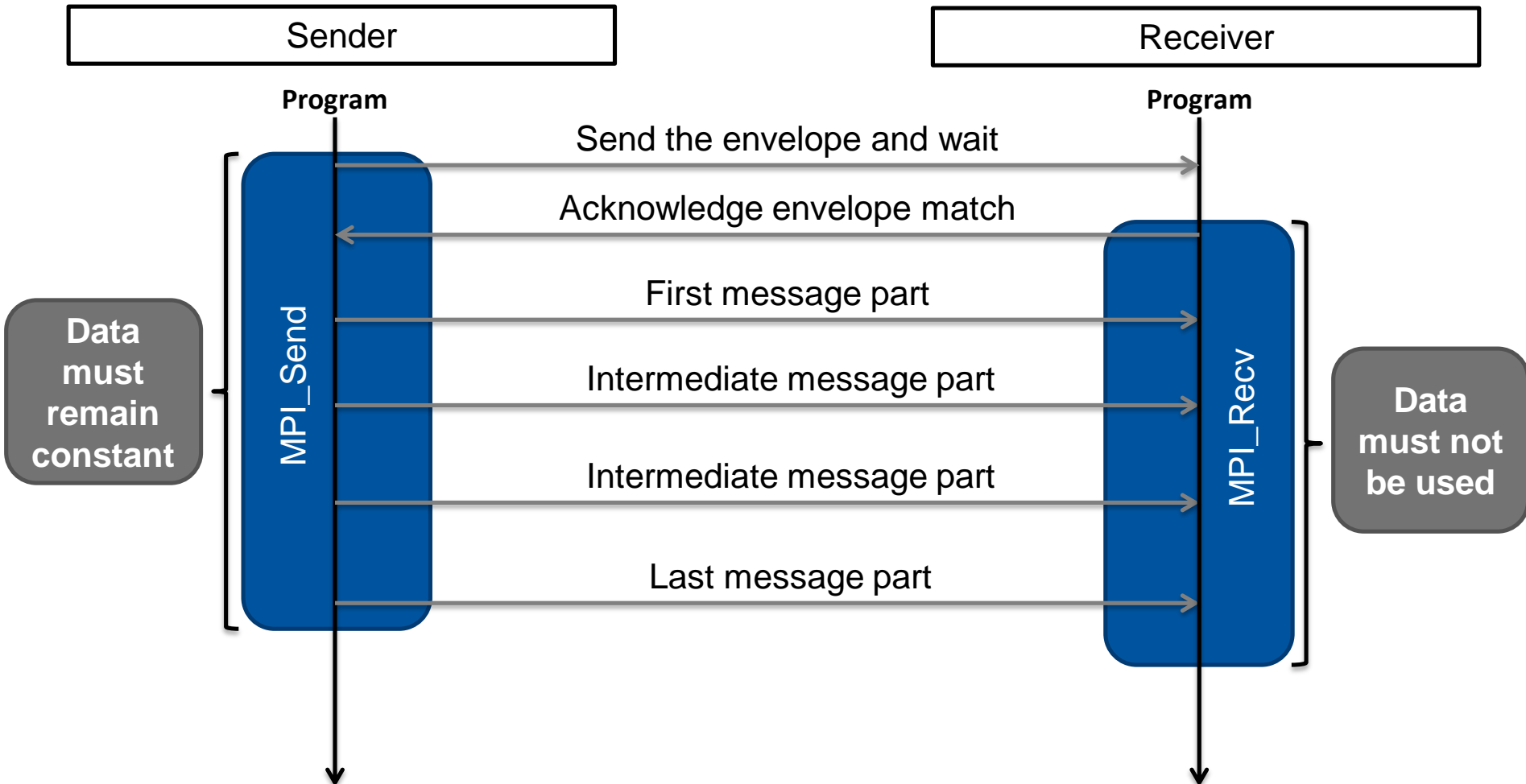
■ Part 2

- Collective operations
- Communicators
- User datatypes

■ Part 3

- Hybrid parallelisation
- Common parallel patterns

■ Blocking send (w/o buffering) and receive calls:



- **Non-blocking MPI calls return immediately while the communication operation continues asynchronously in the background**
- **Each non-blocking operation is represented by a request handle:**
 - C: `MPI_Request`
 - Fortran: `INTEGER`
 - Fortran 2008: `TYPE(MPI_Request)`
- **Non-blocking operations are progressed by certain MPI calls but most notably by the *test* and *wait* MPI calls**
- **Blocking MPI calls are equivalent to making a non-blocking call and waiting immediately afterwards for the operation to complete**
- **Used to overlay communication and computation and to prevent possible deadlocks**

■ Initiation of non-blocking send and receive operations:

```
MPI_Isend (void *data, int count, MPI_Datatype dataType,  
          int dest, int tag, MPI_Comm comm, MPI_Request *request)
```

```
MPI_Irecv (void *data, int count, MPI_Datatype dataType,  
          int source, int tag, MPI_Comm comm, MPI_Request *request)
```

→ **request:** on success set to the handle of the non-blocking operation

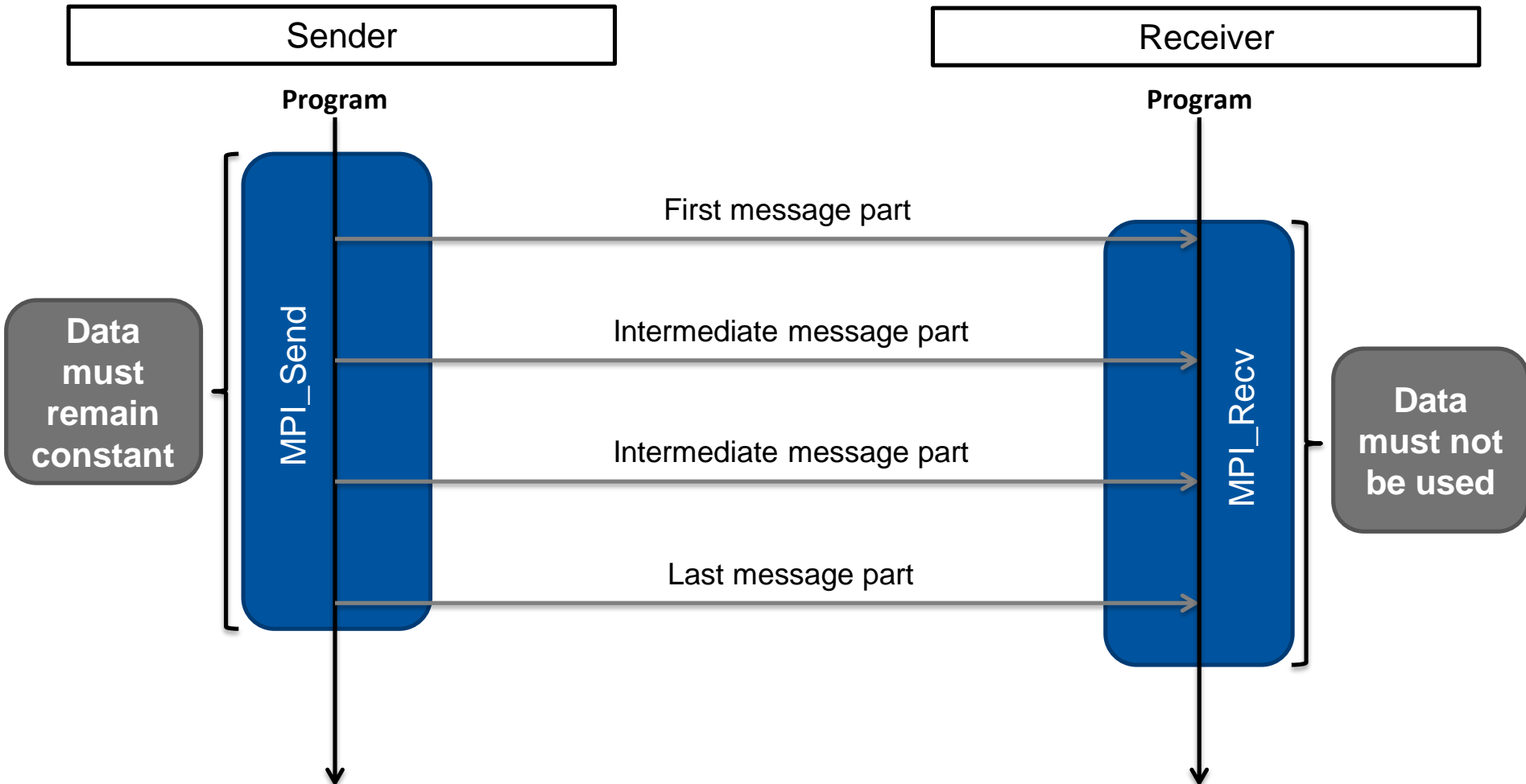
■ Blocking wait for completion:

```
MPI_Wait (MPI_Request *request, MPI_Status *status)
```

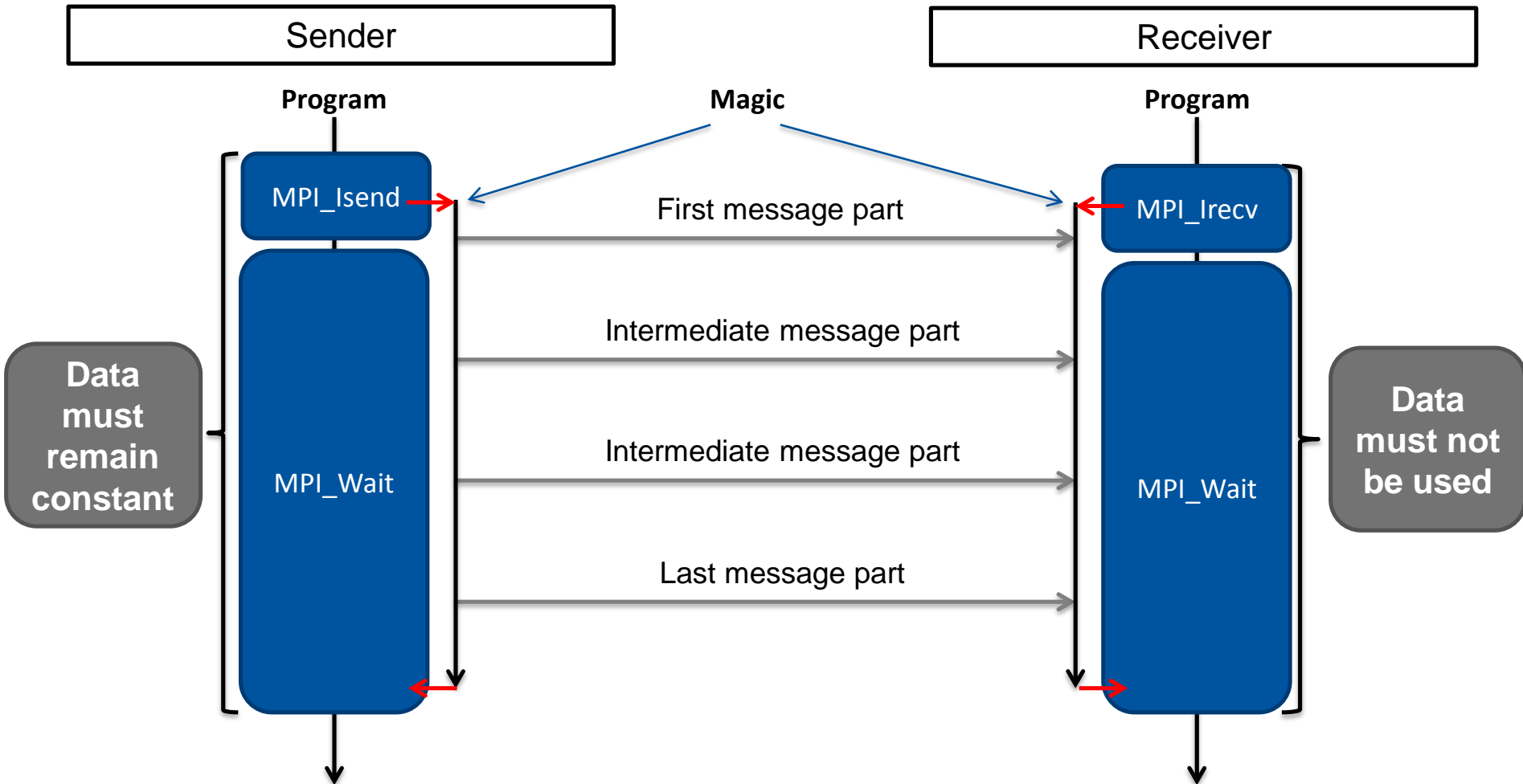
→ **request:** handle for an active non-blocking operation
freed and set to **MPI_REQUEST_NULL** upon successful return

→ **status:** status of the completed operation

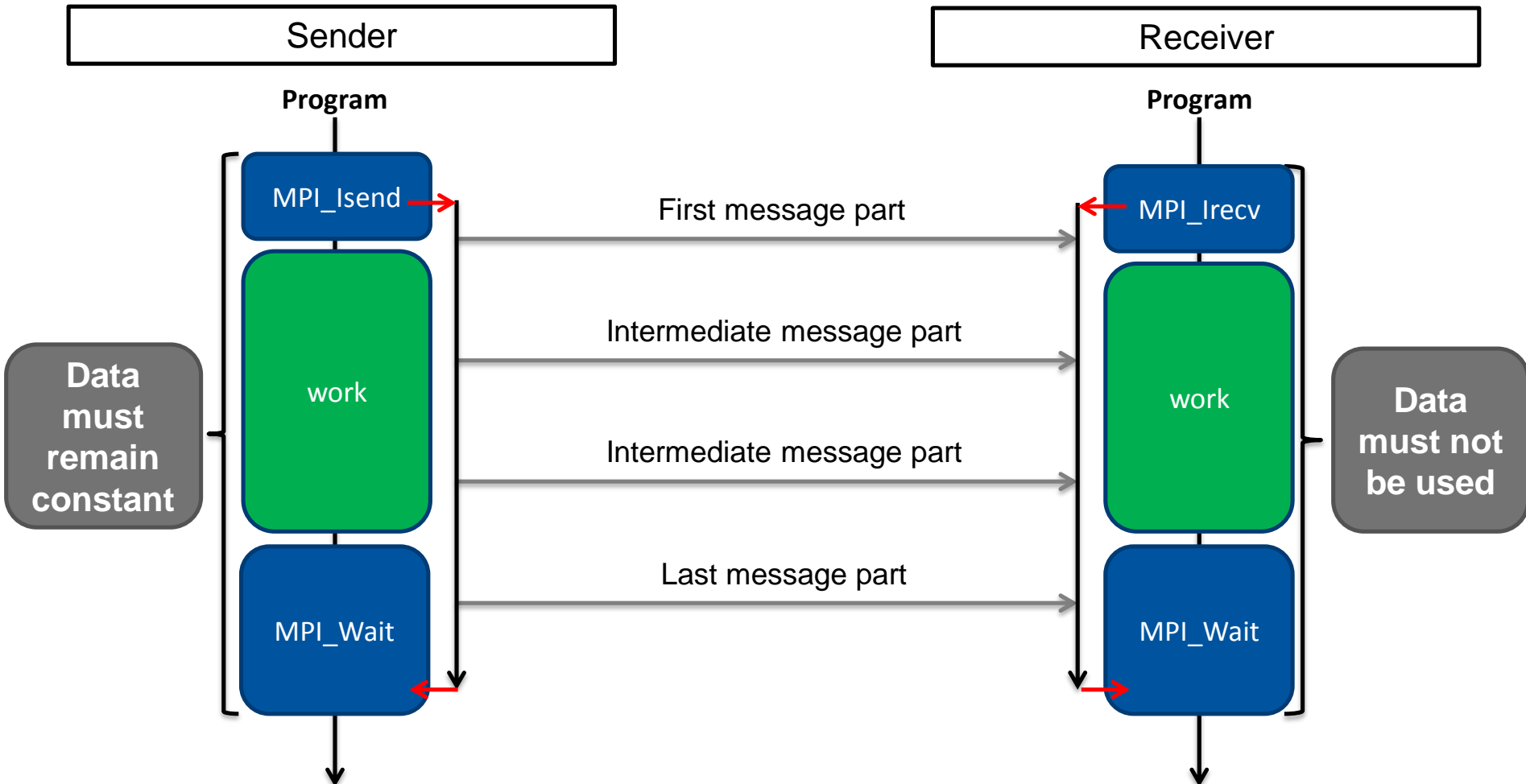
■ Blocking send (w/o buffering) and receive calls:



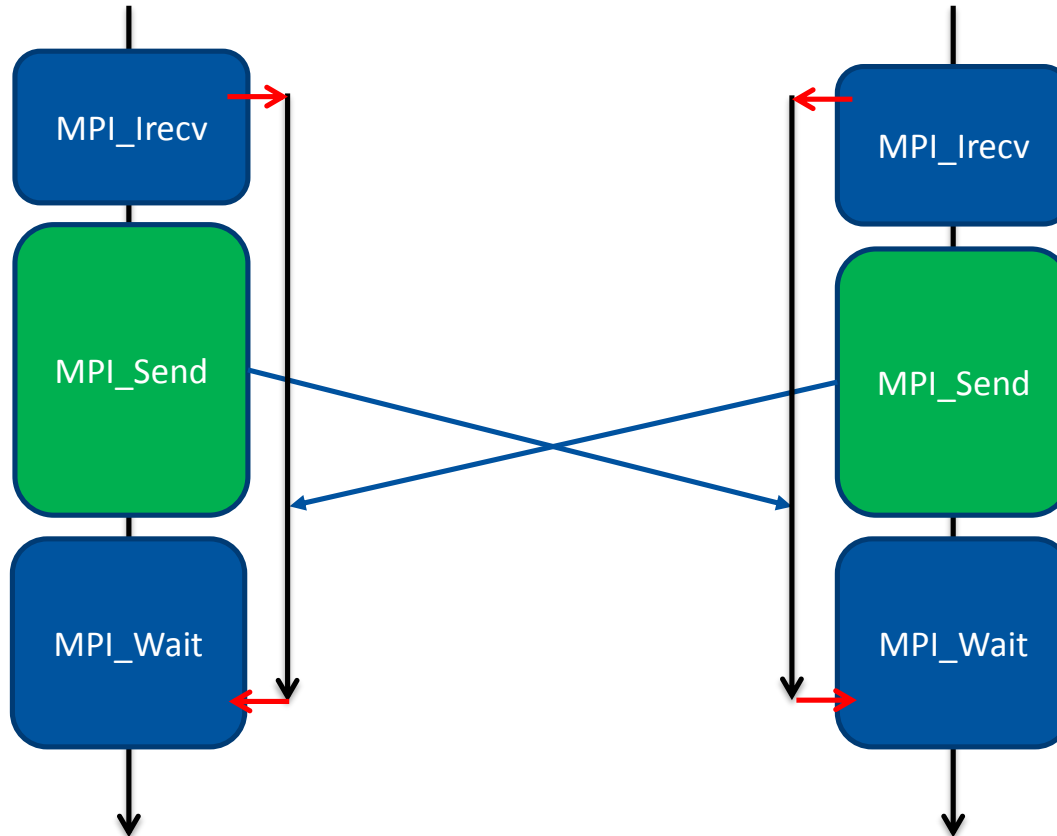
■ Equivalent with non-blocking calls:



Other work can be done in between*:



- Non-blocking operations can be used to prevent deadlocks in symmetric code:



- That is how MPI_Sendrecv is usually implemented

■ Test if given operation has completed:

```
MPI_Test (MPI_Request *request, int *flag, MPI_Status *status)
```

- **flag**: **true** if the operation has completed, otherwise **false**
- **status**: status of the completed operation, only set if **flag** is **true**
- Can be (and usually is) called repeatedly inside a loop
- Upon completion of the operation (i.e. when **flag** is **true**), the operation is freed and the request handle is set to **MPI_REQUEST_NULL**

■ If called with a null request (**MPI_REQUEST_NULL**):

- **MPI_Wait** returns immediately with an empty **status**
- **MPI_Test** sets **flag** to **true** and returns an empty **status**

■ **MPI_Waitany / MPI_Testany**

- Wait for one of the specified requests to complete and free it
- Test if one of the specified requests has completed and free it if it did

■ **MPI_Waitall / MPI_Testall**

- Wait for all the specified requests to complete and free them
- Test if all of the specified requests have completed and free them if they have

■ **MPI_Waitsome / MPI_Testsome**

- Wait for any number of the specified requests to complete and free them
- Test if any number of the specified requests have completed and free these that have

■ **To ignore the status from -all/-some, pass MPI_STATUSES_IGNORE**

- **There are four send modes in MPI:**

- Standard

- Synchronous

- Buffered

- Ready

- **Send modes differ in the relation between the completion of the operation and the actual message transfer**

- **Single receive mode:**

- Synchronous

■ Standard mode

→ The call blocks until the message has either been transferred or copied to an internal buffer for later delivery

■ Synchronous mode

→ The call blocks until a matching receive has been posted and the message reception has started

■ Buffered mode

→ The call blocks until the message has been copied to a user-supplied buffer. Actual transmission may happen at a later point

■ Ready mode (don't use!)

→ The operation succeeds only if a matching receive has already been posted. Behaves as standard send in every other aspect

■ Call names:

- **MPI_Send** blocking standard send
- **MPI_Isend** non-blocking standard send
- **MPI_Ssend** blocking synchronous send
- **MPI_Issend** non-blocking synchronous send
- **MPI_Bsend** blocking buffered send
- **MPI_Ibsend** non-blocking buffered send
- **MPI_Rsend** blocking ready-mode send
- **MPI_Irsend** non-blocking ready-mode send

■ Buffered operations require an explicitly provided user buffer

- **MPI_Buffer_attach** (**void *buf**, **int size**)
- **MPI_Buffer_detach** (**void *buf**, **int *size**)
- Buffer size must account for the envelope size (**MPI_BSEND_OVERHEAD**)

- **One rarely needs anything else except the standard send**
- **The synchronous send can be used to synchronise two ranks**
- **Simple correctness check**
 - Replacing all blocking standard sends with blocking synchronous sends should not result in deadlock
 - If program deadlocks, you are relying on the buffering behaviour of the standard send → change your algorithm
- **Buffered sends guarantee that messages are always buffered, but it is possible to run out of buffer space**
 - No way to test if the buffer is still in use by MPI

■ Attempt to abort all MPI processes in a given communicator:

```
MPI_Abort (MPI_Comm comm, int errorcode)
```

→ **errorcode** is returned to the OS if supported by the implementation.

→ Note: Open MPI does not return the error code to the OS.

■ Portable timer function:

```
double MPI_Wtime ()
```

→ Returns the wall-clock time that has elapsed since an unspecified (but fixed for successive invocations) point in the past

■ Obtain a string ID of the processor:

```
MPI_Get_processor_name (char *name, int *resultlen)
```

→ **name:** buffer of at least **MPI_MAX_PROCESSOR_NAME** characters

→ **resultlen:** length of the returned processor ID (w/o the '\0' terminator)

- MPI can only be initialised once and finalised once for the lifetime of each MPI process

→ Multiple calls to **MPI_Init** or **MPI_Finalize** result in error

- Determine if MPI is already initialised:

```
MPI_Initialized (int *flag)
```

→ **flag** set to **true** if **MPI_Init** was called

- Determine if MPI is already finalised:

```
MPI_Finalized (int *flag)
```

→ **flag** set to **true** if **MPI_Finalize** was called

- Intended for use in parallel libraries built on top of MPI

■ Do not pass pointers to pointers in MPI calls

```
int scalar;
MPI_Send(&scalar, MPI_INT, 1, ...

int array[5];
MPI_Send(array, MPI_INT, 5, ...
... or ...
MPI_Send(&array[0], MPI_INT, 5, ...

int *pointer = new int[5];
MPI_Send(pointer, MPI_INT, 5, ...
... or ...
MPI_Send(&pointer[0], MPI_INT, 5, ...

// ERRONEOUS
MPI_Send(&pointer, MPI_INT, 5, ...
```

&array will work too, but is not recommended

Will result in the value of the pointer itself (i.e. the memory address) being sent, possibly accessing past allocated memory

■ Do not pass pointers to pointers in MPI calls

```
void func (int scalar)
{
    MPI_Send(&scalar, MPI_INT, 1, ...

void func (int& scalar)
{
    MPI_Send(&scalar, MPI_INT, 1, ...

void func (int *scalar)
{
    MPI_Send(scalar, MPI_INT, 1, ...

void func (int *array)
{
    MPI_Send(array, MPI_INT, 5, ...
    ... or ...
    MPI_Send(&array[0], MPI_INT, 5, ...
```

■ Use flat multidimensional arrays; **arrays of pointers do not work**

```
// Static arrays are OK
int mat2d[10][10];
MPI_Send(&mat2d, MPI_INT, 10*10, ...

// Flat dynamic arrays are OK
int *flat2d = new int[10*10];
MPI_Send(flat2d, MPI_INT, 10*10, ...

// DOES NOT WORK
int **p2d[10] = new int*[10];
for (int i = 0; i < 10; i++)
    p2d[i] = new int[10];
MPI_Send(p2d, MPI_INT, 10*10, ...
... or ...
MPI_Send(&p2d[0][0], MPI_INT, 10*10, ...
```

MPI has no way to know that there is a hierarchy of pointers

■ Passing pointer values around makes little to no sense

- Pointer values are process-specific
- No guarantee that memory allocations are made at the same addresses in different processes
 - Especially on heterogeneous architectures, e.g., host + co-processor
- No guarantee that processes are laid out in memory the same way, even when they run on the same host
 - Address space layout randomisation
 - Stack and heap protection

■ Relative pointers could be passed around

- **Non-contiguous array sections should not be passed to non-blocking MPI calls**

```
INTEGER, DIMENSION(10,10) :: mat
```

```
! Probably OK
```

```
CALL MPI_Isend(mat(:,1:3), ...
```

```
! NOT OK
```

```
CALL MPI_Isend(mat(1:3,:), ...
```

```
! NOT OK
```

```
CALL MPI_Isend(mat(1:3,1:3), ...
```

A temporary contiguous array is created and passed to MPI. It might get destroyed on return from the call before the actual send is complete!

- **Solved in MPI-3.0 with the introduction of the new Fortran 2008 interface *mpi_f08*, which allows array sections to be passed**

