Message Passing with MPI

PPCES 2015

Hristo Iliev
IT Center / JARA-HPC

Agenda

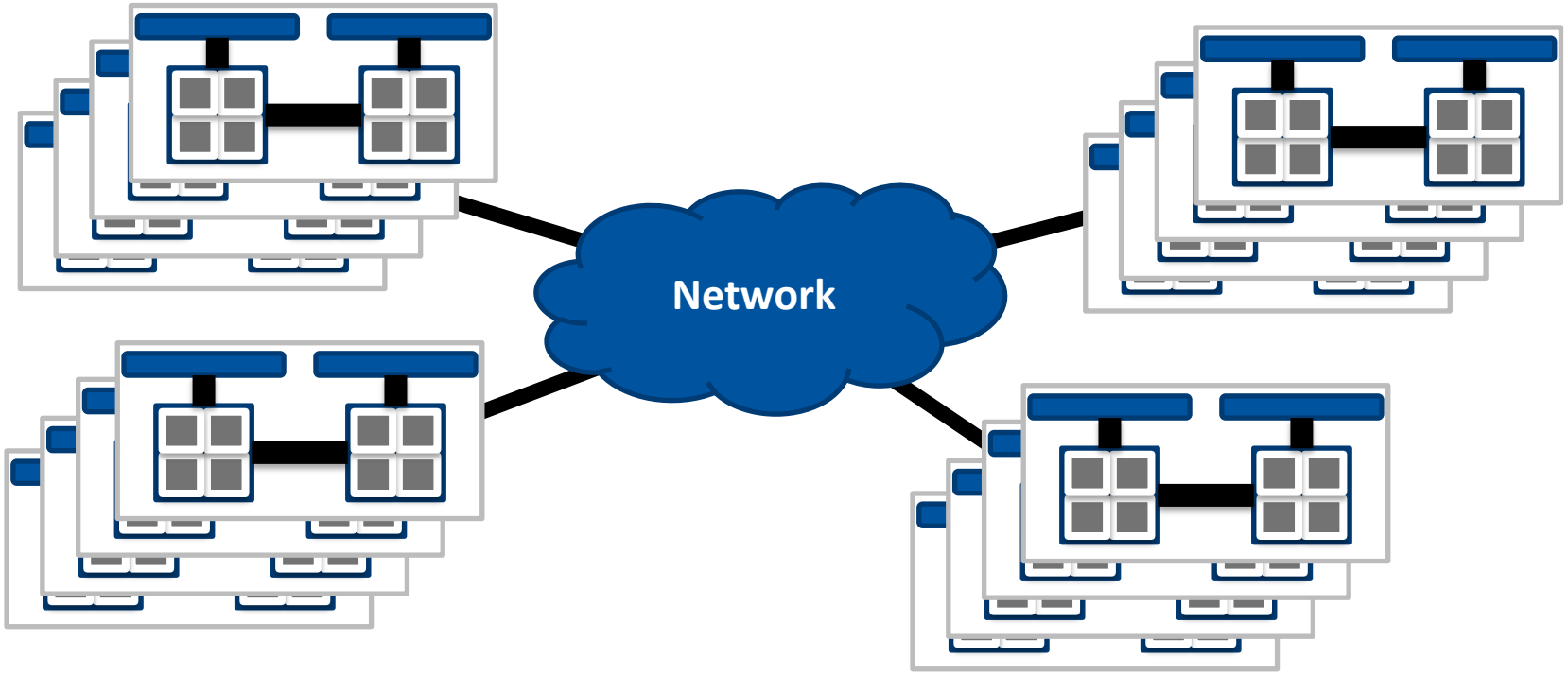
- **Motivation**
- **Part 1**
  - → Concepts
  - → Point-to-point communication
  - → Non-blocking operations
- **Part 2**
  - → Collective operations
  - → Communicators
  - → User datatypes
- **Part 3**
  - → Hybrid parallelization
  - → Common parallel patterns

Agenda

- **Motivation**
- **Part 1**
  - → Concepts
  - → Point-to-point communication
  - → Non-blocking operations
- **Part 2**
  - → Collective operations
  - → Communicators
  - → User datatypes
- **Part 3**
  - → Hybrid parallelization
  - → Common parallel patterns

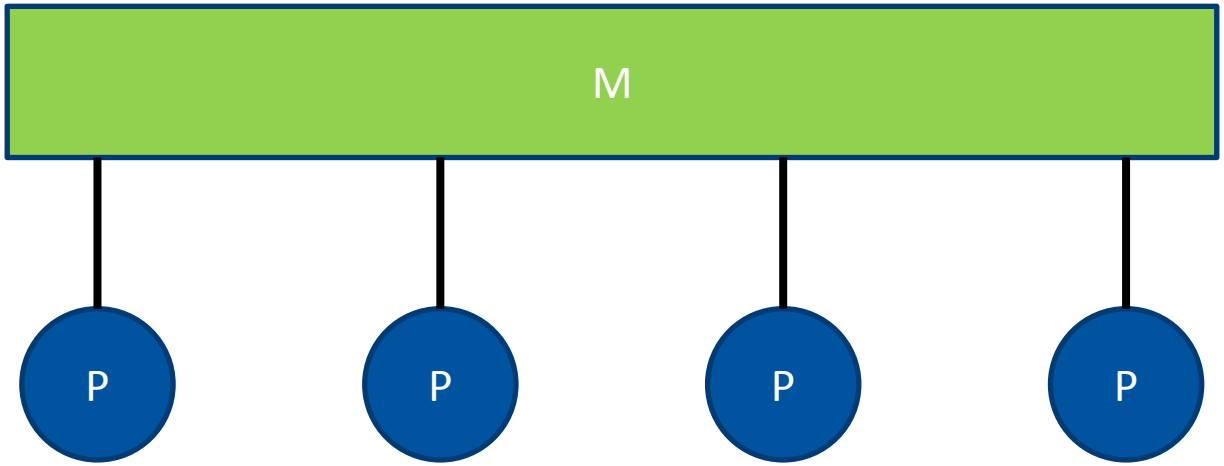
Parallel Architectures

- **Clusters**
  - → HPC market is at large dominated by distributed memory *multicomputers* and *clusters*
  - → Nodes have no direct access to other nodes' memory and usually run their own (possibly stripped down) copy of the OS



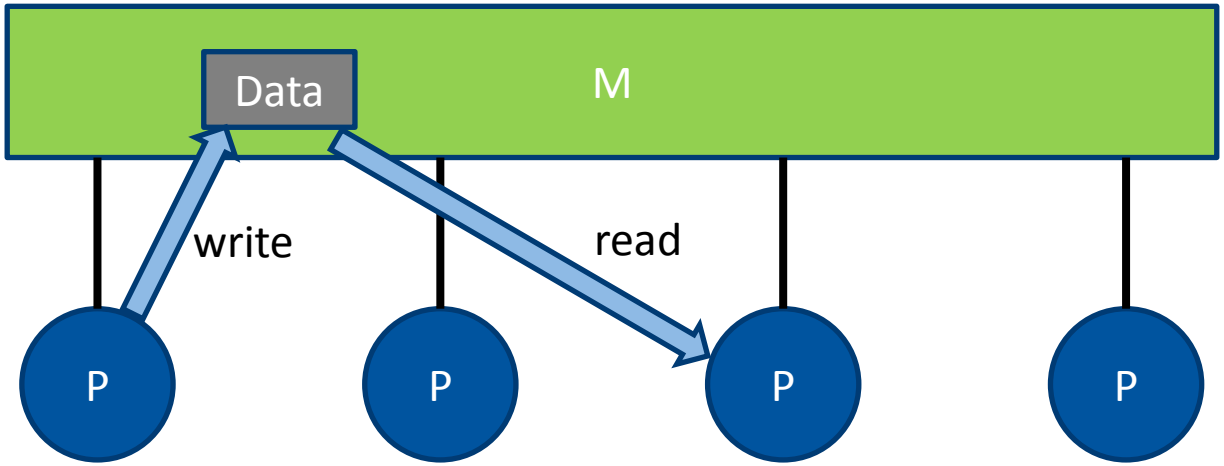
Parallel Architectures

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



Parallel Architectures

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



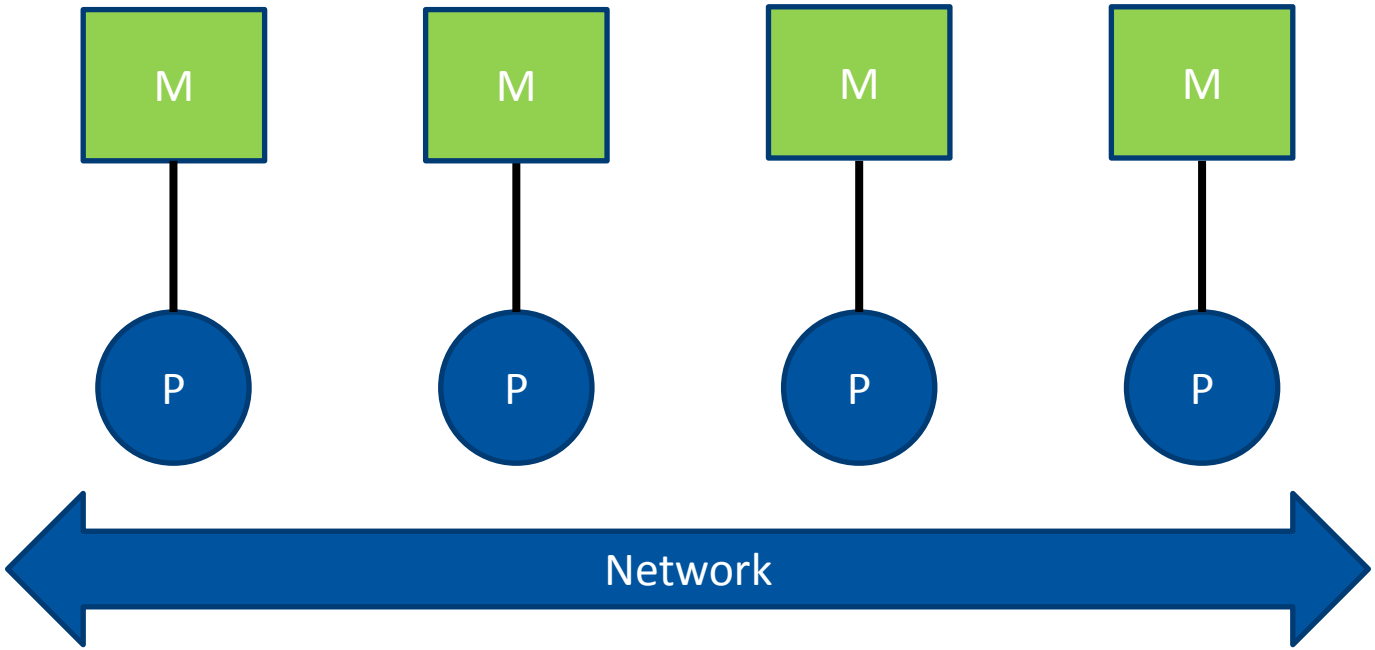
  - → Data exchange is achieved through the means of read/write operations in the global address space

Parallel Architectures

- **Shared Memory**
  - → Shared Memory – all processing elements (P) have direct access to the main memory (M)
  - → Data exchange is achieved through the means of read/write operations in the global address space
  - → Programs typically consist of a set of OS entities that share single virtual address space – *threads*
  - → Shared (global) variables
    - → Data races are possible
    - → Synchronisation needed to enforce read/write order – barriers, semaphores, locks, etc.

Parallel Architectures

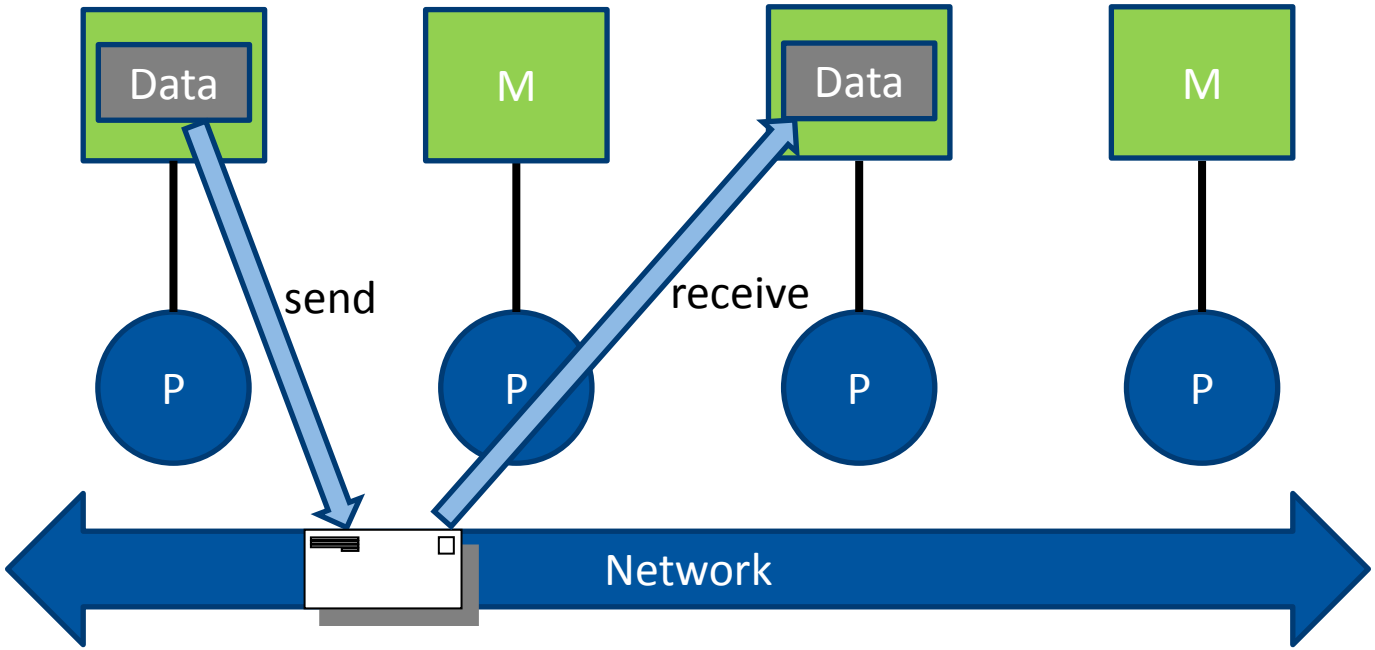
- **Distributed Memory**
  - → Each processing element (P) has its own main memory block (M)



Parallel Architectures

Distributed Memory

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



→ Data exchange is achieved via message passing over a network

Parallel Architectures

- **Distributed Memory**
  - → Each processing element (P) has its own main memory block (M)
  - → Data exchange is achieved via message passing over a network
  - → Message passing could be either explicit (MPI) or implicit (PGAS)
  - → Programs typically consist of a set of OS entities that have their own virtual address spaces – *processes*
  - → No shared variables
    - → No data races
    - → Synchronisation typically results as a by-product of the send-receive semantics

The Process Space

- **Def.: A process is a running in-memory instance of an executable**
  - → Executable code: e.g. binary machine code
  - → Memory: heap, stack, process state (incl. CPU registers)
  - → One or more threads of execution
  - → Operating system context (e.g. signals, I/O handles, etc.)
  - → Virtual address space
  - → PID
- **Important:**
  - → Isolation and protection: One process cannot interoperate with other processes or access their contexts without the help of the operating system
    - → No direct inter-process data exchange
    - → No direct inter-process synchronization

Inter-Process Communication

- **Interaction with other processes**
  - → Shared memory
    - → Restricted to the same machine / system board
  - → 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
    - → **MPI**, PVM – tightly coupled
    - → Globus Toolkit (Grid infrastructure) – loosely coupled
    - → BOINC (SETI@home, Einstein@home, *@home) – decoupled

IPC Example: network sockets

- **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

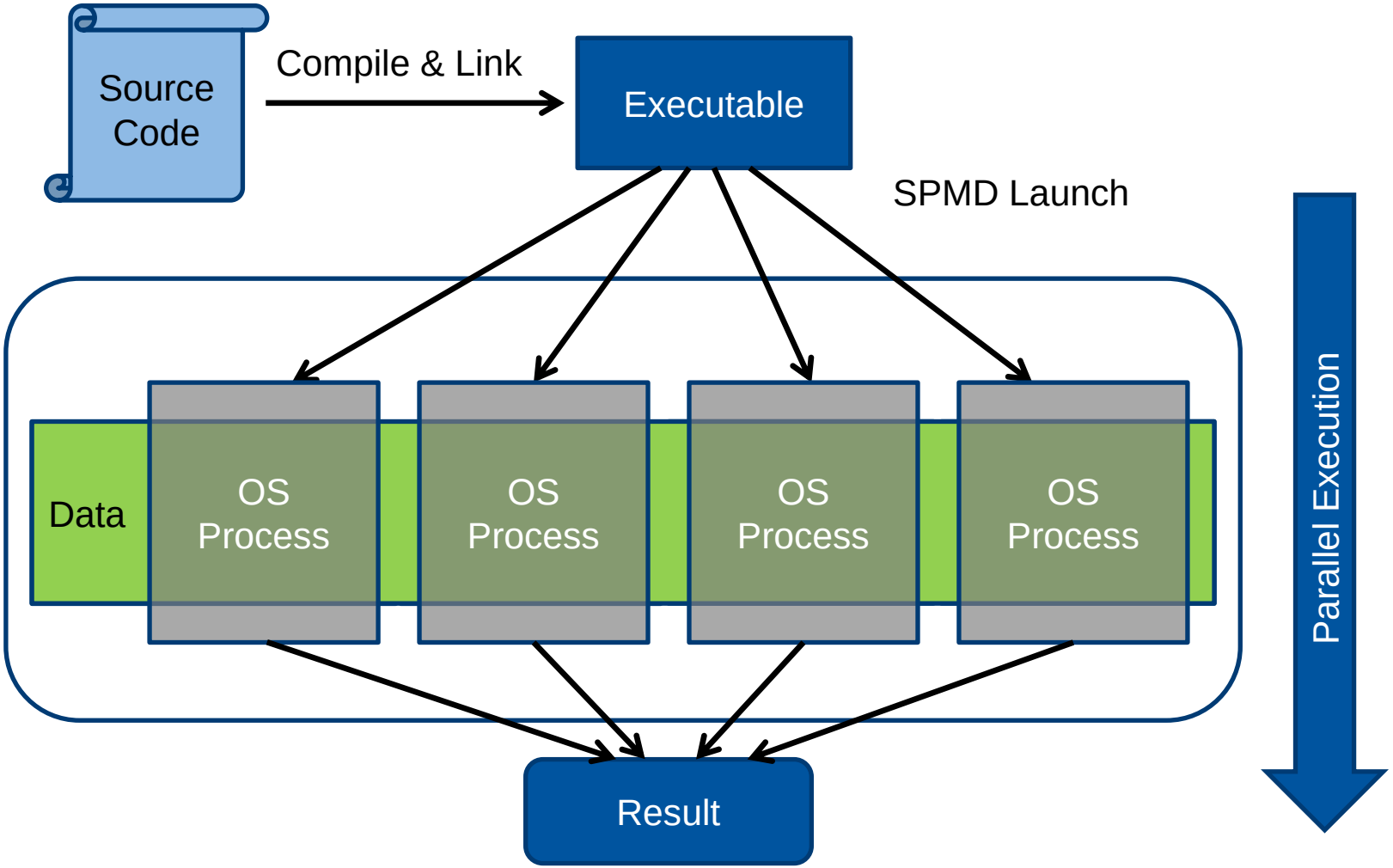
SPMD Model

- **Abstractions make programming and understanding easier**
- **Single Program Multiple Data**
  - → Multiple instruction flows (instances) from the Same 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 Model

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



SPMD Environments

- **Provide identification of all participating processes**
  - → Who is also working on this problem?
- **Provide robust mechanisms to exchange data**
  - → Whom to send data to?
  - → How much data?
  - → What kind of data?
  - → Has the data arrived?
- **Provide a synchronization mechanism**
  - → Are all processes at the same point in the program execution flow?
- **Provide method to launch and control a set of processes**
  - → How do we start multiple processes and get them to work together?

➡ **MPI**

Going Parallel

- **Define the problem**
  - → problem space
  - → methods
- **Solution:**
  - → input domain
  - → algorithm
  - → output domain
- **Decomposition**
  - → Split the input domain into (preferably non-interacting) regions

Agenda

- Motivation
- **Part 1**
  - → Concepts
  - → Point-to-point communication
  - → Non-blocking operations
- Part 2
  - → Collective operations
  - → Communicators
  - → User datatypes
- Part 3
  - → Hybrid parallelization
  - → Common parallel patterns

MPI

- **Message Passing Interface**
  - → The de-facto standard API for explicit message passing nowadays
  - → A moderately large standard (852 pages in printed book format)
  - → Maintained by the 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 describe the interaction (communication) within applications for distributed memory machines**
- **MPI provides source level portability of parallel applications between different hardware platforms**

MPI History

- **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
  - → Fortran 2008 bindings; C++ deleted from the standard

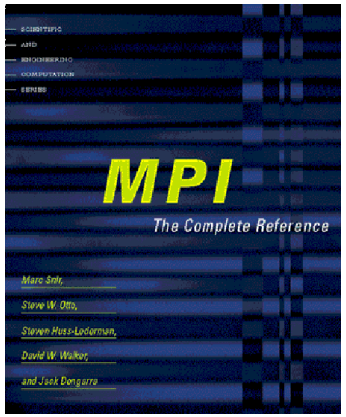
Literature about MPI

- **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

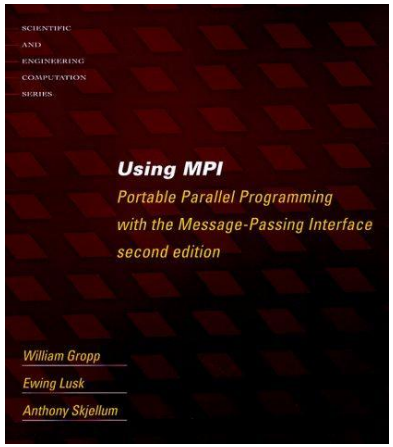
  2nd edition, The MIT Press, 1998

Literature about MPI

- **Using MPI**

  by William Gropp, Ewing Lusk, Anthony Skjellum

  The MIT Press, Cambridge/London, 1999



- **Using MPI-2**

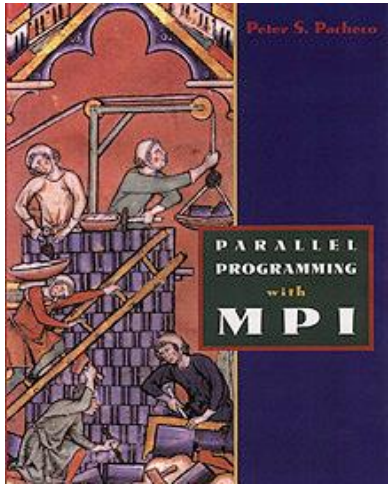
  by William Gropp, Ewing Lusk, Rajeev Thakur

  The MIT Press, Cambridge/London, 2000

- **Parallel Programming with MPI**

  by Peter Pacheco

  Morgan Kaufmann Publishers, 1996



More Information & Documentation

- **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**
  - → http://www.open-mpi.org/
- **Our MPI-related WEB page with further links (German only)**
  - → http://www.rz.rwth-aachen.de/mpi/
- **Manual pages**
  - → man MPI
  - → man MPI_Xxx_yyy_zzz (for all MPI calls)

MPI Documentation

- **Language Independent Specification (LIS)**
  - → e.g. **MPI_SEND(data, count, dtype, …)**
  - → Describes the name of the MPI operation and its arguments
- **Language bindings**
  - → e.g. **int MPI_Send(void *data, int count, MPI_Datatype dtype, …)**
  - → Describe the interface to each MPI operation in a given language
  - → MPI officially supports C and Fortran
- **Documentation is usually cross-implementation compatible**
  - → Simply search online

MPI Naming Conventions

- **C naming convention**
  - → **MPI_CONSTANTS_ALL_CAPS**
  - → **MPI_Calls_named_like_this**
- **Fortran naming convention**
  - → Fortran 77
    - → **MPI_ALL_CAPS_SINCE_FORTRAN_IS_NOT_CASE_SENSITIVE**
  - → Fortran 90+
    - → **MPI_C_like_case_but_fortran_still_not_case_sensitive**
- **Unix manual pages usually follow the C naming convention**
  - → Important since most Unix-like Oses have case sensitive file systems
    - → Exception: HFS+ on OS X (case-insensitive *by default*)

MPI Basics – Agenda

- **MPI Basics**
  - → Startup, initialization, finalization and shutdown
  - → Send and receive
  - → Message envelope
- **Point-to-Point Communication**
  - → Basic MPI data types
  - → Combining send and receive
  - → Non-blocking operations
  - → Send modes
  - → Common mistakes

General Structure of MPI Programs

Startup, initialization, finalization and shutdown – C/C++

C/C++

```
1  #include <mpi.h>

   int main(int argc, char **argv)
2  {
     … some code …
3    MPI_Init(&argc, &argv);



4    … computation & communication …



5    MPI_Finalize();
     … wrap-up …
6    return 0;
   }
```

1. Inclusion of the MPI header file
2. Non-coordinated running mode: serial
   - **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 code
   Typically computation and communication
5. Termination of the MPI environment
   Internal buffers are flushed
6. Non-coordinated running mode: serial
   - **NO MPI function calls allowed afterwards with few exceptions**

General Structure of MPI Programs

Startup, initialization, finalization and shutdown – Fortran

Fortran

```
program example
use mpi

 … some code …

call MPI_INIT(ierr)



… computation  & communication …



call MPI_FINALIZE(ierr)

… wrap-up …
end
```

1. Inclusion of the MPI module
2. Non-coordinated running mode: serial
   - **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 code
   Typically computation and communication
5. Termination of the MPI environment
   Internal buffers are flushed
6. Non-coordinated running mode: serial
   - **NO MPI function calls allowed afterwards with few exceptions**

General Structure of MPI Programs

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

C/C++

```
#include <mpi.h>

int main(int argc, char **argv)
{
  … some code …
  MPI_Init(&argc, &argv);
  … other code …
① MPI_Comm_size(MPI_COMM_WORLD,
      &numberOfProcs);
② MPI_Comm_rank(MPI_COMM_WORLD,
      &rank);
  … computation  & communication …
  MPI_Finalize();
  … wrapup …
  return 0;
}
```

① Obtain the number of processes (ranks) in the MPI program instance

E.g. if there are 2 processes running then **numberOfProcs** will contain 2 after the call

② Obtain the identity of the calling process in the MPI program
Note: MPI processes are numbered starting from "0"

E.g. if there are 2 processes running then **rank** will be "0" in the first process and "1" in the second process after the call

General Structure of MPI Programs

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

Fortran

```
program example
use mpi
integer :: rank, numberOfProcs, ierr
 … some code …
call MPI_INIT(ierr)
… other code …
① call MPI_COMM_SIZE(MPI_COMM_WORLD,
        numberOfProcs, ierr)
② call MPI_COMM_RANK(MPI_COMM_WORLD,
        rank, ierr)
… computation  & communication …
call MPI_FINALIZE(ierr)
… wrap-up …
end program example
```

① Obtain the number of processes (ranks) in the MPI program instance

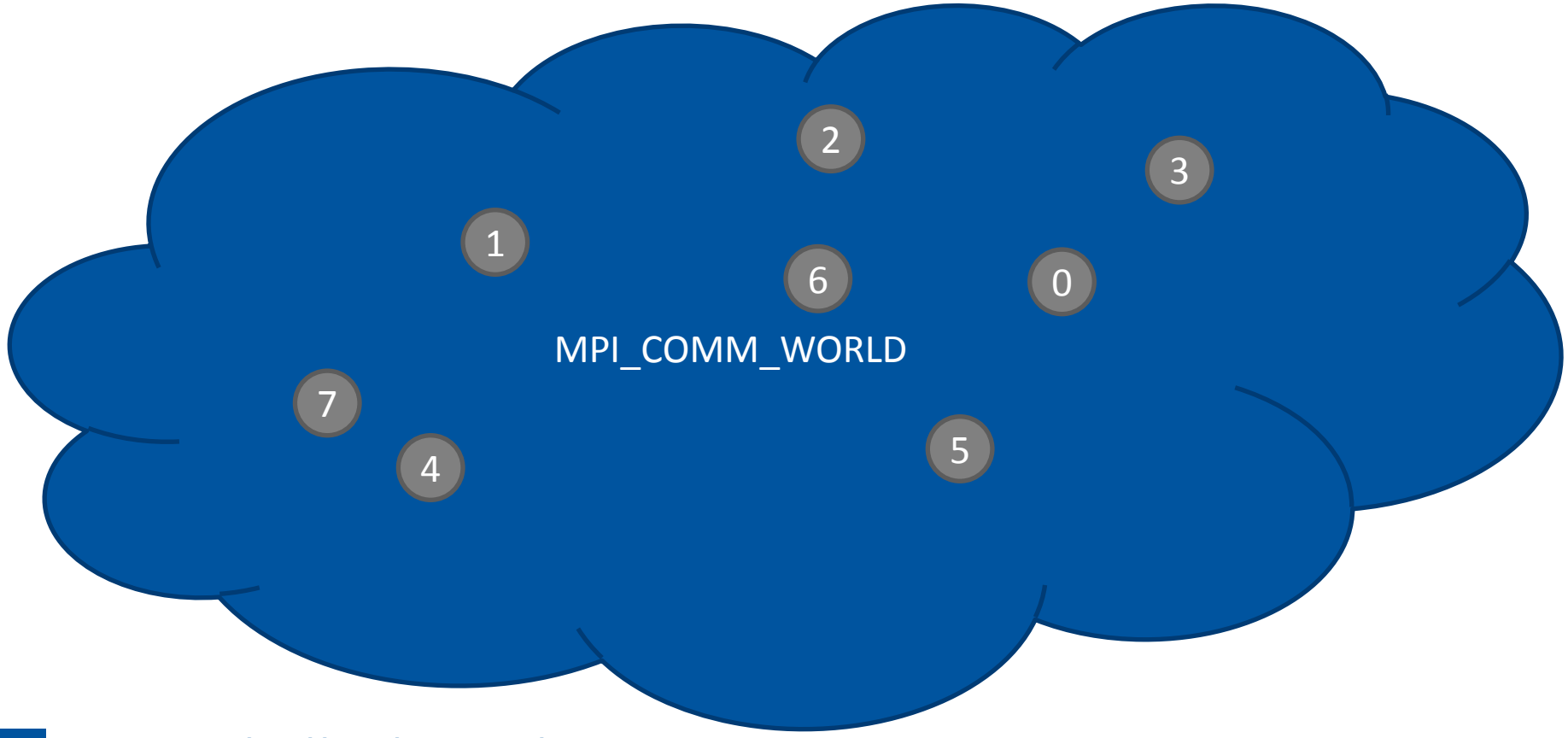
E.g. if there are 2 processes running then **numberOfProcs** will contain 2 after the call

② Obtain the identity of the calling process in the MPI program
Note: MPI processes are numbered starting from "0"

E.g. if there are 2 processes running then **rank** will be "0" in the first process and "1" in the second process after the call

Ranks

- **Processes in an MPI program are initially indistinguishable**
- **MPI_Init assigns each process a unique identity – rank**
  - → Ranks range from 0 up to the number of processes minus 1



MPI as an SPMD Environment

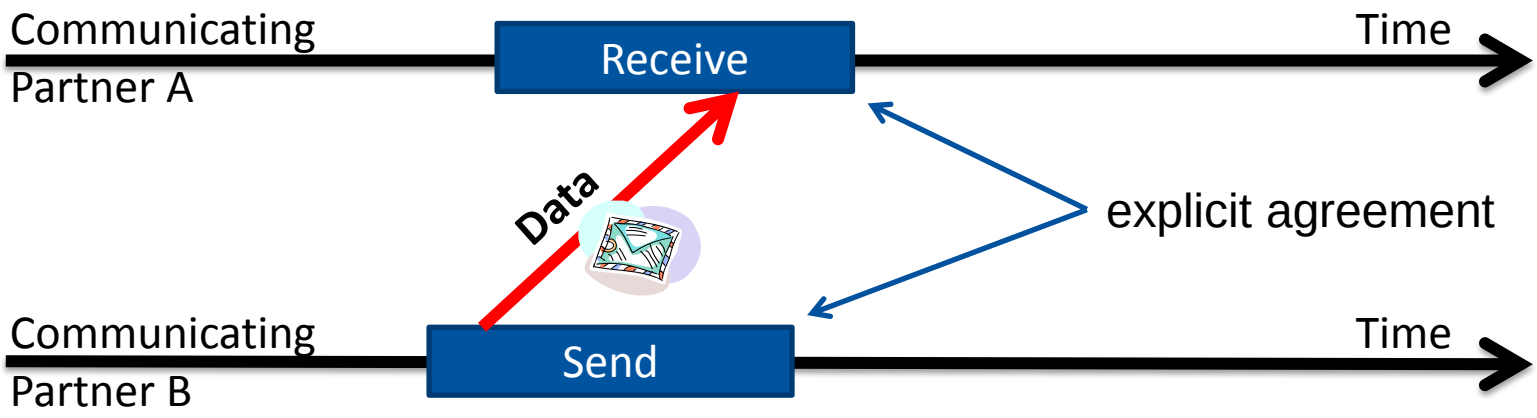
- **Provide identification of all participating processes**
  - → Who am I and who is also working on this problem?
- **Provide robust mechanisms to exchange data**
  - → Whom to send data to?
  - → How much data?
  - → What kind of data?
  - → Has the data arrived?
- **Provide a synchronization mechanism**
  - → Are all processes at the same point in the program execution flow?
- **Provide method to launch and control a set of processes**
  - → How do we start multiple processes and get them to work together?

Agenda

- Motivation
- **Part 1**
  - → Concepts
  - → Point-to-point communication
  - → Non-blocking operations
- Part 2
  - → Collective operations
  - → Communicators
  - → User datatypes
- Part 3
  - → Hybrid parallelization
  - → Common parallel patterns

Message Passing

- **Recall: the goal is to enable communication between processes that share no memory**



- **Explicit message passing requires:**
  - → Send and Receive primitives (operations)
  - → Addresses of both sender and receiver
  - → Specification of what has to be send/received

Sending Data

- **To send a single message:**

What?

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

To whom?

C/C++

→ **data:** location in memory of the data to be sent

→ **count:** number of data elements to be sent

→ **type:** MPI datatype of the data elements

→ **dest:** rank of the receiver

→ **tag:** additional identification of the message (color, tag, etc.)

ranges from 0 to UB (impl. dependant but no less than 32767)

→ **comm:** communication context (communicator)

```
MPI_SEND (DATA, COUNT, TYPE, DEST, TAG, COMM, IERR)
```

Fortran

Receiving Data

- **To receive a single message:**

What?

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

C/C++

From whom?

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

```
MPI_RECV (DATA, COUNT, TYPE, SRC, TAG, COMM, STATUS, IERR)
```

Fortran

MPI Datatypes

- **MPI is a library – it cannot infer on the type of elements in the supplied buffer at run time and that's why it has to be told what it is**
  - → C++ wrapper APIs usually infer the MPI datatype from the arguments type
- **MPI datatypes tell MPI how to:**
  - → read actual memory values from the send buffer
  - → write actual memory values into the receive buffer
  - → convert between machine representations in heterogeneous environments
- **MPI_BYTE is used to send and receive data as-is without any conversion.**
- **MPI datatypes must match the language data type of the data array**
- **MPI datatypes are handles and cannot be used to declare variables**

MPI Datatypes

- **MPI provides many predefined datatypes for each language binding:**
  - → Fortran

MPI data type	Fortran data type
MPI_INTEGER	INTEGER
MPI_REAL	REAL
MPI_DOUBLE_PRECISION	DOUBLE PRECISION
MPI_COMPLEX	COMPLEX
MPI_LOGICAL	LOGICAL
MPI_CHARACTER	CHARACTER(1)
MPI_BYTE	-
…	…

  - → C/C++
  - → User-defined data types

8 binary digits
no conversion

MPI Datatypes

- **MPI provides many predefined datatypes for each language binding:**
  - → Fortran
  - → C/C++

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

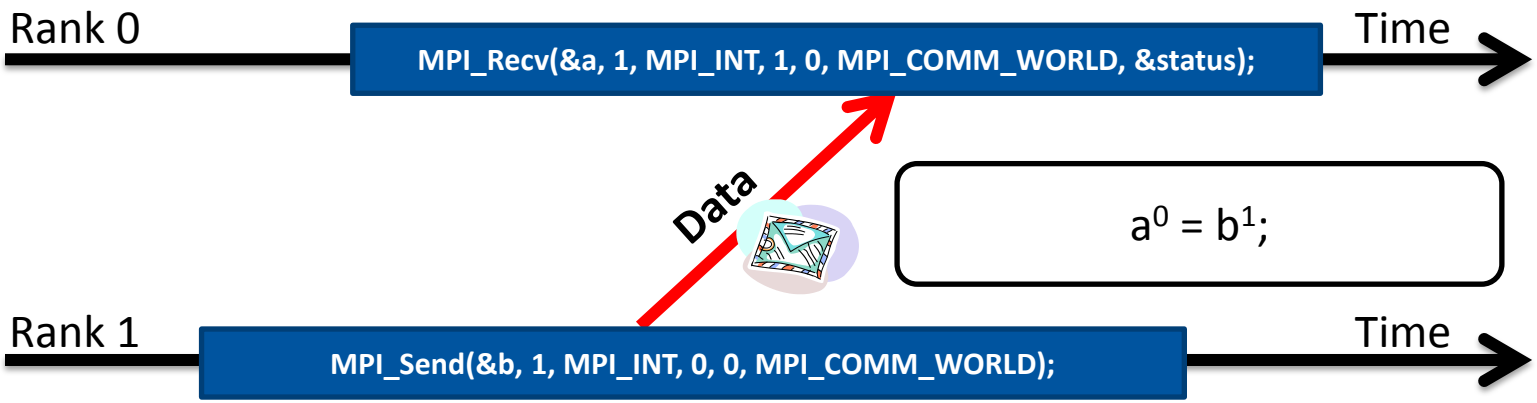
  - → User-defined data types

Return Values and Error Handling

- **Almost every C/C++ MPI call returns an integer error code:**
  - → **int MPI_Send(…)**
- **Fortran MPI calls take an extra INTEGER output argument (<u>always last in the argument list</u>) where the error code is returned:**
  - → **SUBROUTINE MPI_SEND(…, ierr)**
- **Error codes indicate the success of the operation:**
  - → C/C++ **MPI_SUCCESS == MPI_Send(…)**
  - → Fortran: **call MPI_SEND(…, ierr)**
    ierr .eq. MPI_SUCCESS
  - → Failure indicated by error code different from **MPI_SUCCESS**
- **If an error occurs, an MPI error handler is called before the operation returns. The default MPI error handler aborts the MPI job!**
- **Note: MPI error code values are implementation specific.**

Message Passing: The MPI Way

- **Message passing in MPI is explicit:**



- **These two calls transfer the value of variable *b* in rank 1 into variable *a* in rank 0.**

- **For now assume that *comm* is fixed as MPI_COMM_WORLD. We will talk about communicators later on.**

MPI as an SPMD Environment

- **Provide identification of all participating processes**
  - ✓ Who am I and who is also working on this problem?
- **Provide robust mechanisms to exchange data**
  - ✓ Whom to send data to?
  - ✓ How much data?
  - ✓ What kind of data?
  - ✗ Has the data arrived? (only the receiver knows)
- **Provide a synchronization mechanism**
  - → Are all processes at the same point in the program execution flow?
- **Provide method to launch and control a set of processes**
  - → How do we start multiple processes and get them to work together?

Complete MPI Example

C/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. Initialize the MPI library
2. Get entity identification
3. Do smth different in each/a rank
4. Communicate
5. Clean up the MPI library

Compiler Wrappers

- **MPI is a typical library – header files, object code archives, etc.**
- **For convenience some MPI vendors provide compiler wrappers:**

`cc`	→	`mpicc`
`c++`	→	`mpic++`
`f90`	→	`mpif90`

- **RWTH Aachen specific (dependent on the loaded compiler module):**

`$CC`	→	`$MPICC`
`$CXX`	→	`$MPICXX`
`$FC`	→	`$MPIFC`

Compiler Wrappers

- **MPI is a typical library – header files, object code archives, etc.**
- **For convenience some MPI vendors provide compiler wrappers:**

```
cluster:~[1]$ $MPICC --show
icc
-I/opt/MPI/openmpi-1.5.3/linux/intel/include
-I/opt/MPI/openmpi-1.5.3/linux/intel/include/openmpi
-fexceptions
-pthread
-I/opt/MPI/openmpi-1.5.3/linux/intel/lib
-Wl,-rpath,/opt/MPI/openmpi-1.5.3/linux/intel/lib
-I/opt/MPI/openmpi-1.5.3/linux/intel/lib
-L/opt/MPI/openmpi-1.5.3/linux/intel/lib
-lmpi
-ldl
-Wl,--export-dynamic
-lnsl
-lutil
```

Program Launch

- **MPI applications are usually started via 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:**
  → E.g. on IBM BG/Q: **runjob --np 1024 …**

- **RWTH Aachen specific:**
  → interactive mode

```
$MPIEXEC –n <numprocs> … <program> <arg1> <arg2> <arg3> …
```

  → batch jobs

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

MPI as an SPMD Environment

- **Provide identification of all participating processes**
  - ✓ Who am I and who is also working on this problem?
- **Provide robust mechanisms to exchange data**
  - ✓ Whom to send data to?
  - ✓ How much data?
  - ✓ What kind of data?
  - ✗ Has the data arrived?
- **Provide a synchronization mechanism**
  - → Are all processes at the same point in the program execution flow?
- **Provide method to launch and control a set of processes**
  - ✓ How do we start multiple processes and get them to work together?

MPI Basics Demo

- **Compile and Run a Simple MPI Program**

Message Envelope and Matching

- **Reception of MPI messages is done after matching their envelope**
- **MPI_Send**

```
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

- **MPI_Recv**

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

Message Envelope and Matching

- **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

- **For a receive to complete a matching send has to be posted and v.v.**
- **Beware: sends and receives are atomic**

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 …

Incomplete

Message Reception and Status

- **The receive buffer must be able to hold the whole message**
  - → send count ≤ receive count -> **OK** (but check status)
  - → send count > receive count -> **ERROR** (message truncated)

- **The MPI status object holds information about the received message**

- **C/C++: MPI_Status status;**
  - → **status.MPI_SOURCE** message source rank
  - → **status.MPI_TAG** message tag
  - → **status.MPI_ERROR** receive status code

- **Fortran**

Message Reception and Status

- **The receive buffer must be able to hold the whole message**
  - → send count ≤ receive count -> **OK** (but check status)
  - → send count > receive count -> **ERROR** (message truncated)

- **The MPI status object holds information about the received message**

- **C/C++**

- **Fortran: INTEGER, DIMENSION(MPI_STATUS_SIZE) :: status**
  - → **status(MPI_SOURCE)** message source rank
  - → **status(MPI_TAG)** message tag
  - → **status(MPI_ERROR)** receive status code

Inquiry Operations

- **Blocks until a matching message becomes available**

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

  → Message is not received, one should issue **MPI_Recv** to receive it

  → Information about the message is stored in the status field

- **Check for any message in a given context:**

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

- **Status inquiry:**

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

  → Computes how many integral **datatype** elements can be extracted from the message referenced by **status**

  → If the number is not integral **count** is set to **MPI_UNDEFINED**

Operation Completion

- **MPI operations complete then, when the message buffer is no longer in use by the MPI library and is free for reuse**

- **Send operations complete:**
  - → once the message is constructed *and*
    - → put entirely on the network *or*
    - → buffered entirely

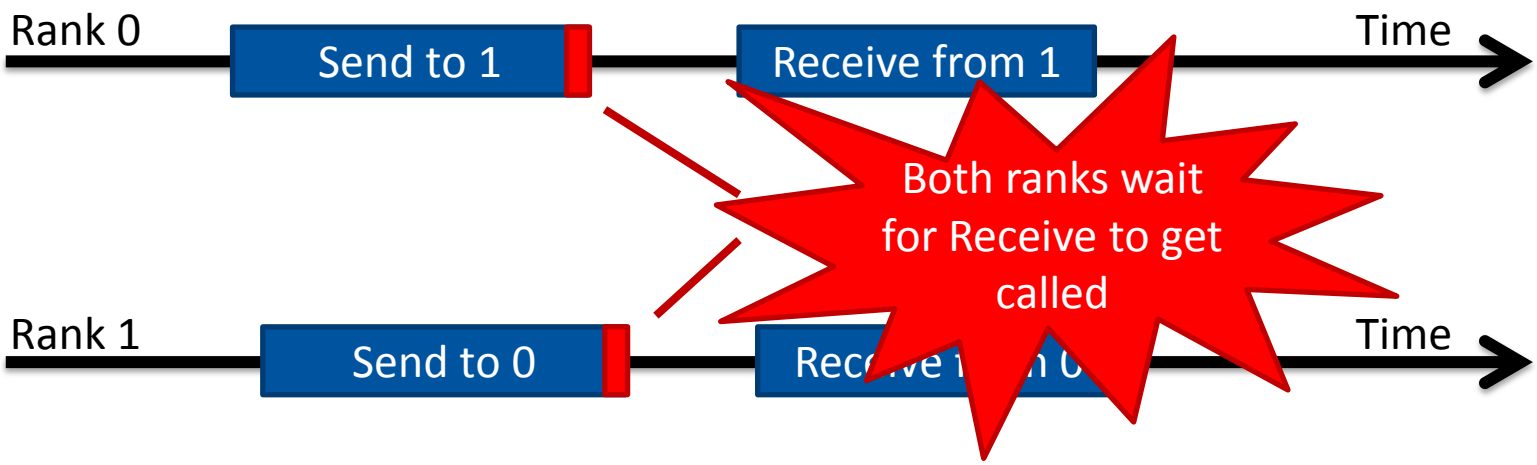
- **Receive operations complete:**
  - → once the entire message has arrived

- **Blocking MPI calls only return once the operation has completed**

Deadlocks

- **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 internally buffered *(implementation-specific!!!)* and therefore return before the message is actually sent over the network

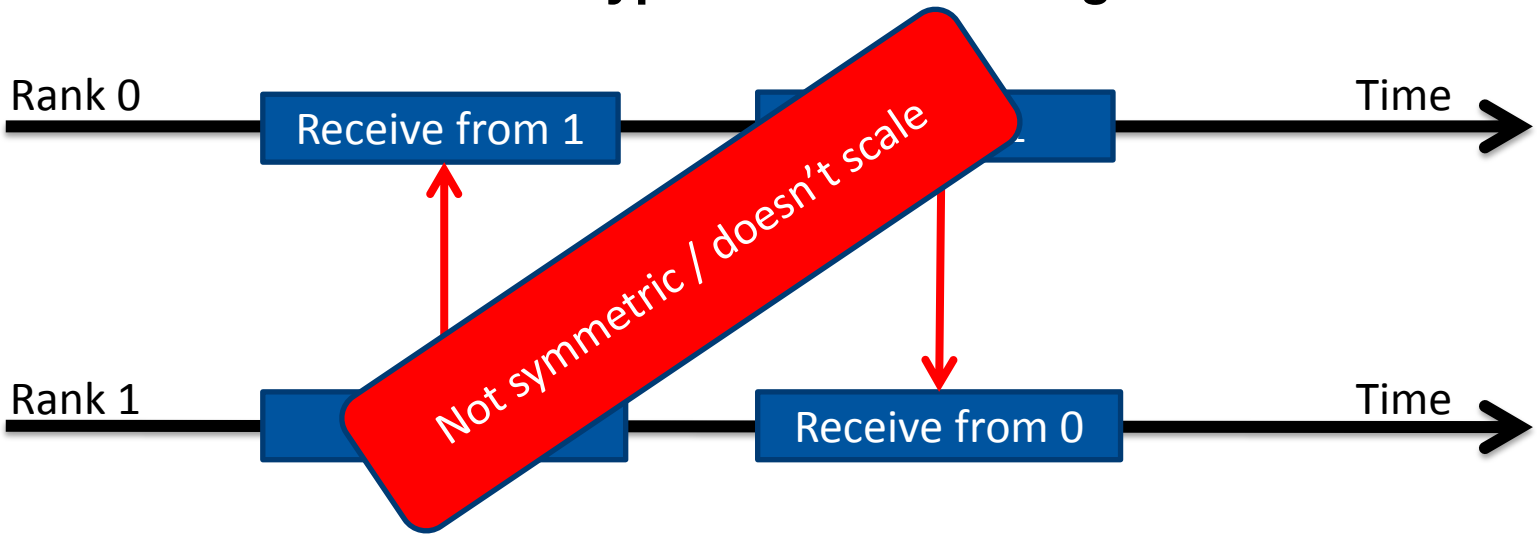
- **Deadlock in a typical data exchange scenario:**



Deadlocks

- **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 internally buffered *(implementation-specific!!!)* and therefore return before the message is actually sent over the network

- **Deadlock resolution in a typical data exchange scenario:**



Combining Send and Receive

```
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)
```

- **Guaranteed to not cause deadlocks with matching peers**

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

Combining Send and Receive

```
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 evading deadlocks (unless unmatched).**
- **Send and receive buffers <u>must not</u> 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)
```

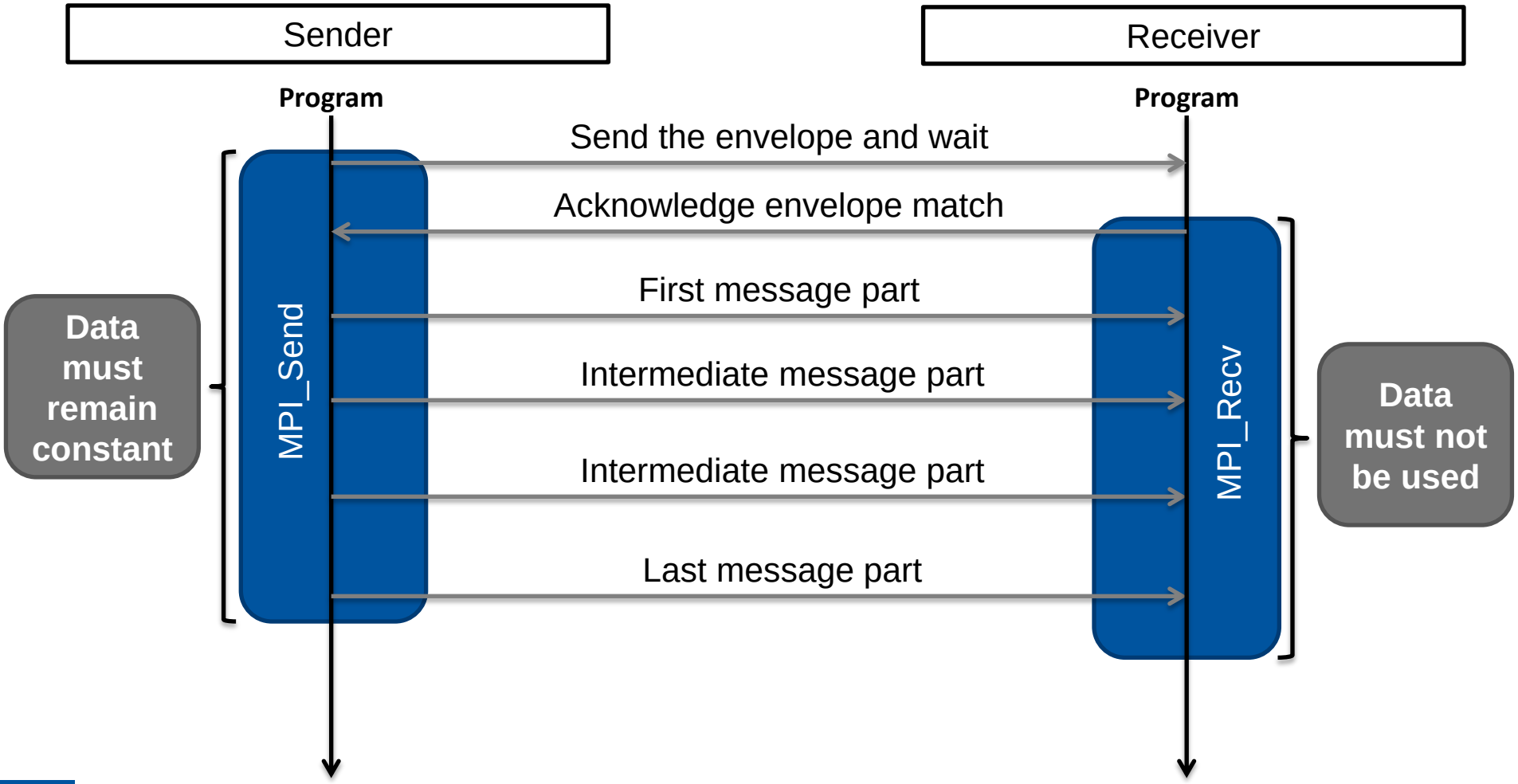
- **First sends a message to *dest*, then receives a message from *source*, using <u>the same memory location, elements count and datatype</u> for both operations.**

Agenda

- Motivation
- **Part 1**
  - → Concepts
  - → Point-to-point communication
  - → Non-blocking operations
- Part 2
  - → Collective operations
  - → Communicators
  - → User datatypes
- Part 3
  - → Hybrid parallelization
  - → Common parallel patterns

Blocking Calls

- **Blocking send (w/o buffering) and receive operate as follows:**



Non-Blocking Calls

- **Non-blocking MPI calls return immediately while the communication operation continue asynchronously in the background.**

- **Each asynchronous operation is represented by a request handle:**
  - → C/C++: **MPI_Request**
  - → Fortran: **INTEGER**

- **Asynchronous operations are progressed by certain MPI events but most notably by the *test* and *wait* MPI calls.**
- **Blocking MPI calls are equivalent to posting an asynchronous call and waiting for it to complete immediately afterwards.**

- **Can be used to overlay communication and computation when computation does not involve the data being communicated.**

Non-Blocking Send and Receive

- **Initiation of non-blocking (async) 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 posted operation

- **Blocking wait for the completion of an asynchronous operation:**

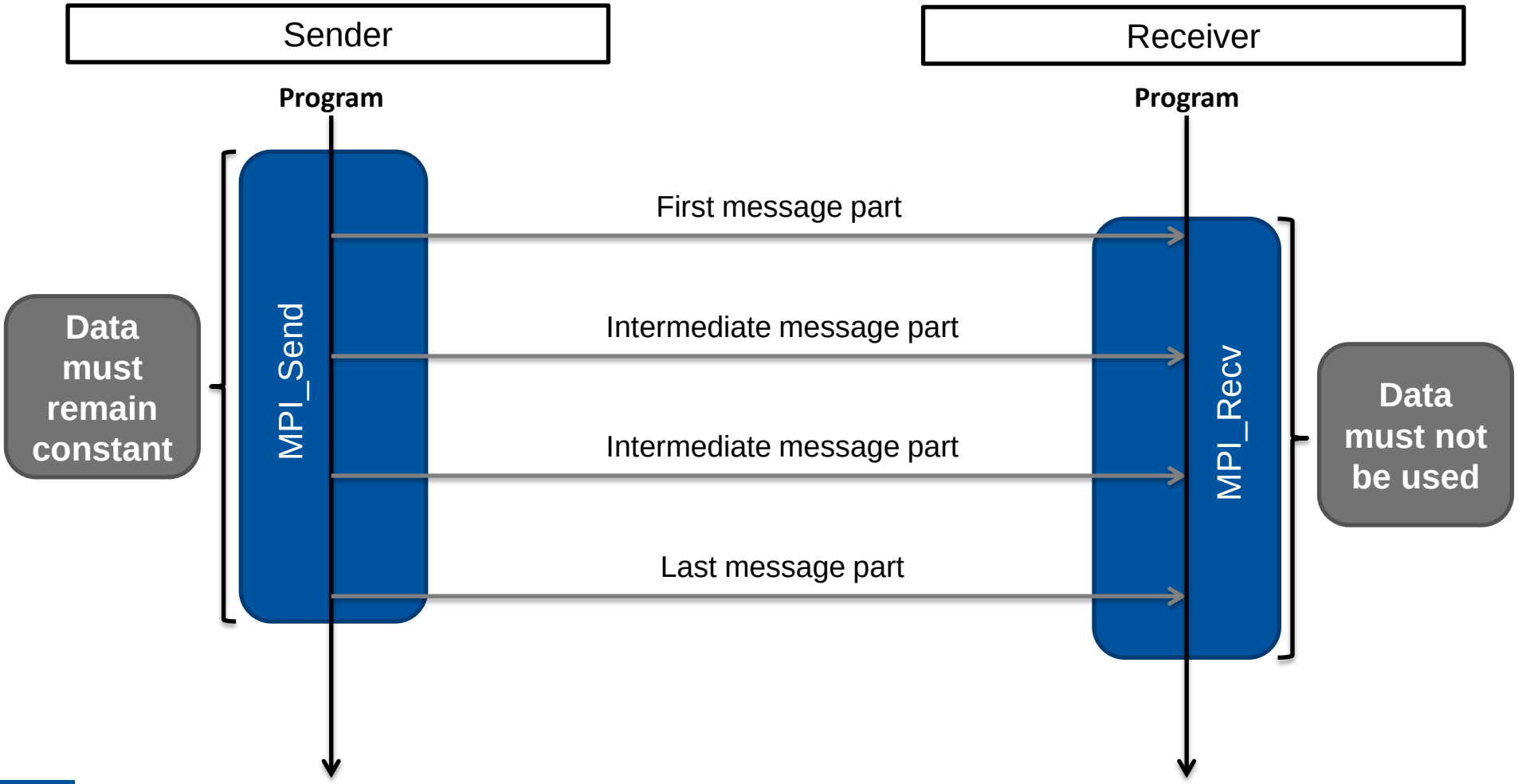
```
MPI_Wait (MPI_Request *request, MPI_Status *status)
```

→ **request:** handle for an active asynchronous operation
freed and set to **MPI_REQUEST_NULL** on successful return

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

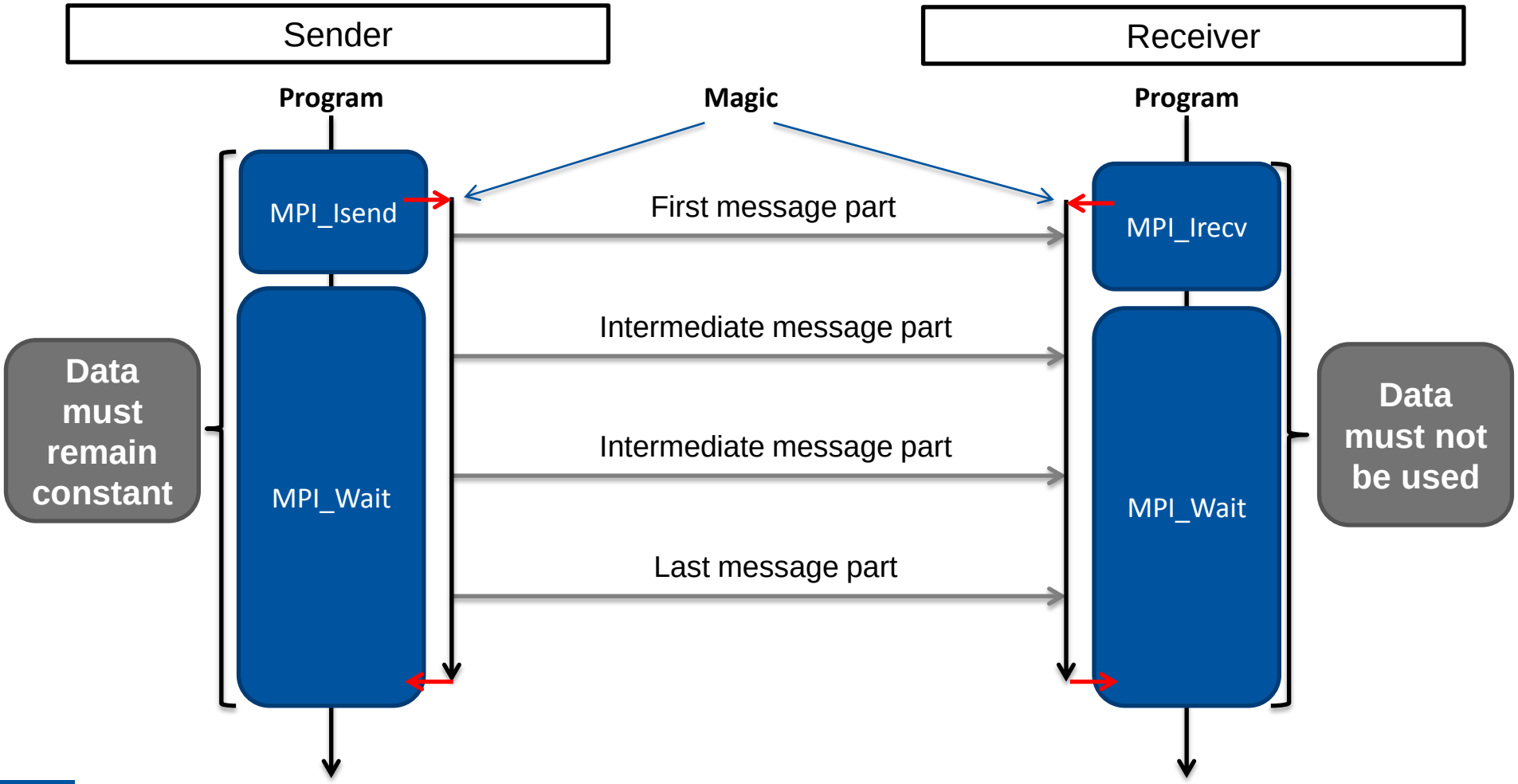
Communication-Computation Overlay

- **Blocking send (w/o buffering) and receive operate as follows:**



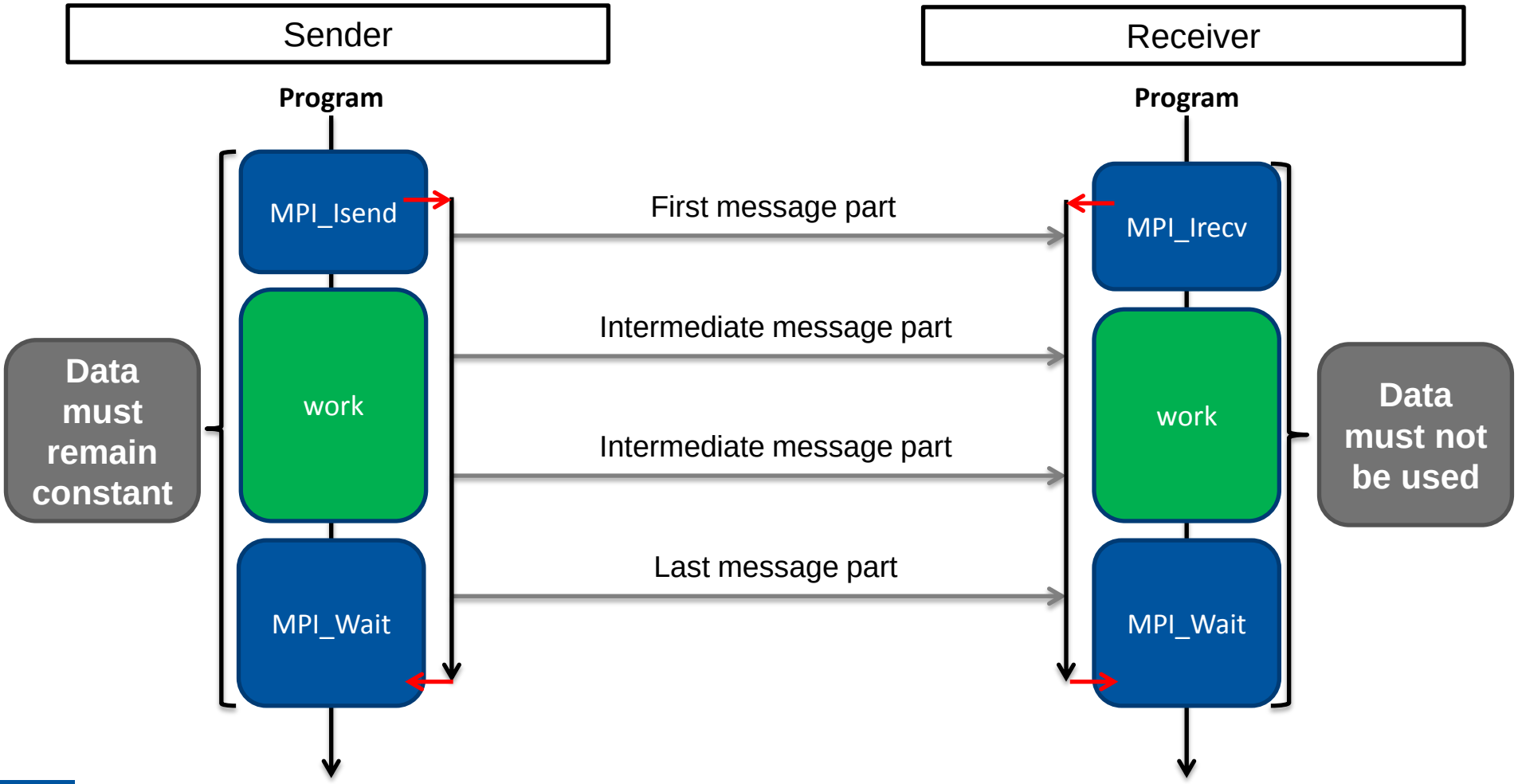
Communication-Computation Overlay

- **Equivalent with non-blocking send and receive operations:**



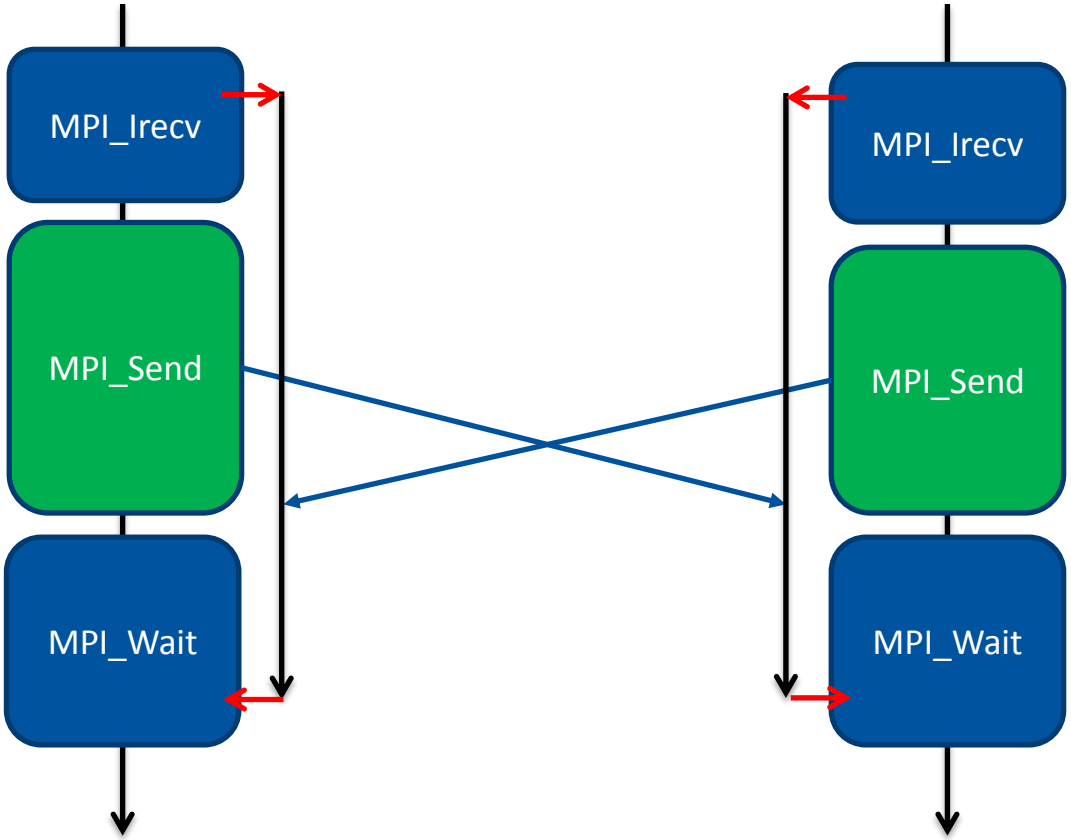
Communication-Computation Overlay

- **Other work can be done in between***



Deadlock Prevention

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



- **That is how MPI_Sendrecv is usually implemented**

Non-Blocking Request Testing

- **Test if given operation has completed:**

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

→ **flag:** **true** once the operation has completed, otherwise **false**

→ **status:** status of the operation (if completed), only set if **flag** is **true**

→ Can be (and usually is) called repeatedly from 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**

Test and Wait on Many Requests

- **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.**

Communication Modes

- **There are four send modes in MPI:**
  - → Standard
  - → Buffered
  - → Synchronous
  - → Ready
- **Send modes differ in the relation between the completion of the operation and the actual message transfer**
- **Single receive mode:**
  - → Synchronous

Send Modes

- **Standard mode**
  - → The call blocks until the message has <u>either</u> been transferred <u>or</u> copied to an internal buffer for later delivery.
- **Buffered mode**
  - → The call blocks until the message has been copied to a user-supplied buffer. Actual transmission may happen at a later point.
- **Synchronous mode**
  - → The call blocks until a matching receive has been posted and the message reception has started.
- **Ready mode**
  - → The operation succeeds <u>only if a matching receive has been posted already</u>. Behaves as standard send in every other aspect.

Send Modes

- **Call names:**
  - → **MPI_Bsend** blocking buffered send
  - → **MPI_Ibsend** non-blocking buffered send
  - → **MPI_Ssend** blocking synchronous send
  - → **MPI_Issend** non-blocking synchronous 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**)

Utility Calls

- **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 real time that has elapsed since an unspecified (but fixed between 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 Lifecycle Management

- **MPI can only be <u>initialized once</u> and <u>finalized once</u> for the lifetime of the process. Multiple calls to MPI_Init or MPI_Finalize result in error.**

- **Test if MPI is already initialized:**

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

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

- **Test if MPI is already finalized:**

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

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

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

Common Pitfalls – C/C++

- **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.

Common Pitfalls – C/C++

- **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 *array)
{
  MPI_Send(array, MPI_INT, 5, …
  … or …
  MPI_Send(&array[0], MPI_INT, 5, …
```

MPI Basics: Common Pitfalls – C/C++

- **Use flat multidimensional arrays; arrays of pointers don't 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

MPI Basics: Common Pitfalls – C/C++

- **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**

MPI Basics: Common Pitfalls – Fortran 90

- **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 ISEND. 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**

?

Agenda

- Motivation
- Part 1
  - → Concepts
  - → Point-to-point communication
  - → Non-blocking operations
- **Part 2**
  - → Collective operations
  - → Communicators
  - → User datatypes
- Part 3
  - → Hybrid parallelization
  - → Common parallel patterns

Collective Operations

- **MPI collective operations involve all ranks in a given communication context (communicator) at the same time.**

- **All ranks in the communicator must make the same MPI call for the operation to succeed. There should be one call per MPI rank (e.g. not per thread).**

- **Some collective operations are globally synchronous**

  → The MPI standard allows for early return in some ranks

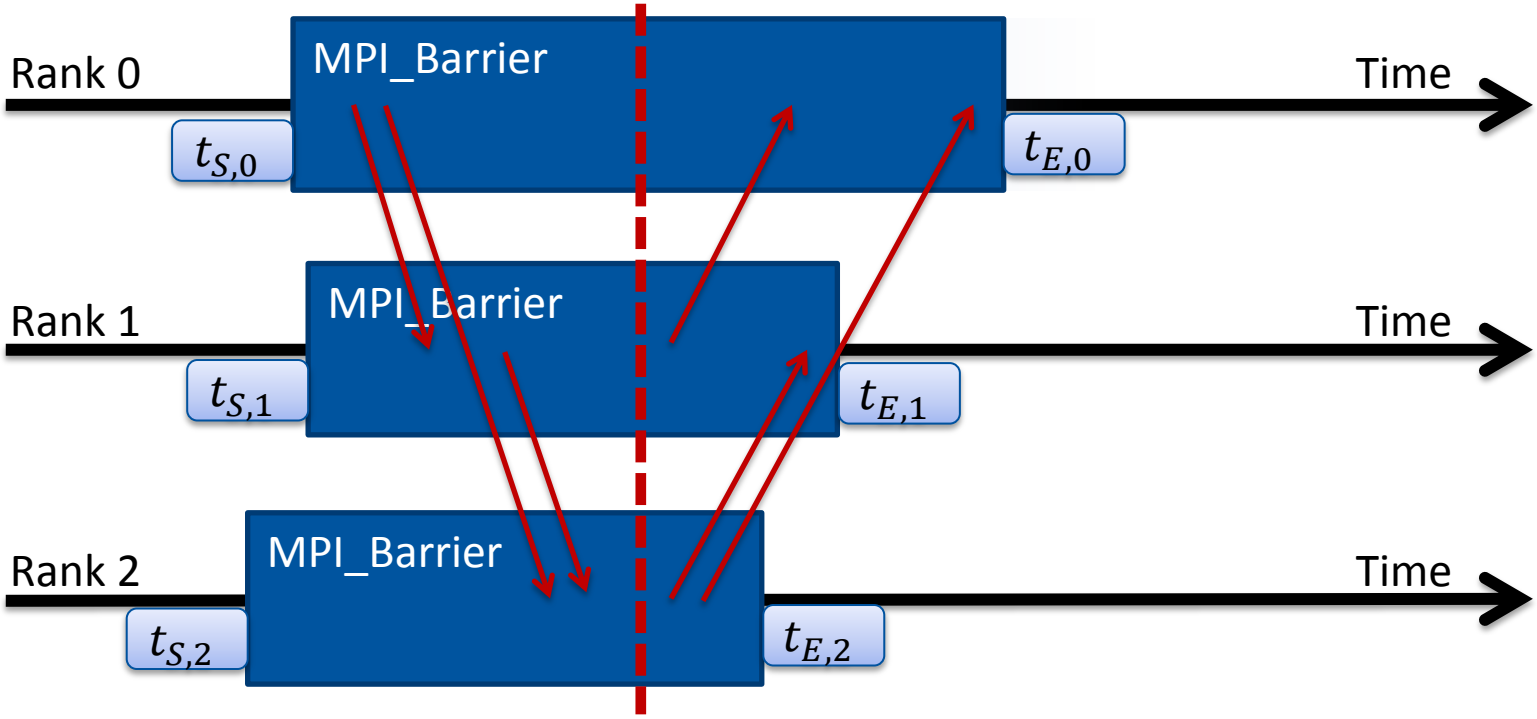
- **Collective operations are provided as convenience to the end user. They can be (and they often are) implemented with basic point-to-point communication primitives.**

  → But they are usually tuned to deliver the best system performance.

Barrier Synchronization

- **The only explicit synchronization primitive in MPI**

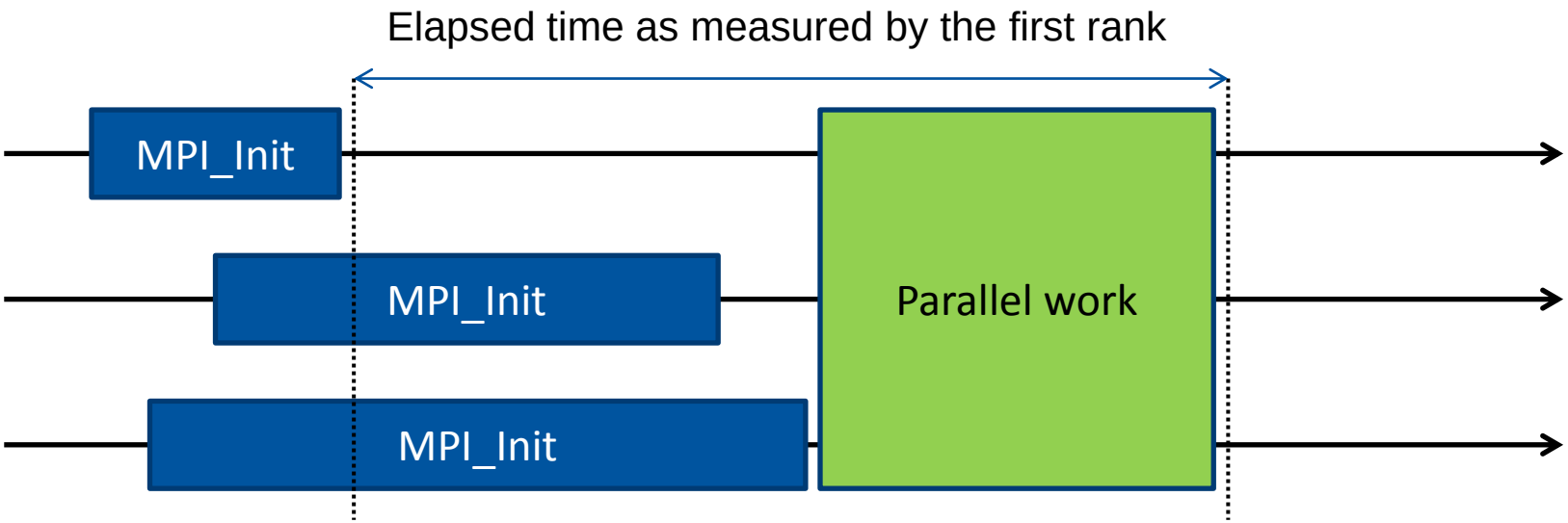
```
MPI_Barrier (MPI_Comm comm)
```



$$\max(t_{S,0}; t_{S,1}; t_{S,2}) < \min(t_{E,0}; t_{E,1}; t_{E,2})$$

Barrier synchronisation

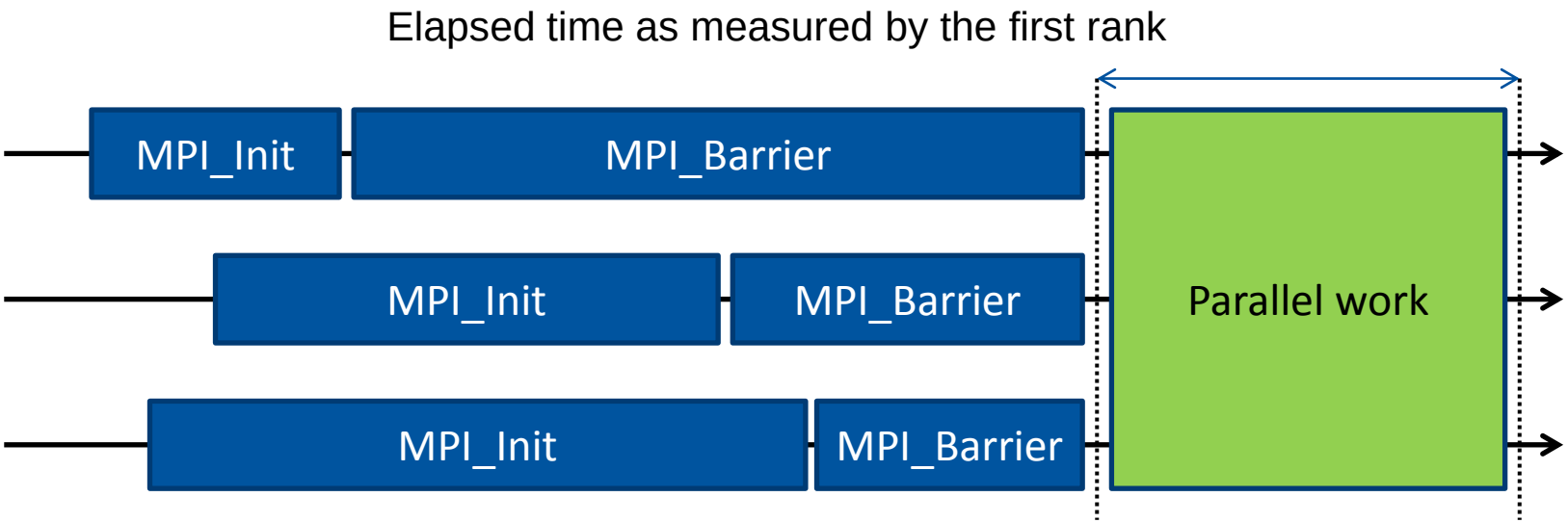
- **Useful when benchmarking**
  - → Always synchronise before taking time measurements



  - → Huge discrepancy between the actual work time and the measurement

Barrier synchronisation

- **Useful when benchmarking**
  - → Always synchronise before taking time measurements



  - → Dispersion of the barrier exit times is usually quite low

Data Replication – Broadcast

- **Broadcast from one rank (root) to all other ranks:**

```
void broadcast (void *data, int count, MPI_Type dtype,
                int root, MPI_Comm comm)
{
  int rank, numProcs, i;

  MPI_Comm_rank(comm, &rank);
  MPI_Comm_size(comm, &numProcs);
  if (rank == root) {
    for (i = 0; i < numProcs; i++)
      if (i != root)
        MPI_Send(data, count, dtype, i, 0, comm);
  }
  else
    MPI_Recv(data, count, dtype, root, 0, comm,
             MPI_STATUS_IGNORE);
}
```

Data Replication – Broadcast

- **MPI broadcast operation:**

```
MPI_Bcast (void *data, int count, MPI_Datatype dtype,
           int root, MPI_Comm comm)
```

→ **data:** data to be sent at **root**; place to put the data in all other ranks
→ **count:** number of data elements
→ **dtype:** elements' datatype
→ **root:** source rank; all ranks must specify the same value
→ **comm:** communication context

- **Notes:**

→ in all ranks but **root**, **data** is an output argument

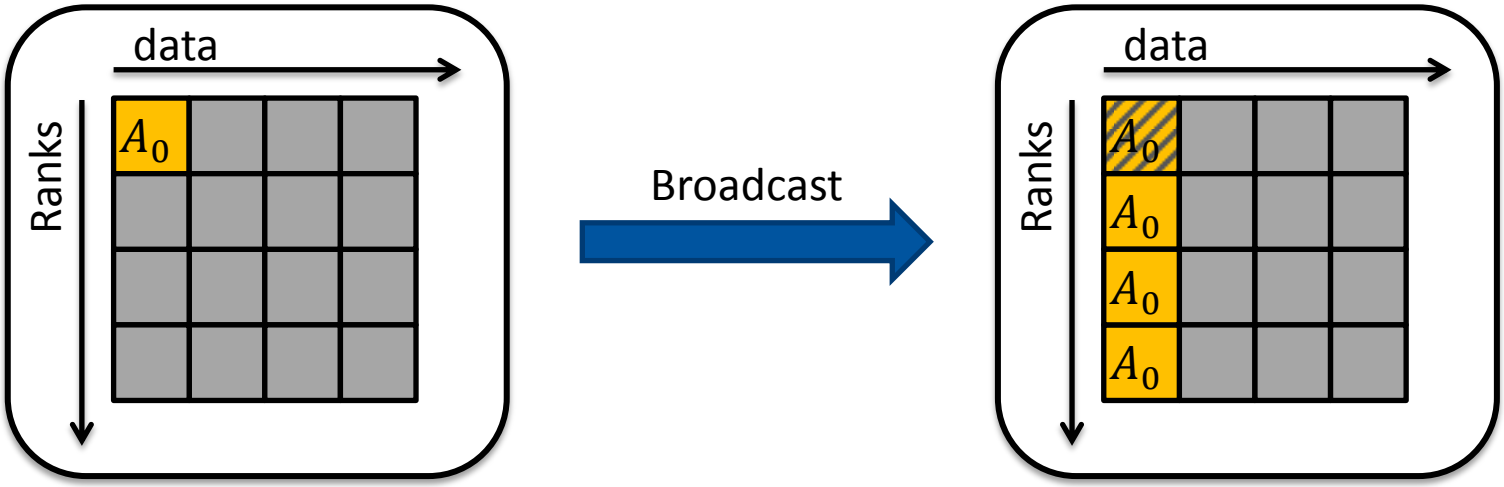
→ in rank **root**, **data** is an input argument

→ **MPI_Bcast** completes only after all ranks in **comm** have made the call

Data Replication – Broadcast

- **MPI broadcast operation:**

```
MPI_Bcast (void *data, int count, MPI_Datatype dtype,
           int root, MPI_Comm comm)
```



Data Replication – Broadcast

- **Replicate data from process root to all other processes in comm**

```
MPI_Bcast (void *data, int count, MPI_Datatype dtype,
           int root, MPI_Comm comm)
```

→ **data** points to the data source in process with rank **root**

→ **data** points to the destination buffer in all other processes

→ example use:

```
int ival;

if (rank == 0)
   ival = read_int_from_user();

MPI_Bcast(&ival, 1, MPI_INT, 0, MPI_COMM_WORLD);
```

Data Scatter

- **Distribute equally sized chunks of data from one rank to all ranks:**

```
MPI_Scatter (void *sendbuf, int sendcount, MPI_Datatype sendtype,
        void *recvbuf, int recvcount, MPI_Datatype recvtype,
        int root, MPI_Comm comm)
```

Significant at root rank only

- → **sendbuf:** data to be distributed
- → **sendcount:** size of each chunk in data elements
- → **sendtype:** source datatype
- → **recvbuf:** buffer for data reception
- → **recvcount:** number of elements to receive
- → **recvtype:** receive datatype
- → **root:** source rank
- → **comm:** communication context

Data Scatter

- **Distribute equally sized chunks of data from one rank to all ranks:**

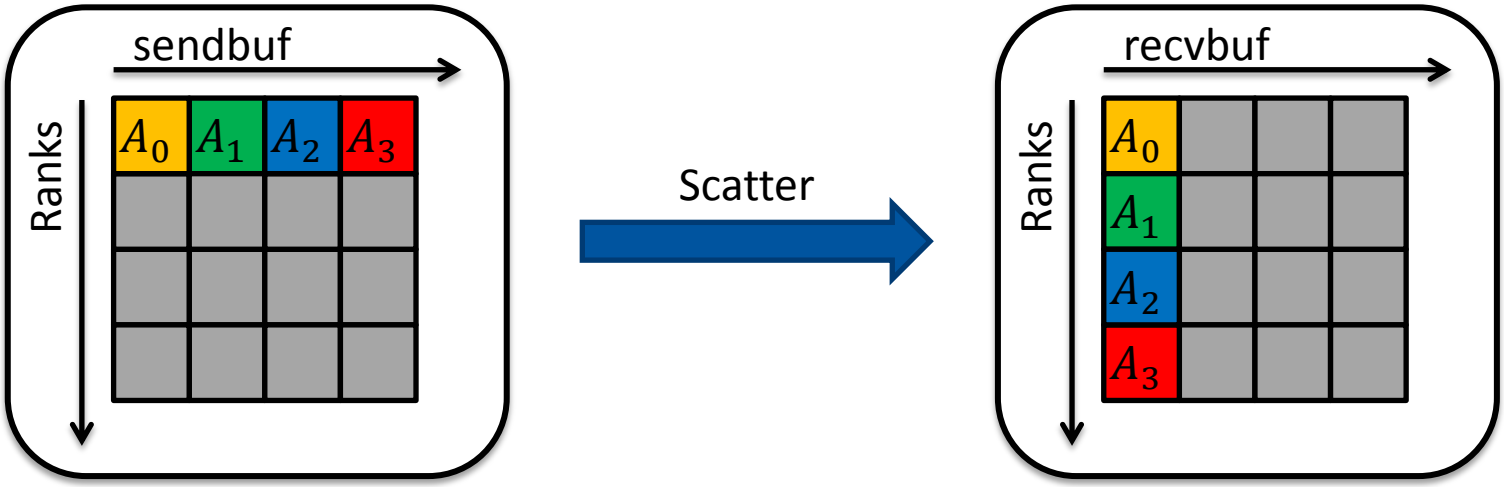
```
MPI_Scatter (void *sendbuf, int sendcount, MPI_Datatype sendtype,
       void *recvbuf, int recvcount, MPI_Datatype recvtype,
       int root, MPI_Comm comm)
```

- **Notes:**
  - → **sendbuf** must be large enough in order to supply **sendcount** elements of data to each rank in the communicator
  - → data chunks are taken in increasing order of receiver's rank
  - → root also sends one data chunk to itself
  - → for each chunk the amount of data sent must match the receive size, i.e. if **sendtype == recvtype** holds, then **sendcount == recvcount** must hold too

Data Scatter

- **Distribute equally sized chunks of data from one rank to all ranks:**

```
MPI_Scatter (void *sendbuf, int sendcount, MPI_Datatype sendtype,
        void *recvbuf, int recvcount, MPI_Datatype recvtype,
        int root, MPI_Comm comm)
```



Data Scatter

- **Distribute equally sized chunks of data from one rank to all ranks:**

```
MPI_Scatter (void *sendbuf, int sendcount, MPI_Datatype sendtype,
       void *recvbuf, int recvcount, MPI_Datatype recvtype,
       int root, MPI_Comm comm)
```

→ **sendbuf** is only accessed in the root rank

→ **recvbuf** is written into in all ranks

→ example use:

```
// Assume there are 10 MPI ranks
int bigdata[100];
int localdata[10];

MPI_Scatter(bigdata, 10, MPI_INT,
            localdata, 10, MPI_INT,
            0, MPI_COMM_WORLD);
```

Data Gather

- **Collect chunks of data from all ranks in one place:**

```
MPI_Gather (void *sendbuf, int sendcount, MPI_Datatype sendtype,
        void *recvbuf, int recvcount, MPI_Datatype recvtype,
        int root, MPI_Comm comm)
```

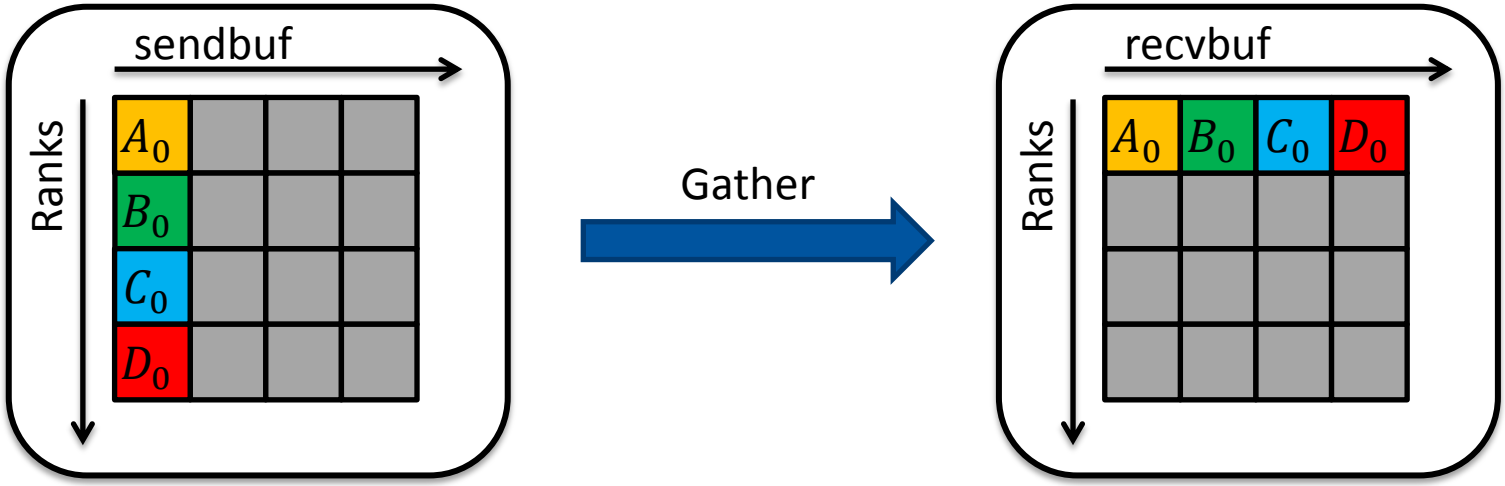
Significant at root rank only

- **The opposite operation of MPI_Scatter:**
  - → **recvbuf** must be large enough to hold **recvcount** elements from each rank
  - → root also receives one data chunk from itself
  - → data chunks are stored in increasing order of receiver's rank
  - → for each chunk the receive size must match the amount of data sent

Data Gather

- **Collect chunks of data from all ranks in one place:**

```
MPI_Gather (void *sendbuf, int sendcount, MPI_Datatype sendtype,
        void *recvbuf, int recvcount, MPI_Datatype recvtype,
        int root, MPI_Comm comm)
```



Gather-to-All

- **Collect chunks of data from all ranks in all ranks:**

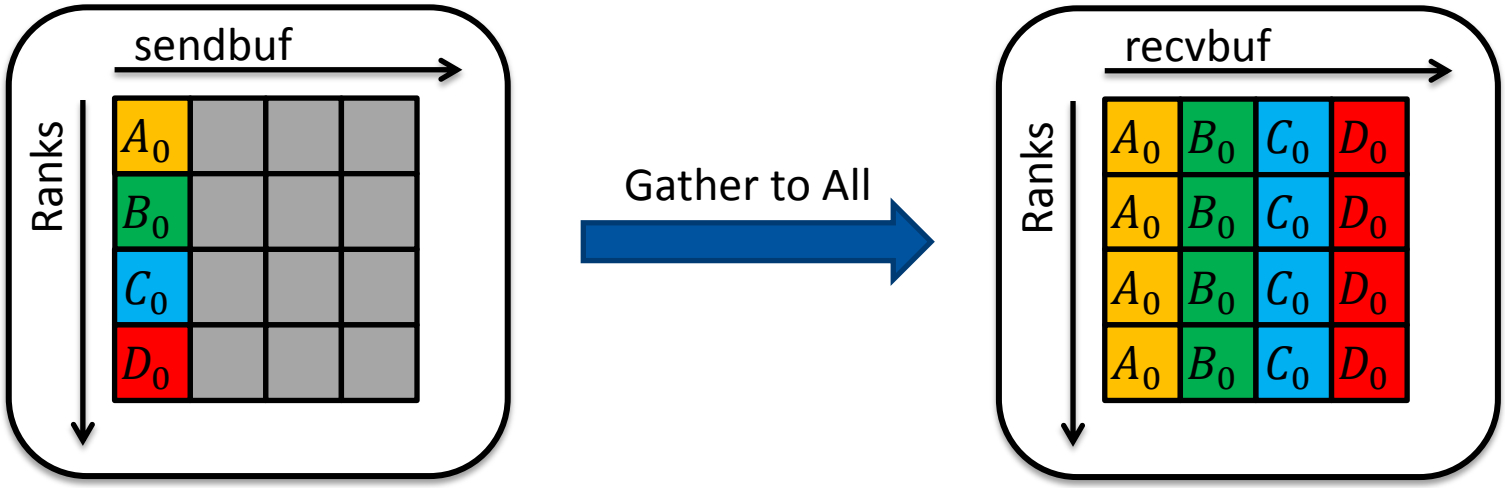
```
MPI_Allgather (void *sendbuf, int sendcount, MPI_Datatype sendtype,
       void *recvbuf, int recvcount, MPI_Datatype recvtype, MPI_Comm comm)
```

- **Note:**
  - → rootless operation – all ranks receive a copy of the gathered data
  - → each rank also receives one data chunk from itself
  - → data chunks are stored in increasing order of sender's rank
  - → for each chunk the receive size must match the amount of data sent
  - → equivalent to **MPI_Gather + MPI_Bcast**, but possibly more efficient

Gather-to-All

- **Collect chunks of data from all ranks in all ranks:**

```
MPI_Allgather (void *sendbuf, int sendcount, MPI_Datatype sendtype,
       void *recvbuf, int recvcount, MPI_Datatype recvtype, MPI_Comm comm)
```



All-to-All

- **Combined scatter and gather operation:**

```
MPI_Alltoall (void *sendbuf, int sendcount, MPI_Datatype sendtype,
      void *recvbuf, int recvcount, MPI_Datatype recvtype, MPI_Comm comm)
```

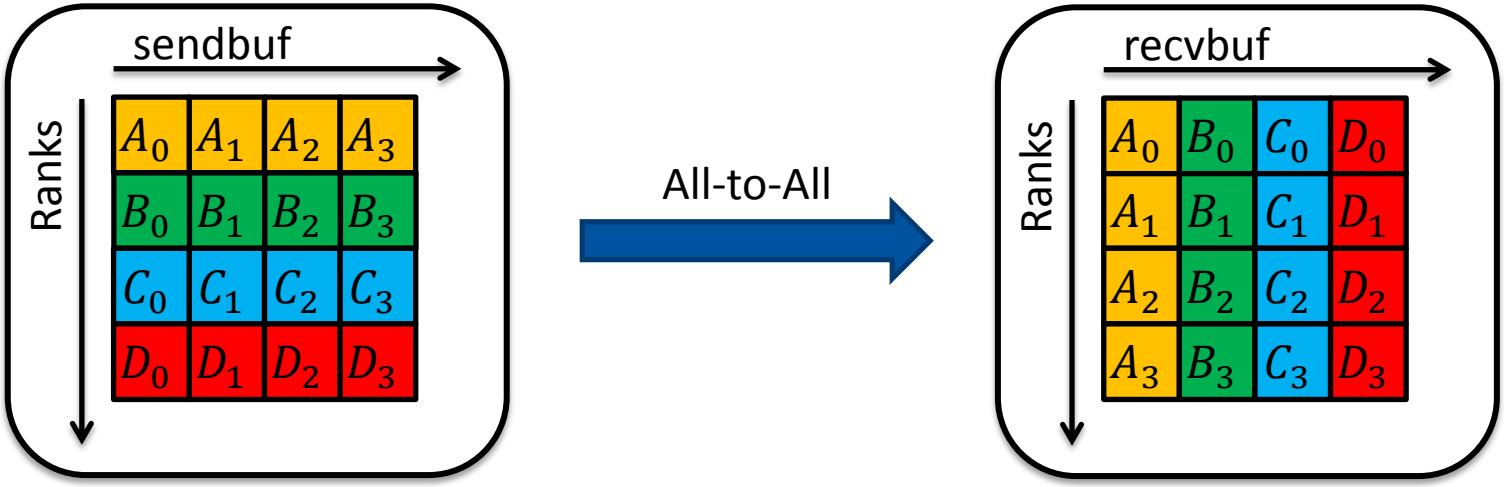
- **Notes:**
  - → rootless operation – each rank distributes its sendbuf to every rank in the communicator (including itself)
  - → data chunks are taken in increasing order of receiver's rank
  - → data chunks are stored in increasing order of sender's rank
  - → almost equivalent to **MPI_Scatter + MPI_Gather**
    (one cannot mix data from separate collective operations)

All-to-All

- **Combined scatter and gather operation:**

```
MPI_Alltoall (void *sendbuf, int sendcount, MPI_Datatype sendtype,
      void *recvbuf, int recvcount, MPI_Datatype recvtype, MPI_Comm comm)
```

- **Note: a kind of global chunked transpose**



Caveats

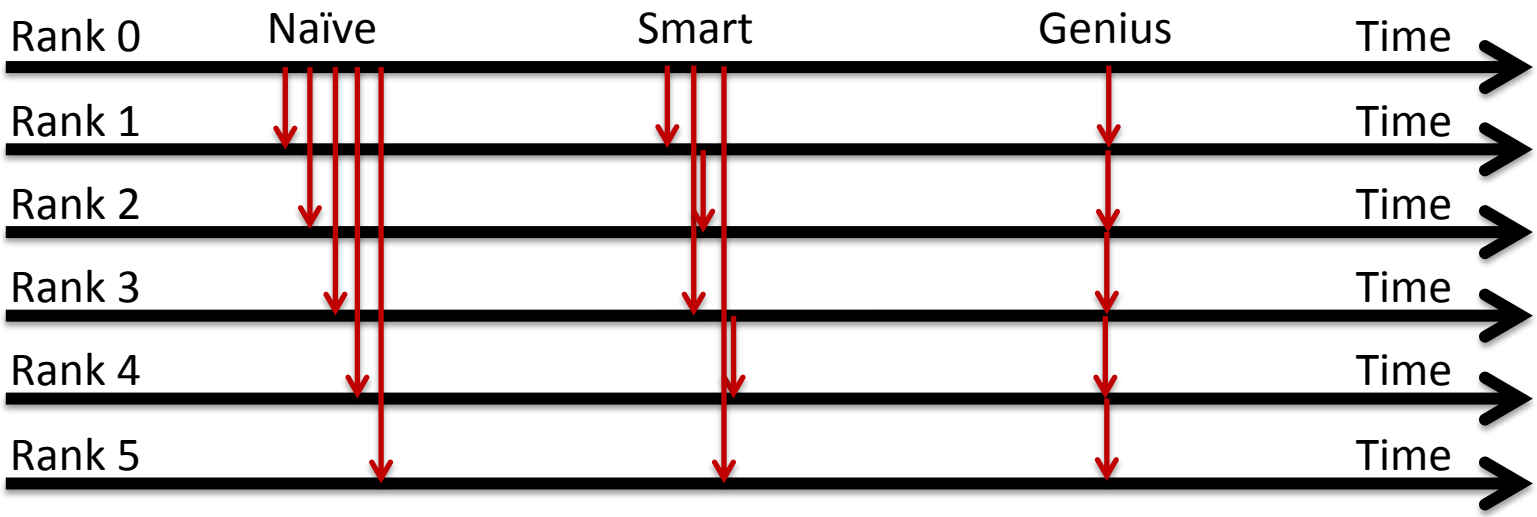
- **All ranks in the communicator must call the MPI collective operation for it to complete successfully:**
  - → both data sources (root) and data receivers have to make the same call
  - → observe the significance of each argument

- **Multiple collective operations have to be called in the same sequence by all ranks**

- **One cannot use MPI_Recv to receive data sent by MPI_Scatter**
- **One cannot use MPI_Send to send data to MPI_Gather**

Advantages

- **Collective operations implement portably common SPMD patterns**
- **Implementation-specific magic, but standard behavior**
- **Example: Broadcast**
  - → Naïve: root sends separate message to every other rank, O(#ranks)
  - → Smart: tree-based hierarchical communication, O(log(#ranks))
  - → Genius: pipelined segmented transport, O(1)



Global Reduction

- **Perform an arithmetic reduction operation while gathering data**

```
MPI_Reduce (void *sendbuf, void *recvbuf, int count,
       MPI_Datatype datatype, MPI_Op op, int root, MPI_Comm comm)
```

→ **sendbuf:** data to be reduced
→ **recvbuf:** location for the result(s) (significant only at the root)
→ **count:** number of data elements
→ **datatype:** elements' datatype
→ **op:** handle of the reduction operation
→ **root:** destination rank
→ **comm:** communicator

- **Result is computed in or out of order depending on the operation:**
  → All predefined operations are commutative
  → Beware of non-commutative effects on floats

Global reduction

- **Element-wise and cross-rank operation**

→ **rbuf[i] = sbuf$_0$[i]** op **sbuf$_1$[i]** op **sbuf$_2$[i]** op … **sbuf$_{nranks-1}$[i]**

sbuf$_0$[]	1	2	3	4	5	6	7	8	9
	⊗	⊗	⊗	⊗	⊗	⊗	⊗	⊗	⊗
sbuf$_1$[]	10	11	12	13	14	15	16	17	18
	⊗	⊗	⊗	⊗	⊗	⊗	⊗	⊗	⊗
sbuf$_2$[]	19	20	21	22	23	24	25	26	27
	⊗	⊗	⊗	⊗	⊗	⊗	⊗	⊗	⊗
sbuf$_3$[]	28	29	30	31	32	33	34	35	36
	↓	↓	↓	↓	↓	↓	↓	↓	↓
rbuf[]	58	62	66	70	74	78	82	86	90

⊗ = MPI_SUM

Global Reduction

- **Some predefined operation handles:**

MPI_Op	Result value
MPI_MAX	Maximum value
MPI_MIN	Minimum value
MPI_SUM	Sum of all values
MPI_PROD	Product of all values
MPI_LAND	Logical AND of all values
MPI_BAND	Bit-wise AND of all values
MPI_LOR	Logical OR of all values
…	…

- **Users can register their own operations, but that goes beyond the scope of these course.**

Global Reduction

- **Perform an arithmetic reduction and broadcast the result:**

```
MPI_Allreduce (void *sendbuf, void *recvbuf, int count,
               MPI_Datatype datatype, MPI_Op op, MPI_Comm comm)
```

- **Notes:**
  - → rootless operation – every rank receives the result of the reduction operation
  - → equivalent to **MPI_Reduce + MPI_Bcast** with the same root
  - → can be slower with non-commutative operations because of the forced in-order execution (the same applies to **MPI_Reduce**)

Other Collective Calls

- **Versions with variable chunk size:**
  - → **MPI_Gatherv**
  - → **MPI_Scatterv**
  - → **MPI_Allgatherv**
  - → **MPI_Alltoallv**

- **Special reduction operations:**
  - → **MPI_Scan**
  - → **MPI_Reduce_scatter**
  - → **MPI_Reduce_scatter_block**
  - → …

Collective Operations

?

Agenda

- Motivation
- Part 1
  - → Concepts
  - → Point-to-point communication
  - → Non-blocking operations
- **Part 2**
  - → Collective operations
  - → Communicators
  - → User datatypes
- Part 3
  - → Hybrid parallelization
  - → Common parallel patterns

Communication Contexts

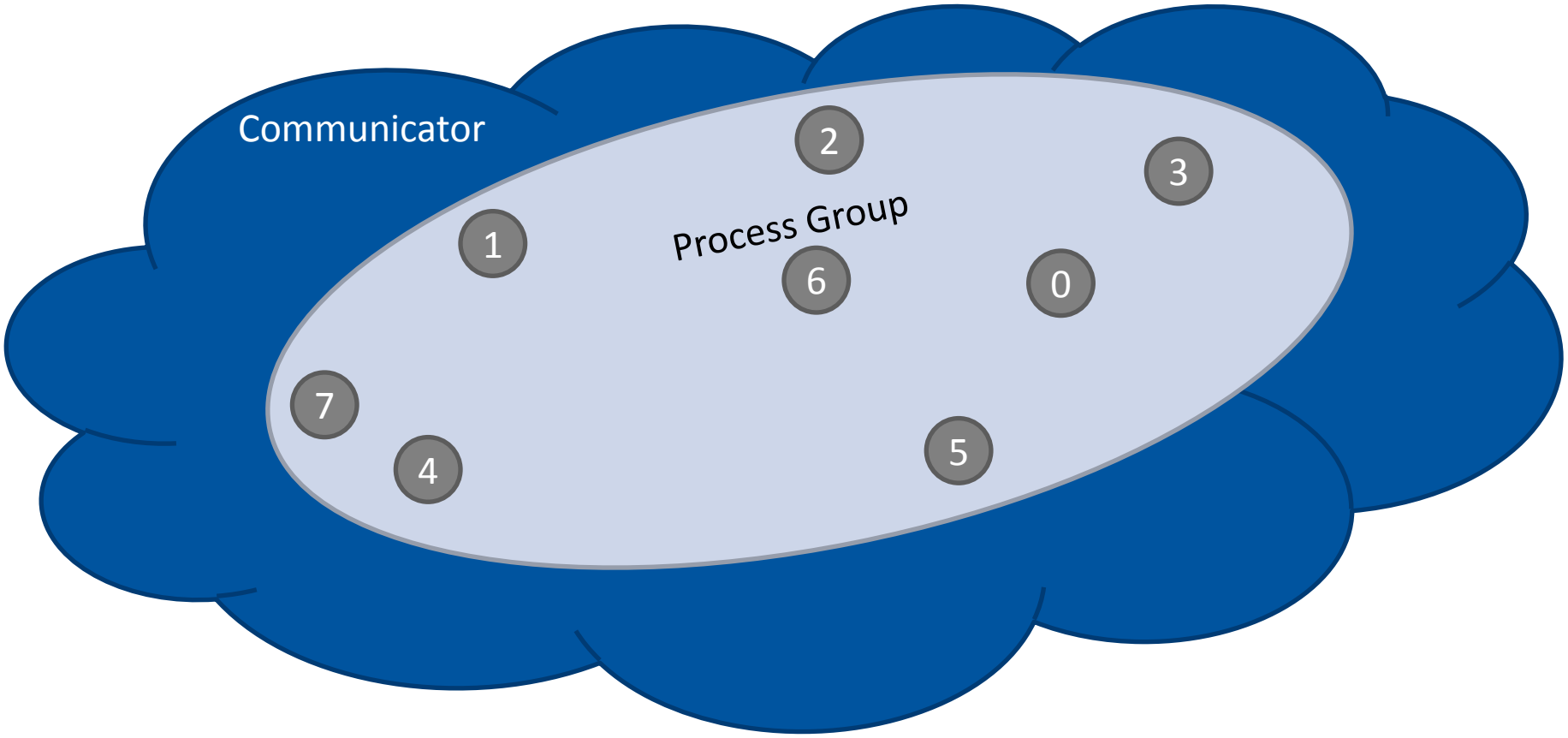
- **Each communication operation in MPI happens in a certain context:**
  - → Group of participating partners (process groups)
  - → Error handlers
  - → Key/value pairs
  - → Virtual topologies
- **MPI provides two predefined contexts:**
  - → **MPI_COMM_WORLD**

    contains all processes launched as part of the MPI program
  - → **MPI_COMM_SELF**

    contains only the current process
- **A unique communication endpoints consists of a communicator handle and a rank from that communicator.**

Communicators

- **Communicator – process group – ranks**



Query Operations

- **Obtain the size of the process group of a given communicator:**

```
MPI_Comm_size (MPI_Comm comm, int *size)
```

  → ranks in the group are numbered from 0 to size-1

- **Obtain the rank of the calling process in the given communicator:**

```
MPI_Comm_rank (MPI_Comm comm, int *rank)
```

- **Special "null" rank – MPI_PROC_NULL**
  - → member of any communicator
  - → can be sent messages – results in a no-op
  - → can be received messages from – zero-size message tagged **MPI_ANY_TAG**
  - → use it to write symmetric code and handle process boundaries

Message Envelope Matching

- **Recall: 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

- **Cross-communicator messaging is not possible**
  - → messages sent in one communicator can only be received by ranks from the same communicator
  - → communicators can be used to isolate communication to prevent interference and tag clashes – useful when writing parallel libraries
- **This course only deals with MPI_COMM_WORLD**
  - → simple flat addressing

Virtual topologies

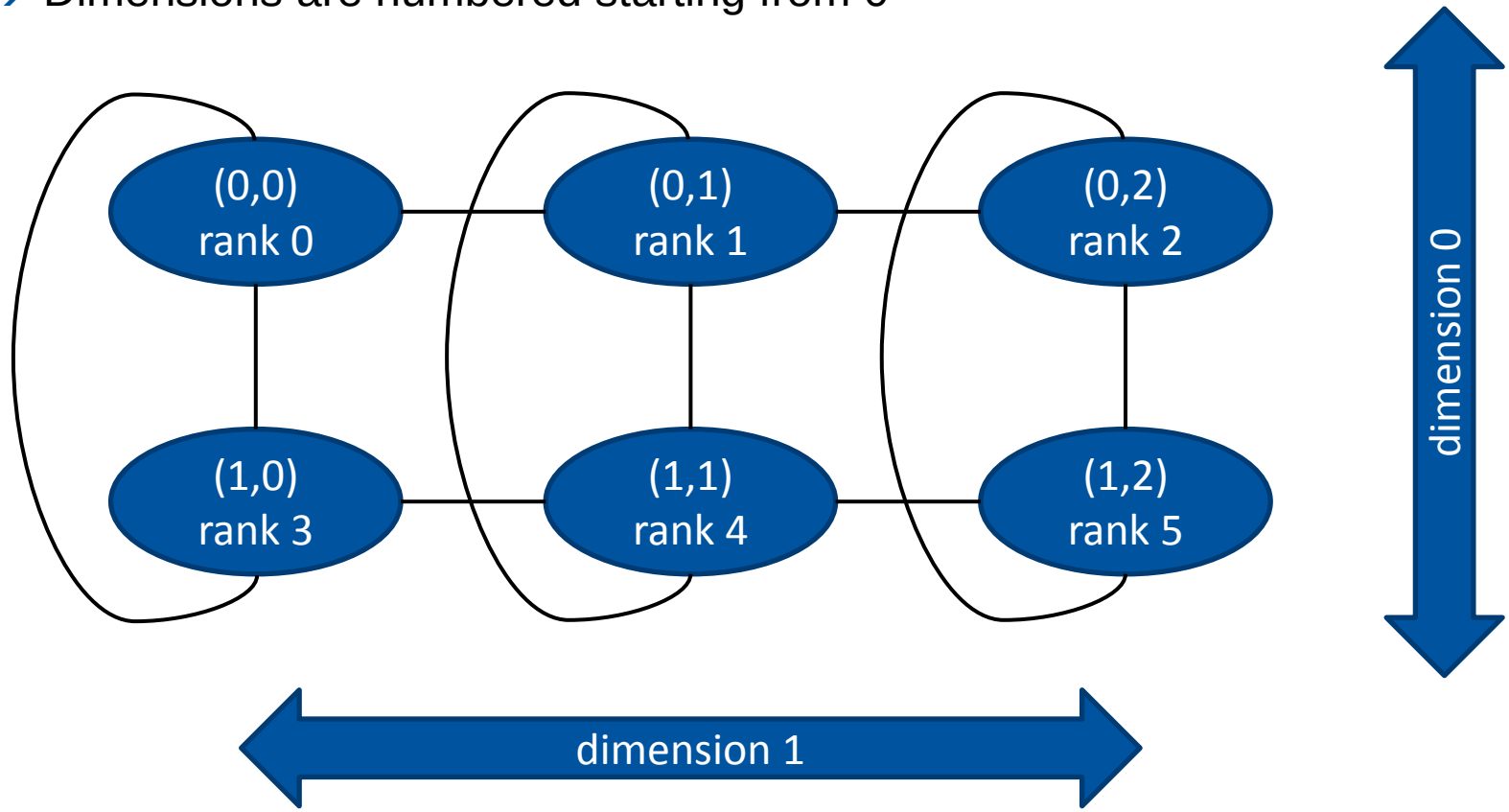
- **Each communicator can have an associated topology**
  - → Mapping between ranks and abstract addresses
  - → Connectivity (neighbouring links) information

- **Three different topology kinds:**
  - → no topology given – e.g. **MPI_COMM_WORLD**
  - → graph topology – general connectivity graph
  - → Cartesian topology – regular *n*-dimensional grid of processes

- **Missing neighbouring links in the topology do not prevent processes from communicating with each other**

Cartesian topology

- **Regular n-dimensional grid**
  - → Dimensions are numbered starting from 0



Cartesian topology

Construct a Cartesian topology

```
MPI_Cart_create (old_comm, ndims, dims, periods, reorder, comm_cart)
```

- Creates a new communicator **comm_cart** from the process group of **old_comm** with an **ndims**-dimensional Cartesian topology attached
- **dims[]** – specifies the number of processes along each dimension
- **periods[]** – specifies the periodicity in each dimension
- **reorder** – if set to .TRUE./non-zero, hints the MPI runtime to reorder the ranks in the new communicator so that their physical connectivity matches as closely as possible the virtual one; otherwise ranks are kept

Balanced Cartesian distribution

- **Create a balance distribution of a number of processes**

```
MPI_Dims_create (nnodes, ndims, dims)
```

→ Computes the most balanced way to arrange **nnodes** processes into an **ndims**-dimensional grid

→ Non-zero elements in **dims** fix the number of processes in the corresponding dimension

→ Zero elements are filled with the optimal number of processes along the corresponding dimension

→ Error if the product of non-zero elements of **dims** does not divide **nnodes**

Balanced Cartesian distribution

- **Create a balance distribution of a number of processes**

```
MPI_Dims_create (nnodes, ndims, dims)
```

→ The computed sizes are set in non-increasing order

→ the lowest-numbered dimension receives the biggest size

→ Example (taken from the MPI standard):

dims before call	function call	dims on return
(0,0)	`MPI_Dims_create(6, 2, dims)`	(3,2)
(0,0)	`MPI_Dims_create(7, 2, dims)`	(7,1)
(0,3,0)	`MPI_Dims_create(6, 3, dims)`	(2,3,1)
(0,3,0)	`MPI_Dims_create(7, 3, dims)`	erroneous call

Coordinate conversion

- **Translate Cartesian coordinate tuples into ranks**

```
MPI_Cart_rank (comm, coords, rank)
```

→ **comm** – Cartesian communicator

→ **coords** – an array of at least **ndims** elements – Cartesian coordinates

→ **rank** – corresponding process rank in **comm**

- **Translate ranks into Cartesian coordinate tuples**

```
MPI_Cart_coords (comm, rank, maxdims, coords)
```

→ **coords** – an array of **maxdims** elements to receive the coordinates

→ **maxdims** should be equal to or larger than **ndims**

Cartesian shifts

- **Find neighbouring processes' ranks**

```
MPI_Cart_shift (comm, dir, disp, source, dest)
```

→ Computes the neighbouring ranks to communicate with in order to perform data shift (using **MPI_Sendrecv**) at distance of **disp** in direction **dir**

→ Equivalent to:

→ obtain the Cartesian coordinates of the calling process

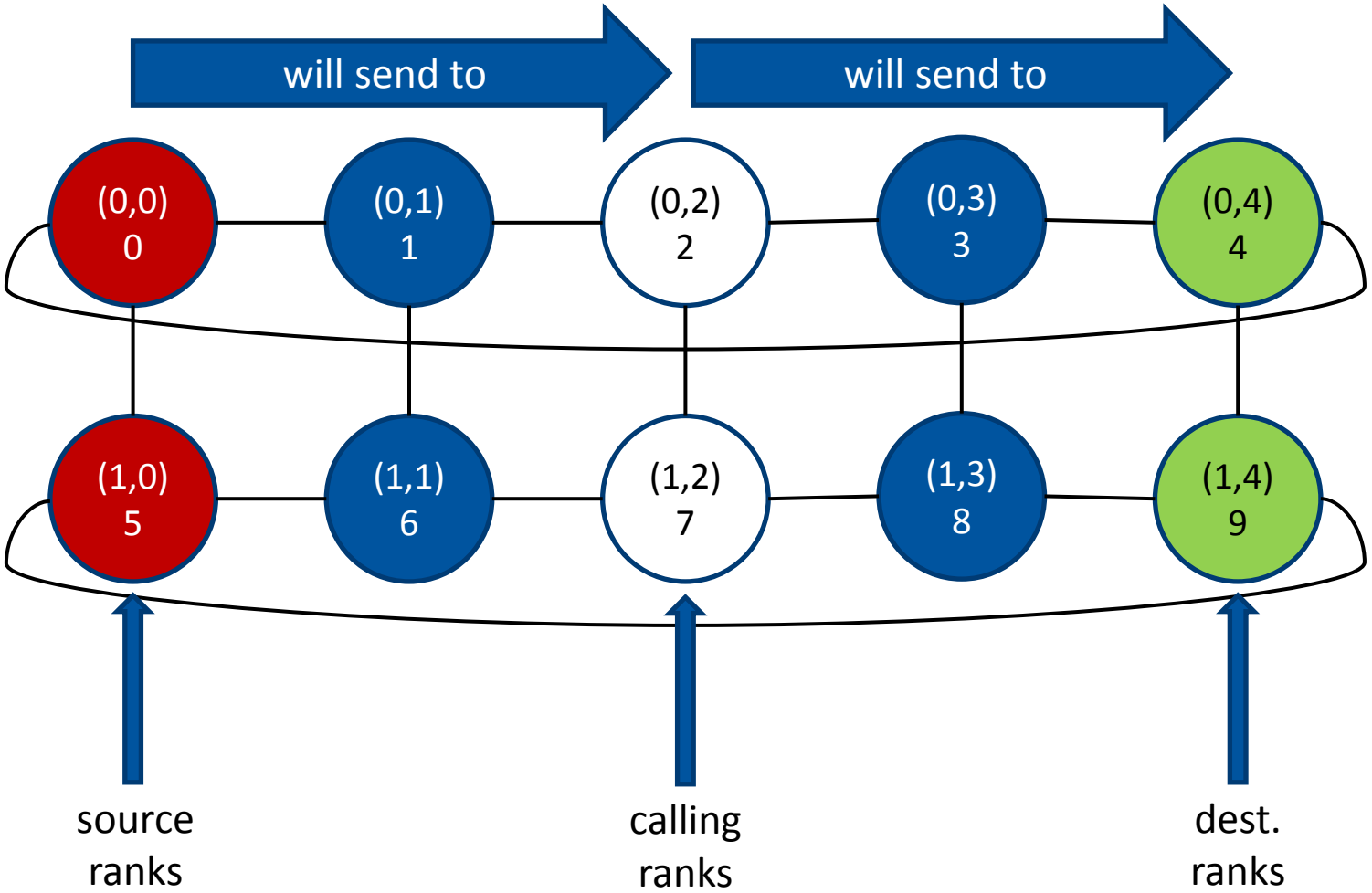
→ translate (…, $coord_{dir}$ + disp, …) into rank **dest**

→ translate (…, $coord_{dir}$ – disp, …) into rank **source**

→ If the source or the destination lies beyond a non-periodic boundary, the corresponding rank is set to **MPI_PROC_NULL**

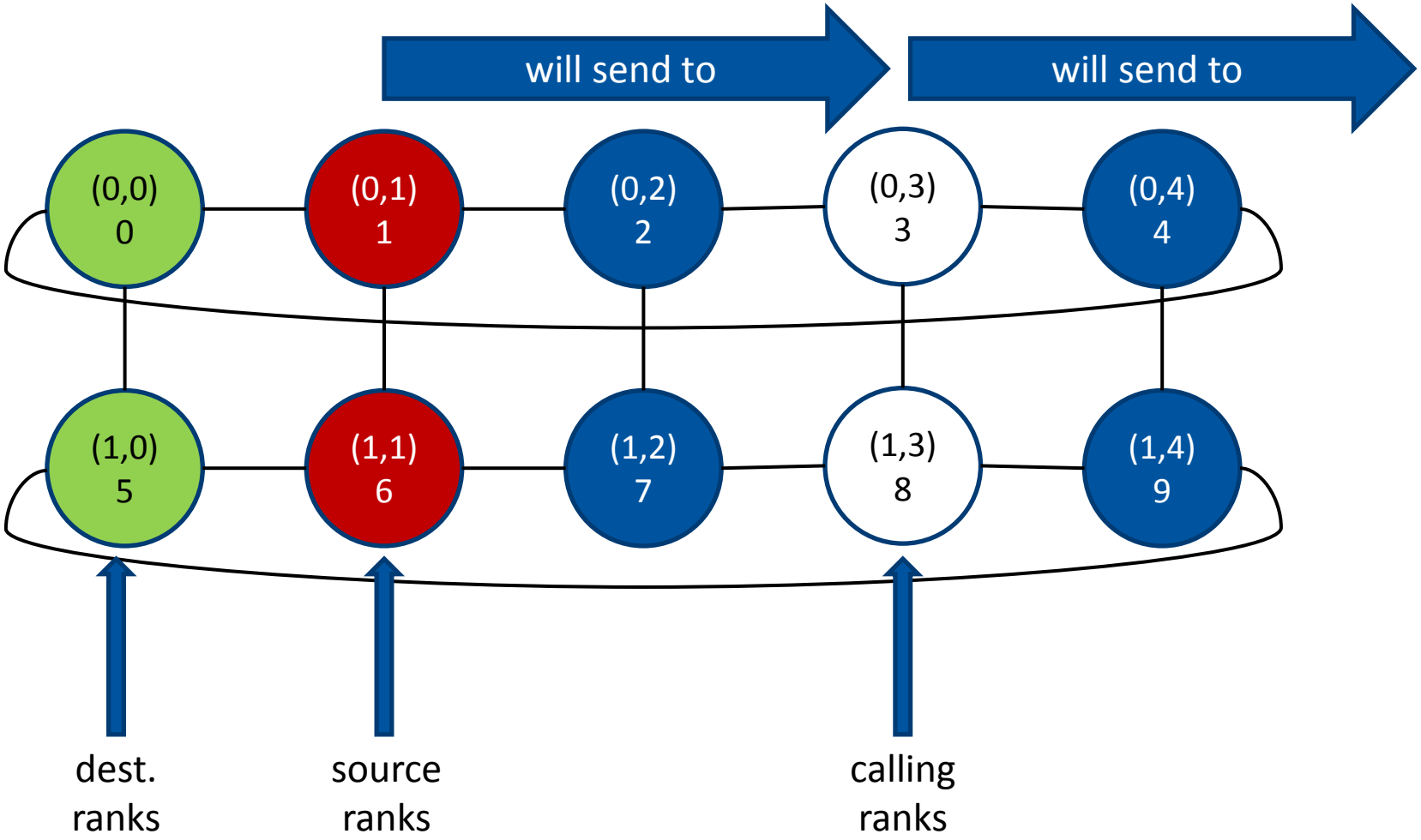
Cartesian shifts

- **Example: periodic boundary (dir = 1, disp = 2)**



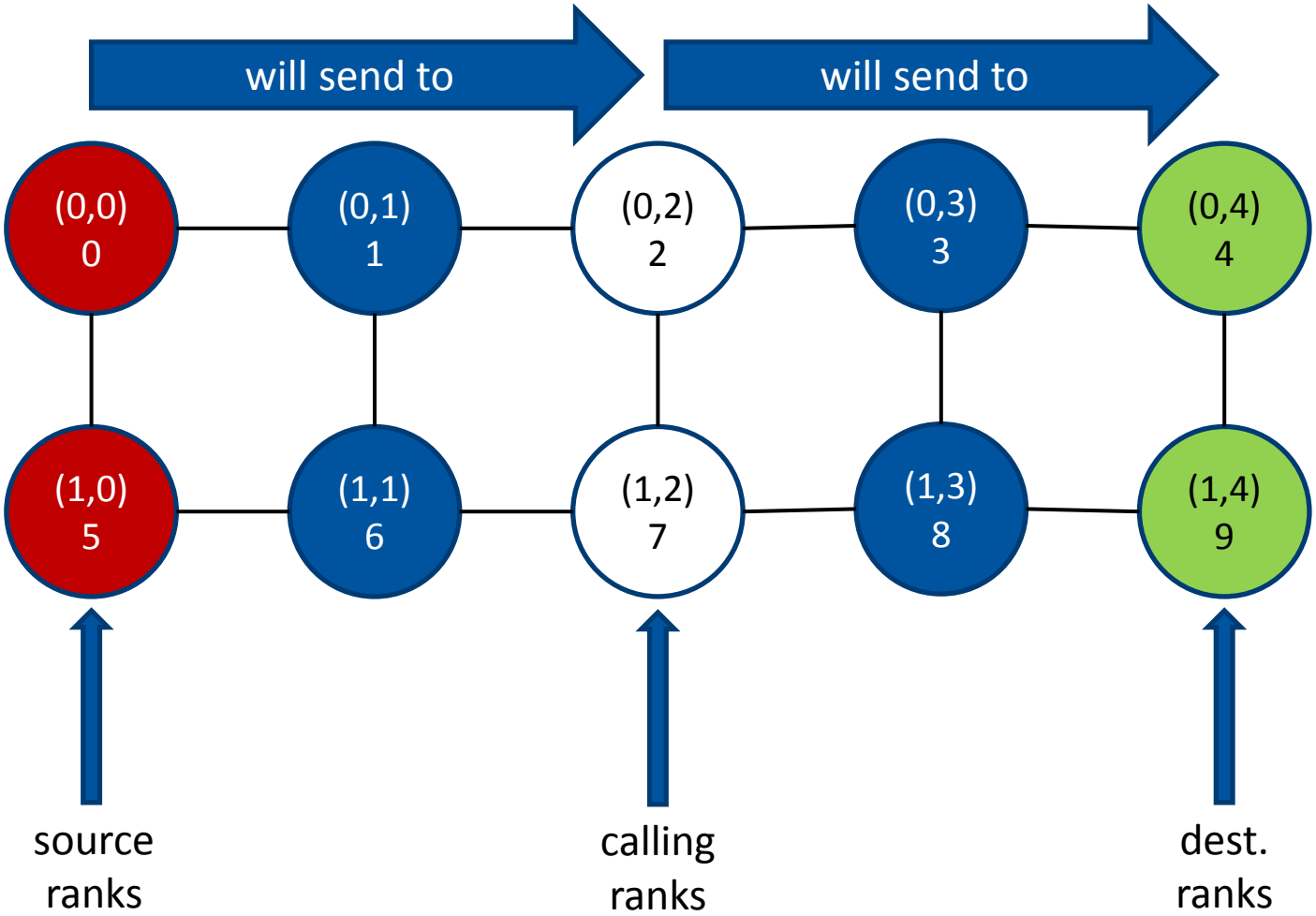
Cartesian shifts

- **Example: periodic boundary (dir = 1, disp = 2)**



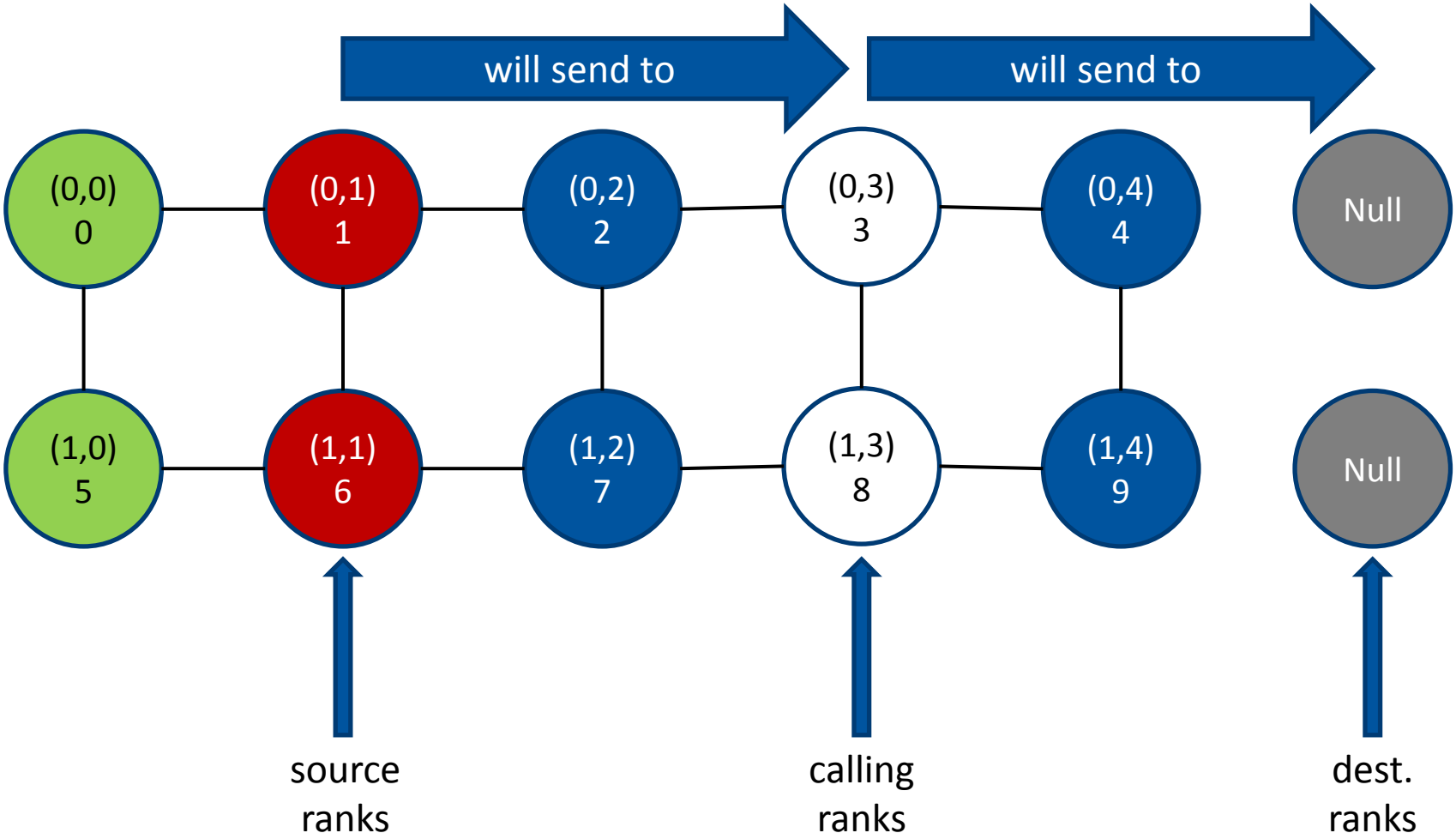
Cartesian shifts

- **Example: non-periodic boundary (dir = 1, disp = 2)**



Cartesian shifts

- **Example: non-periodic boundary (dir = 1, disp = 2)**



Cartesian partitioning

- **Split a Cartesian communicator along some dimensions**

```
MPI_Cart_sub (comm, remain_dims, newcomm)
```

→ **remain_dims** – boolean array; a true element flags particular dimension as being preserved by the operation (i.e. no splitting along that dimension)

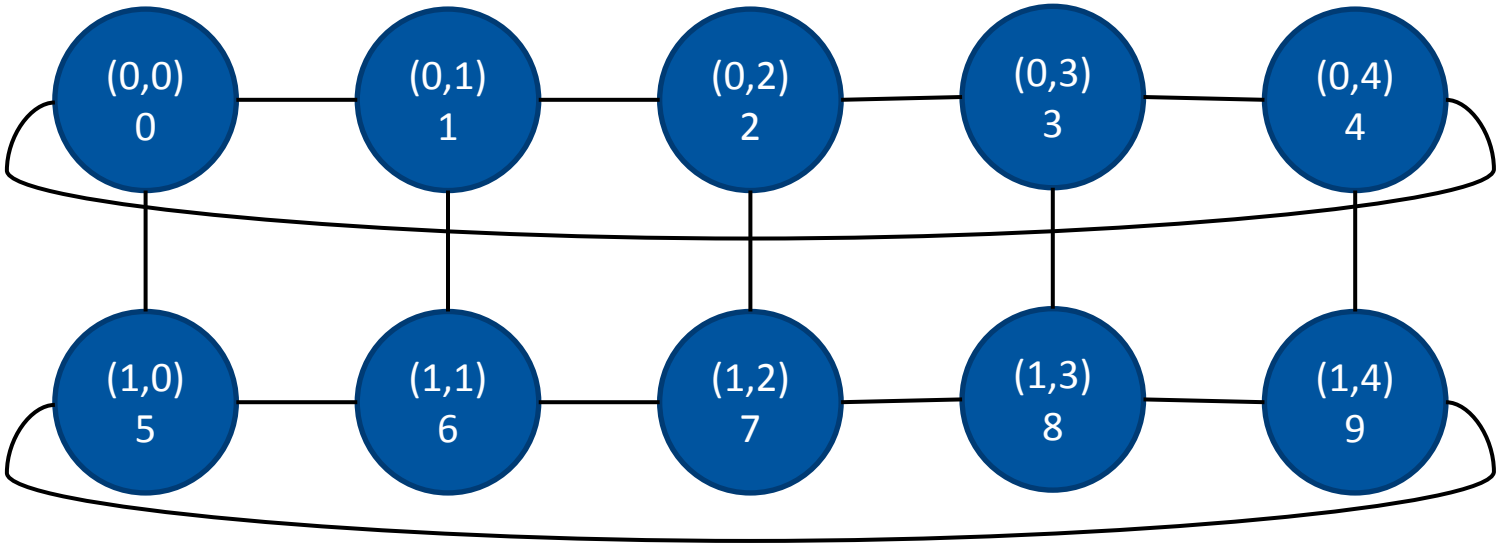
→ Creates a new Cartesian subcommunicator for each coordinate along non-preserved dimensions

→ Processes with the same coordinates along non-preserved dimensions become members of the same subcommunicator

→ Periodicity along the preserved dimensions is carried on into the newly created subcommunicators

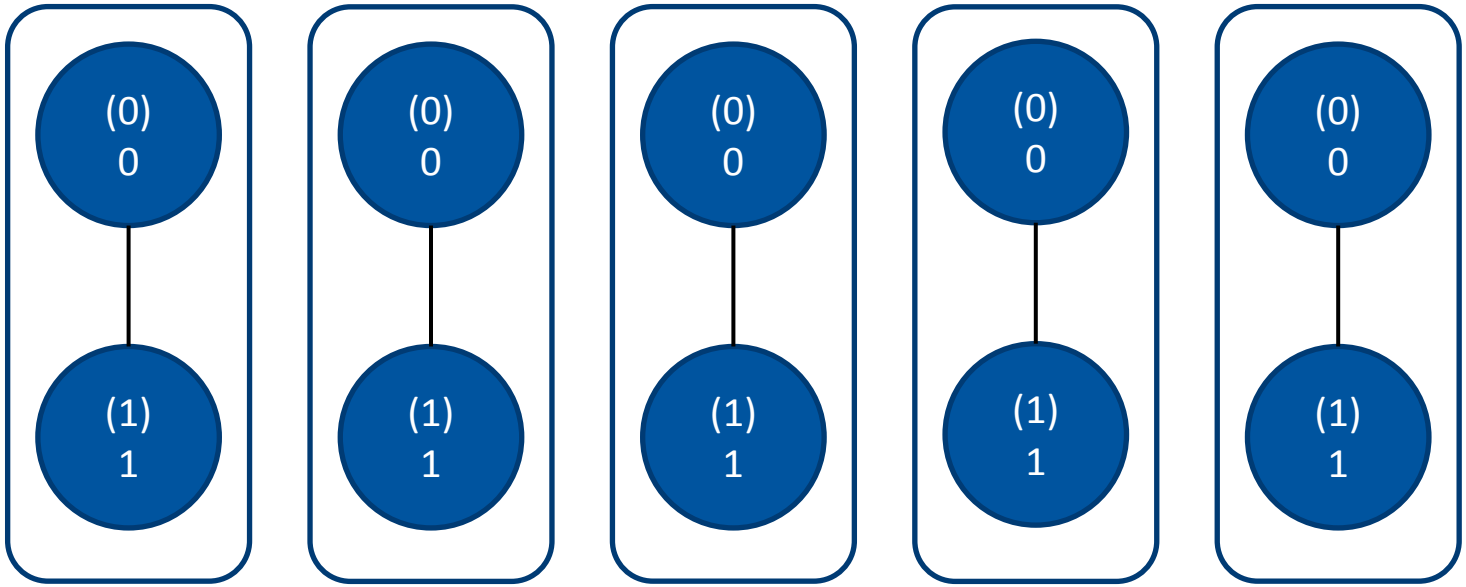
Cartesian partitioning

- **Example: initial 2x5 Cartesian topology**



Cartesian partitioning

- **Example: remain_dims = { true, false }**

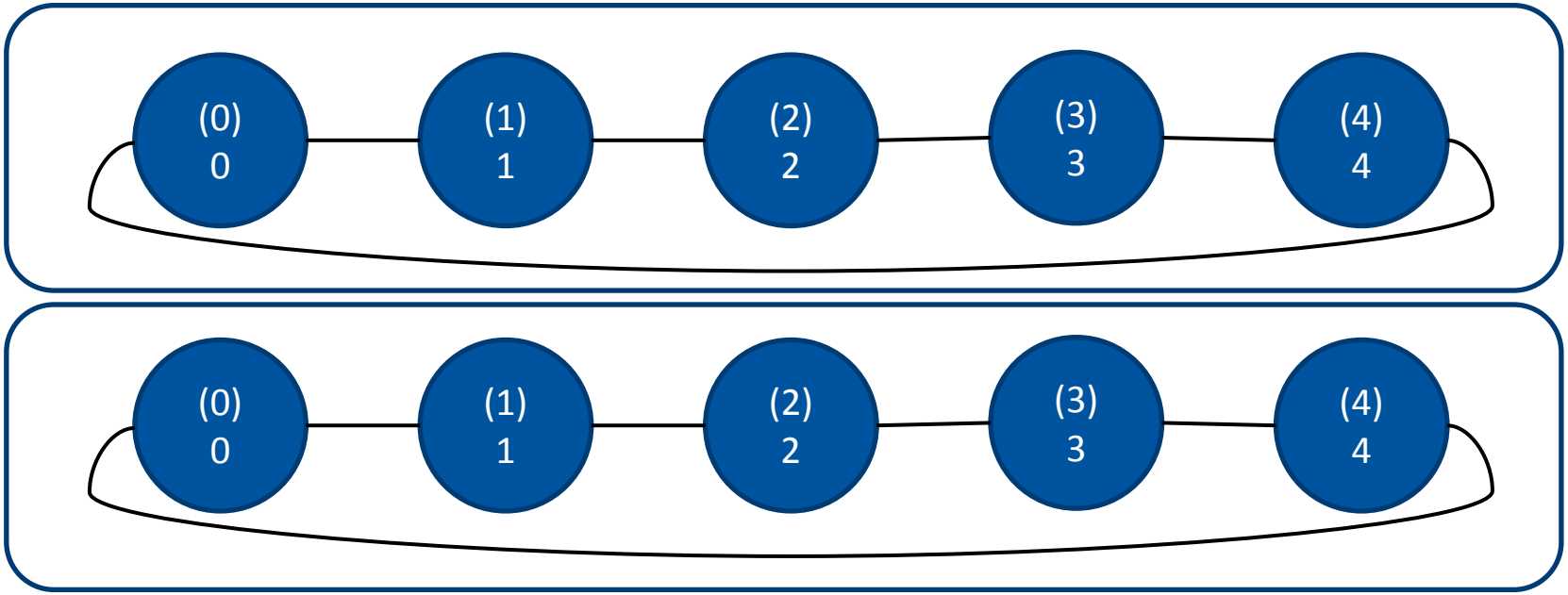


→ Five one-dimensional subcommunicators created as a result

→ Each subcommunicator contains 2 processes

Cartesian partitioning

- **Example: remain_dims = { false, true }**

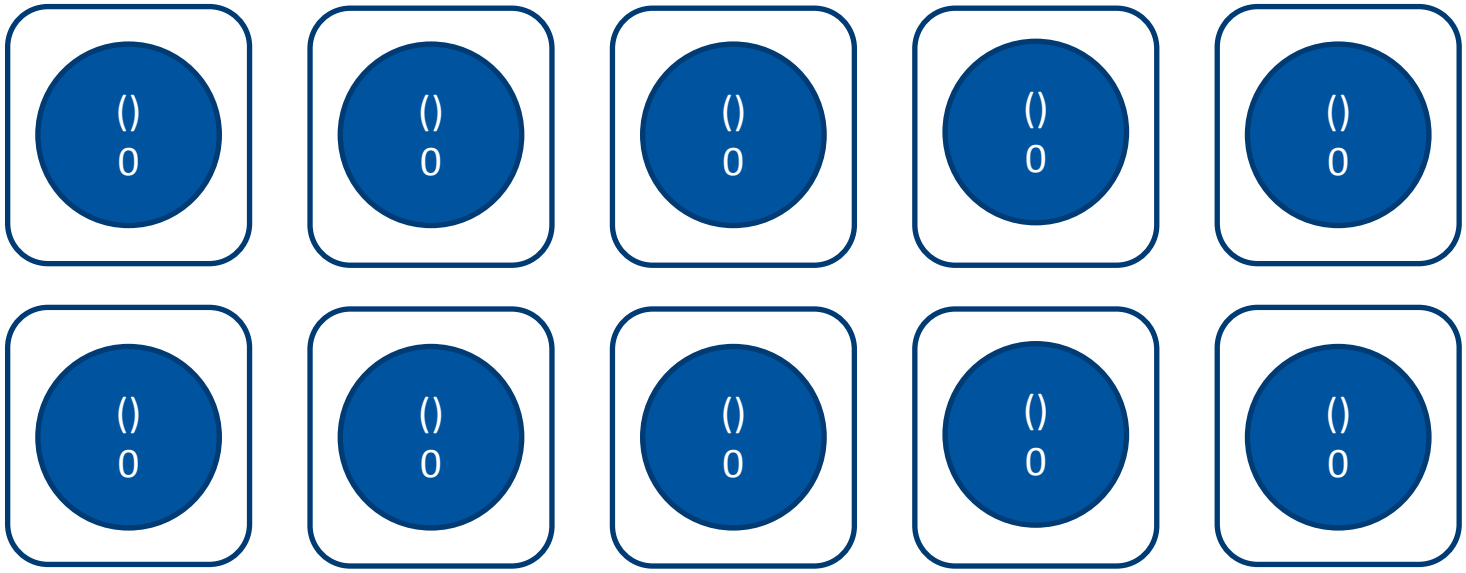


→ Two one-dimensional subcommunicators created as a result

→ Each subcommunicator contains 5 processes

Cartesian partitioning

- **Example: remain_dims = { false, false }**



→ Ten zero-dimensional subcommunicators created as a result

→ Each subcommunicator contains only one process

Freeing Communicators

- **Communicators take up memory and other precious resources**
- **Should be freed once no longer needed**

```
MPI_Comm_free (MPI_Comm *comm)
```

→ Marks **comm** for deletion

→ **comm** is set to **MPI_COMM_NULL** on return

→ The actual communicator object is only deleted once all pending operations are completed

- **Do not try to free predefined communicators such as MPI_COMM_WORLD, MPI_COMM_SELF, and MPI_COMM_NULL**

Communicators

?

Agenda

- Motivation
- Part 1
  - → Concepts
  - → Point-to-point communication
  - → Non-blocking operations
- **Part 2**
  - → Collective operations
  - → Communicators
  - → User datatypes
- Part 3
  - → Hybrid parallelization
  - → Common parallel patterns

User datatypes

- **Basic MPI datatypes can be combined into complex user datatypes**
  - → User (derived) datatypes can be further combined into even more complex derived datatypes

- **MPI datatypes are essentially instructions for accessing the contents of the buffer's memory**
  - → type sequence – *(basic datatype, displacement)*
    - → displacements are relative to the beginning of the memory buffer and can be positive or negative
  - → type map – { ($type_0$, $disp_0$), …, ($type_{n-1}$, $disp_{n-1}$) }
  - → type signature – { $type_0$, …, $type_{n-1}$ }
- **The type signature at the sender <u>must</u> match that at the receiver**

Characteristics of a datatype

- **Lower and upper bounds:**
  - → *lb*(datatype) = min $disp_j$
  - → *ub*(datatype) = max ($disp_j$ + sizeof($type_j$)) + padding
- **Extent**
  - → *extent*(datatype) = *ub*(datatype) – *lb*(datatype)
  - → The span in memory from the first to the last basic element
- **Size**
  - → *size*(datatype) = sum sizeof($type_j$)
  - → The total amount of bytes taken by the datatype, not counting any gaps in it

Characteristics of a datatype

- **Example: MPI_INT**
  - → type map = (*int*, 0)
  - → *lb* = 0
  - → *ub* = 4
  - → *extent* = 4 bytes
  - → *size* = 4 bytes

- **All predefined basic MPI datatypes have lower bound of 0**
- **Platform-specific alignment rules are taken into account**
  - → The upper bound is therefore adjusted if necessary

Contiguous datatypes

- **Create a sequence of elements of an existing datatype**

```
MPI_Type_contiguous (count, oldtype, newtype)
```

→ The new datatype represents a contiguous sequence of **count** elements of the old datatype

→ The elements are separated from each other by the extent of **oldtype**

→ Padding (e.g. for alignment) is therefore automatically taken care of

→ A send/receive of one element of **newtype** is congruent with a receive/send of **count** elements of **oldtype**

- **Useful for sending whole matrix rows (C/C++) or columns (Fortran)**

Vector datatypes

- **Create a sequence of equally spaced blocks of elements**

```
MPI_Type_vector (count, blen, stride, oldtype, newtype)
```

  → The new datatype represents a sequence of **count** blocks, each containing **blen** elements of the old datatype

  → Every two consecutive blocks are separated by **stride** <u>elements</u> each

- **Useful for sending matrix columns (C/C++) or rows (Fortran)**

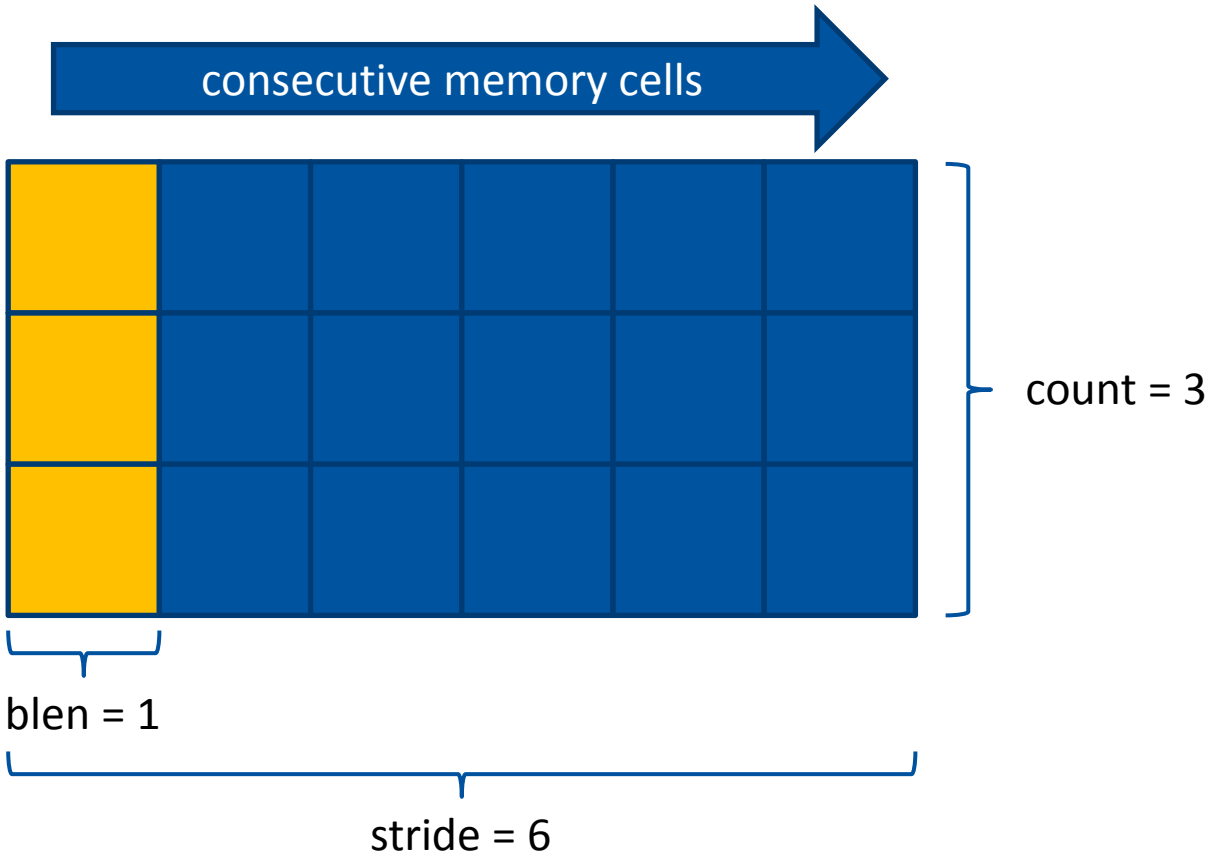
  → **stride** = row (C/C++) | column (Fortran) length (in number of elements)

  → **blen** = 1 (or the number of consecutive rows/columns)

  → **count** = number of rows (C/C++) | columns (Fortran)

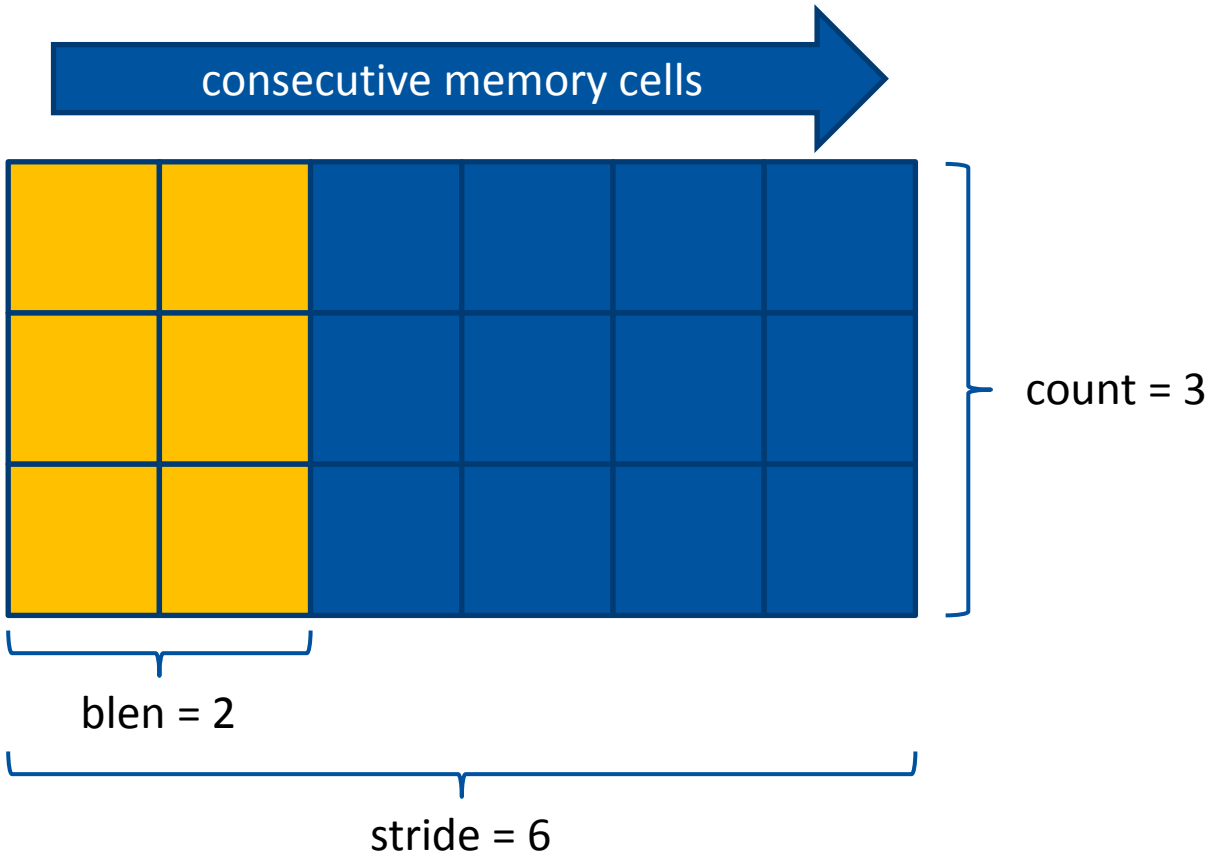
Vector datatypes

- **Example: single column of a C/C++ matrix**
  - → mat[3][6]



Vector datatypes

- **Example: two consecutive columns of a C/C++ matrix**
  - → mat[3][6]



Using derived datatypes

- **Register a datatype for use with communication operations**

```
MPI_Type_commit (datatype)
```

→ A datatype must be committed before it can be used in communications

→ All predefined datatypes are already committed

→ Intermediate datatypes, i.e. ones used in building more complex datatypes but not used in communication, can be left uncommitted

- **Deregister and free a datatype**

```
MPI_Type_free (datatype)
```

→ Derived datatypes, build from the freed datatype, are not affected

Usage of derived datatypes

- **Datatypes can be mixed and matched on both sides of a communication operation as long as their type signatures match**
  - → E.g. one can send 10 **MPI_INT** elements and receive them as a single element of a contiguous datatype with **count = 10** and **oldtype** = **MPI_INT**
  - → Extra care should be taken when using derived datatypes in collective operations

- **If the amount of data in the received message is not enough to build an integral number of elements of a derived datatype, a count of MPI_UNDEFINED is returned by MPI_Get_count**

Communicators

?

Agenda

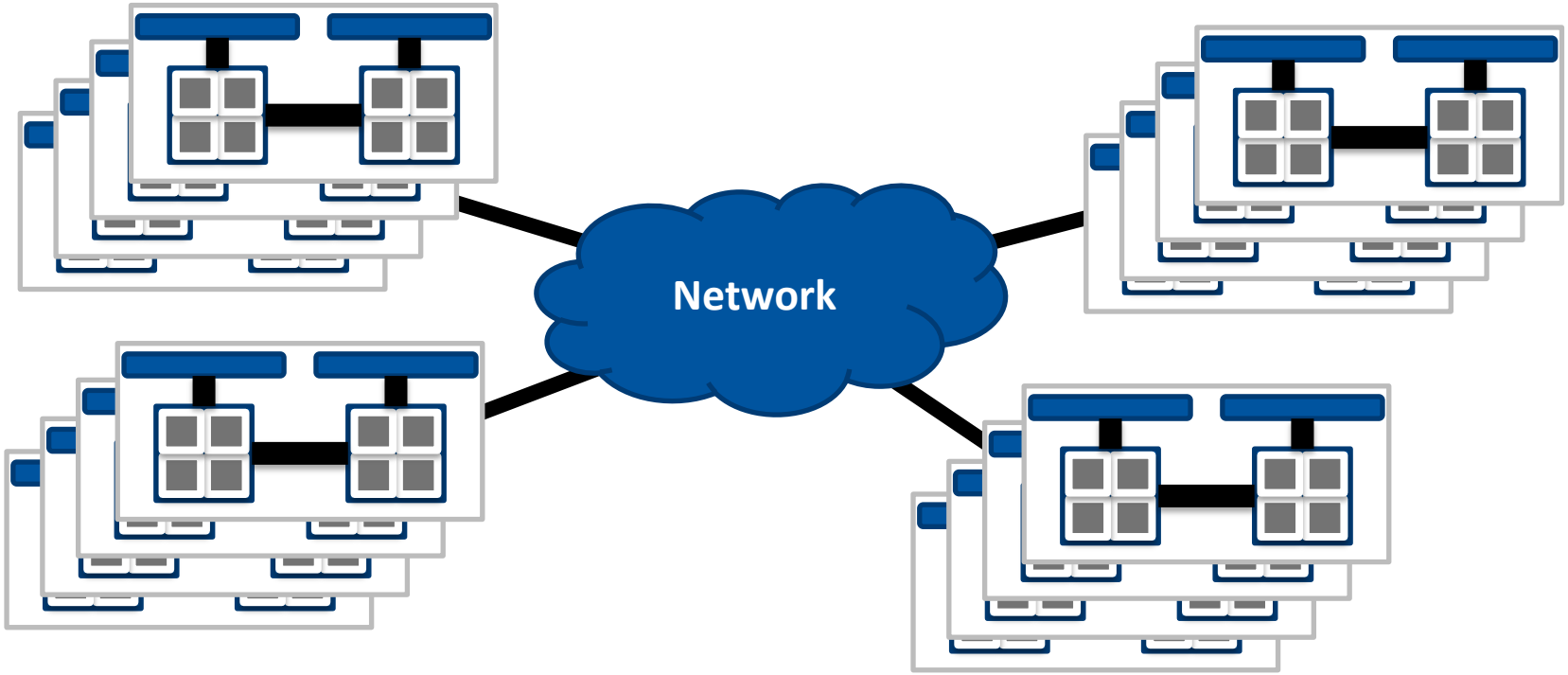
- Motivation
- Part 1
  - → Concepts
  - → Point-to-point communication
  - → Non-blocking operations
- Part 2
  - → Collective operations
  - → Communicators
  - → User datatypes
- **Part 3**
  - → Hybrid parallelization
  - → Common parallel patterns

Hybrid Programming: Motivation

- **MPI is sufficiently abstract so it runs perfectly fine on a single node:**
  - → it doesn't care where processes are located as long as they can communicate
  - → message passing implemented using shared memory and IPC
    - → all details hidden by the MPI implementation;
    - → usually faster than sending messages over the network;
  - → but…
- **… this is far from optimal:**
  - → MPI processes are separate (heavyweight) OS processes
  - → portable data sharing is hard to achieve
  - → lots of program control / data structures have to be duplicated (uses memory)
  - → reusing cached data is practically impossible

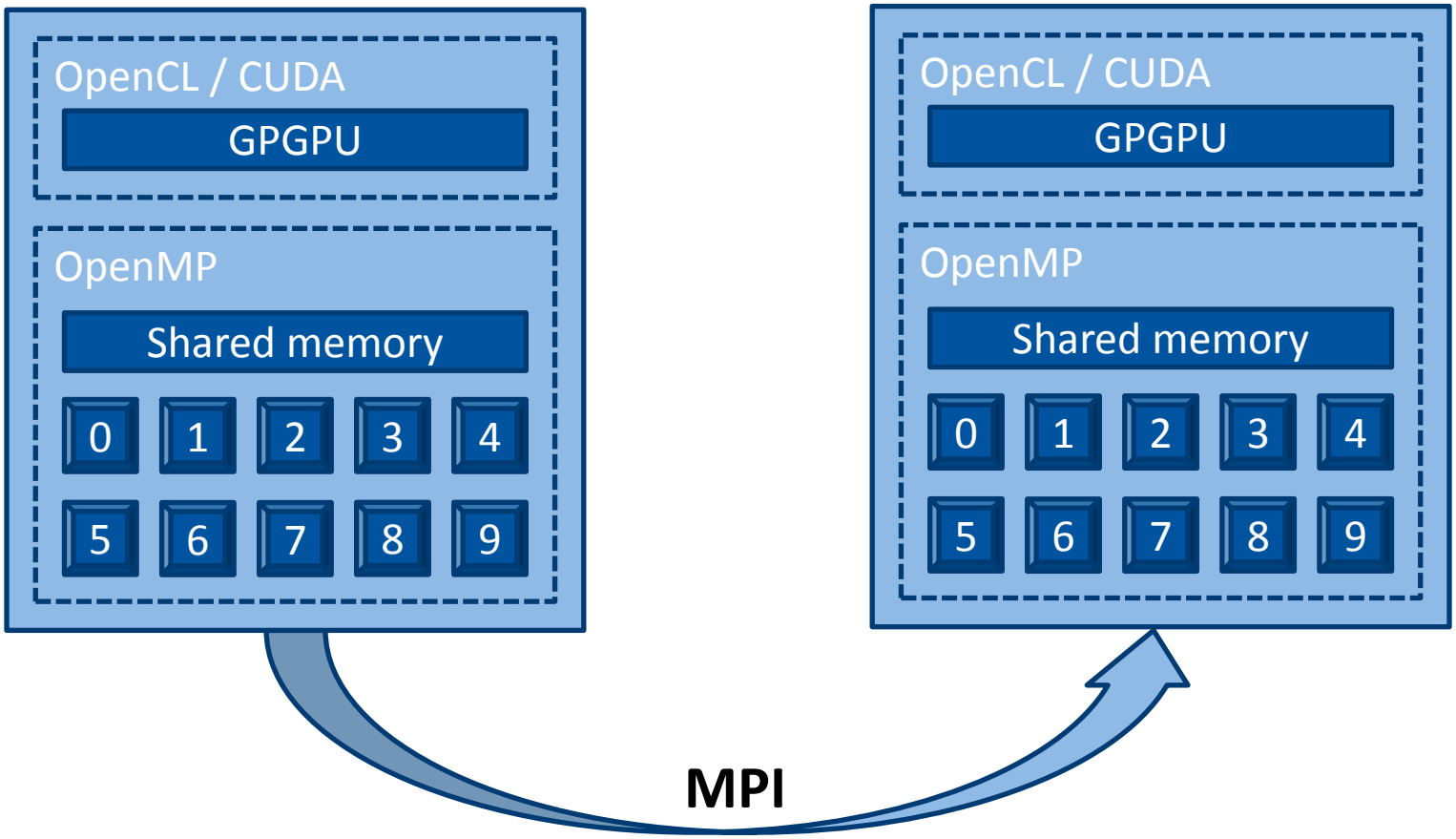
Distributed Memory Architecture

- **Clusters**
  - → HPC market is at large dominated by distributed memory *multicomputers* and *clusters*
  - → Nodes have no direct access to other nodes' memory and usually run their own (possibly stripped down) copy of the OS



Basic Idea

- **Hierarchical mixing of different programming paradigms**



MPI – Threads Interaction

- **Most MPI implementation are threaded (e.g. for non-blocking requests) but not thread-safe.**

- **Four levels of threading support in increasing order:**

Level identifier	Description
`MPI_THREAD_SINGLE`	Only one thread may execute
`MPI_THREAD_FUNNELED`	Only the main thread may make MPI calls
`MPI_THREAD_SERIALIZED`	Only one thread may make MPI calls at a time
`MPI_THREAD_MULTIPLE`	Multiple threads may call MPI at once with no restrictions

- **All implementations support MPI_THREAD_SINGLE, but some does not support MPI_THREAD_MULTIPLE.**

Threaded Initialization

- **Initialize MPI with thread support:**

```
MPI_Init_thread (int *argc, char ***argv, int required, int *provided)

SUBROUTINE MPI_INIT_THREAD (required, provided, ierr)
```

  → **required** specifies what thread level support one requires from MPI

  → **provided** is set to the actual thread level support provided

    → could be lower or higher than the required level – always check!

  → **MPI_Init** – equivalent to **required = MPI_THREAD_SINGLE**

- **The thread that called MPI_Init_thread becomes the main thread.**

- **The level of thread support cannot be changed later.**

Query Functions

- **Obtain the provided level of thread support:**

```
MPI_Query_thread (int *provided)
```

→ If MPI was initialized by **MPI_Init_thread**, then **provided** is set to the same value as the one returned by the initialization call

→ If MPI was initialized by **MPI_Init**, then **provided** is set to an implementation specific default value

- **Find out if running inside the main thread:**

```
MPI_Is_thread_main (int *flag)
```

→ **flag** set to **true** if running in the main thread

MPI Thread Support @ RWTH cluster

- **Both official MPI libraries support threads:**

requested	Open MPI	Open MPI mt	Intel MPI w/o -mt_mpi	Intel MPI w/ -mt_mpi
SINGLE	SINGLE	SINGLE	SINGLE	SINGLE
FUNNELED	SINGLE	FUNNELED	SINGLE	FUNNELED
SERIALIZED	SINGLE	SERIALIZED	SINGLE	SERIALIZED
MULTIPLE	SINGLE	MULTIPLE	SINGLE	MULTIPLE

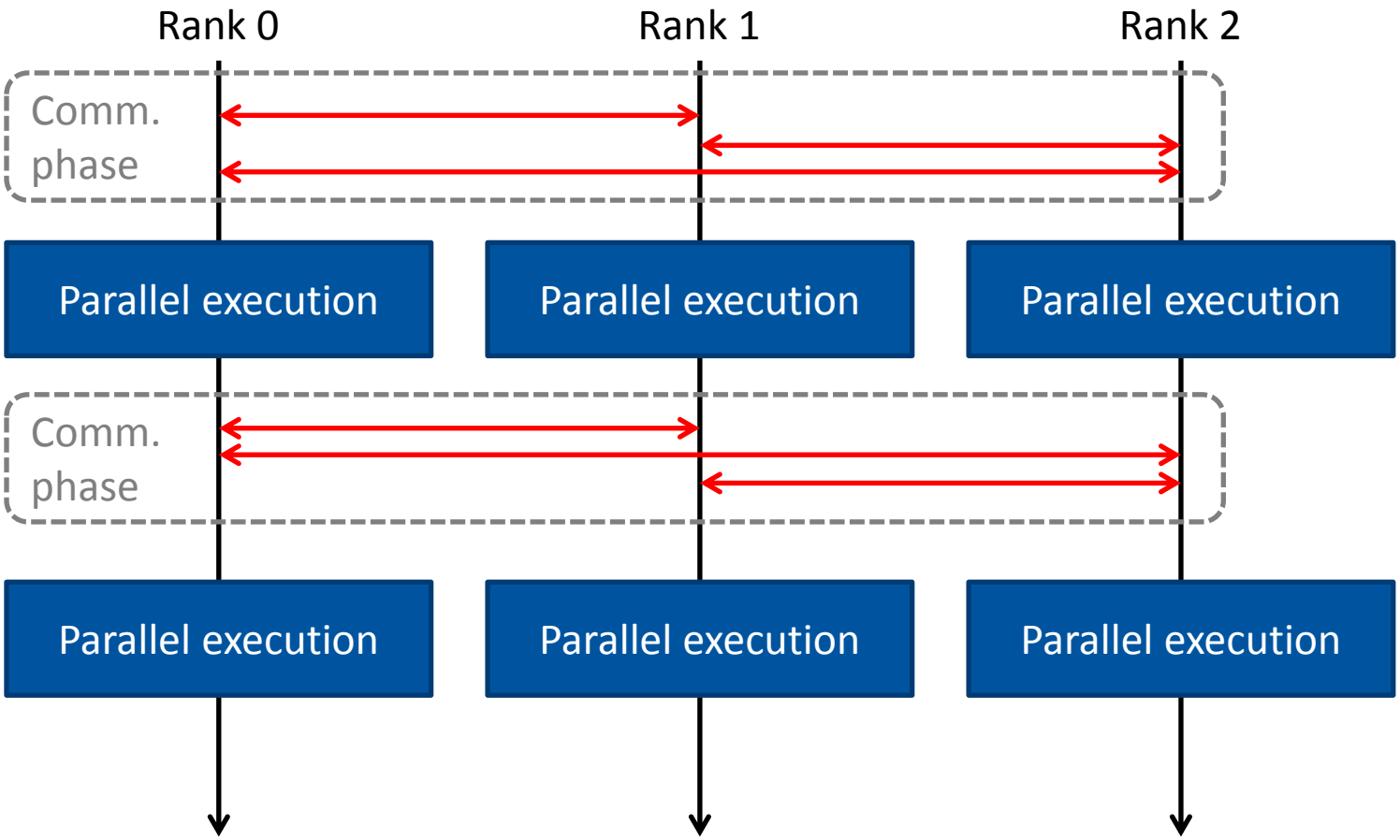
- **Open MPI**
  - → Use (non-default) module versions ending with **mt**, e.g. **openmpi/1.6.5mt**
  - → The InifiniBand BTL does not support **MPI_THREAD_MULTIPLE**!!!
- **Intel MPI**
  - → Compile with **-openmp**, **-parallel** or **-mt_mpi**

Caveats

- **Beware of possible data races:**
  - → messages, matched by **MPI_Probe** in one thread, can be received by a matching receive in another thread, stealing the message from the first one
- **MPI provides no way to address specific threads in a process**
  - → clever use of message tags
  - → clever use of many communicators
- **In general thread-safe MPI implementations perform worse than non-thread-safe because of the added synchronization overhead.**
- **Don't use Open MPI on our cluster if full thread support is required.**

Typical Hybrid Application

- **Separate computation and communication phases**
  - → Most portable since no full thread safety is required from the MPI library



Now What?

- **Here is a list of important MPI topic that this course does not cover:**
  - → Virtual topologies
  - → Parallel I/O

- **And a list of more exotic topics:**
  - → Dynamic process control
  - → Client/server relations
  - → One-sided operations

?

Agenda

- Motivation
- Part 1
  - → Concepts
  - → Point-to-point communication
  - → Non-blocking operations
- Part 2
  - → Collective operations
  - → Communicators
  - → User datatypes
- **Part 3**
  - → Hybrid parallelization
  - → Common parallel patterns

No parallel program can outrun the sum of its sequential parts!

Keep this in mind for the rest of your (parallel) life!

Amdahl's Law in Theory

- **The run time of a program consists of two parts:**
  - → serial (non-parallelizable) part: T_s
  - → parallelizable part: T_p
  - → total execution time: $T = T_s + T_p$
  - → serial share: $s = T_s / T$

- **An n-fold parallelization yields:**
  - → total execution time: $T_n = T_s + T_p / n$
  - → parallel speedup: $S_n = T / T_n = n / [1 + (n-1).s]$
  - → parallel efficiency: $E_n = T / (n.T_n) = 1 / [1 + (n-1).s]$

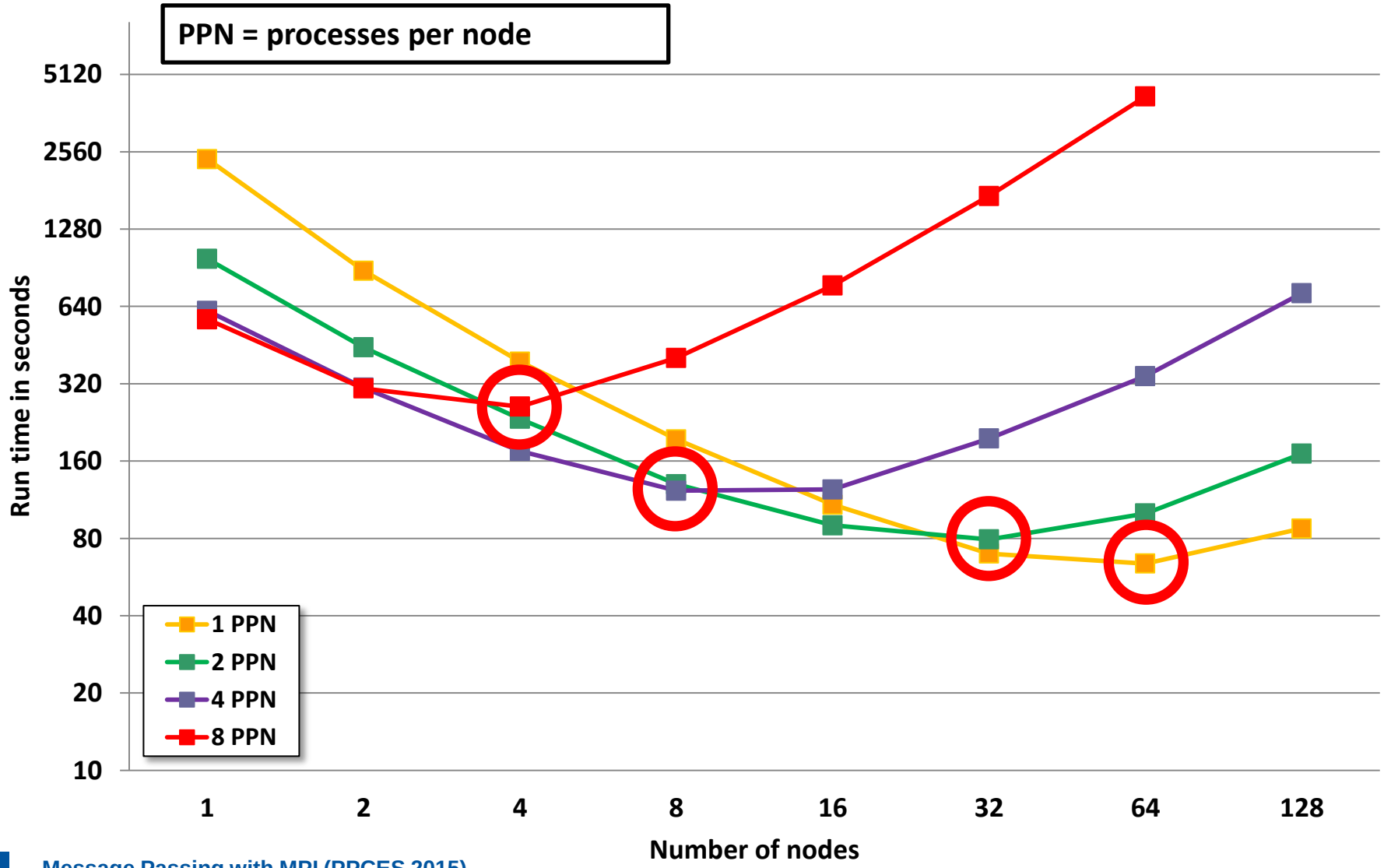
- **Asymptotic values in the limit of infinite number of processors:**
  - → total execution time: $T_\infty = T_s + T_p / \infty = T_s$
  - → parallel speedup: $S_\infty = T / T_\infty = T / T_s = 1 / s$
  - → parallel efficiency: $E_\infty = 1$ if $s = 0$; $E_\infty = 0$ otherwise

Amdahl's Law – The Ugly Truth™

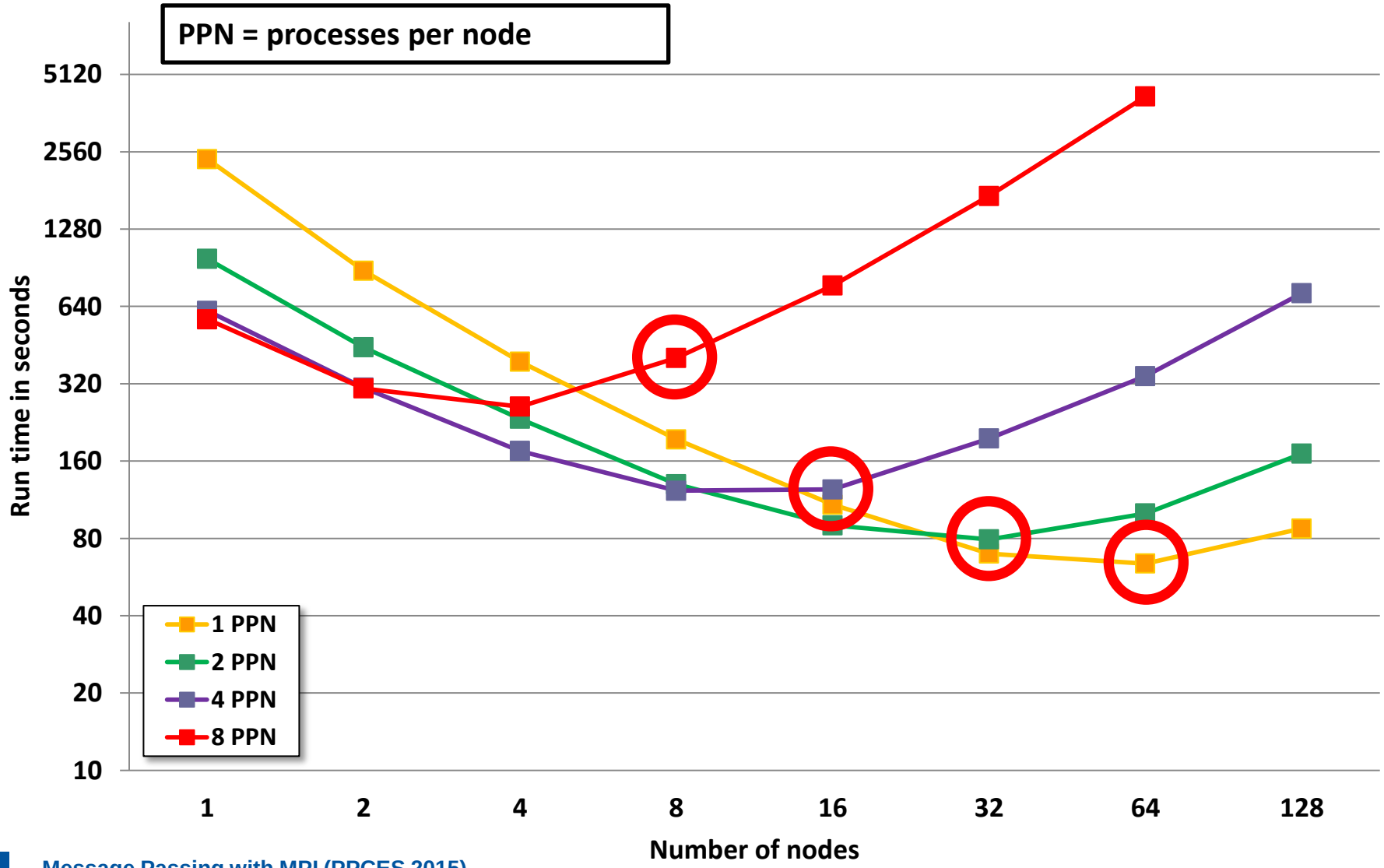
- **Parallelization usually (if not always) introduces overhead:**
  - → communication is inherently serial → *s* increases
  - → usually $E_n < 1$ – you pay more CPU time than if you run the serial program
  - → but sometimes cache effects result in $E_n > 1$ – <u>superlinear speedup</u>

- **Communication overhead increases with the number of processes:**
  - → more processes → more messages (depends on the communication pattern) especially true for the collective operations *(you did not reimplement them, did you?)*
  - → more messages → more network latency → more serial time
  - → more serial time → lower parallel efficiency
  - → with large process counts the overhead could negate the parallelization gain

Amdahl's Law – The Ugly Truth™



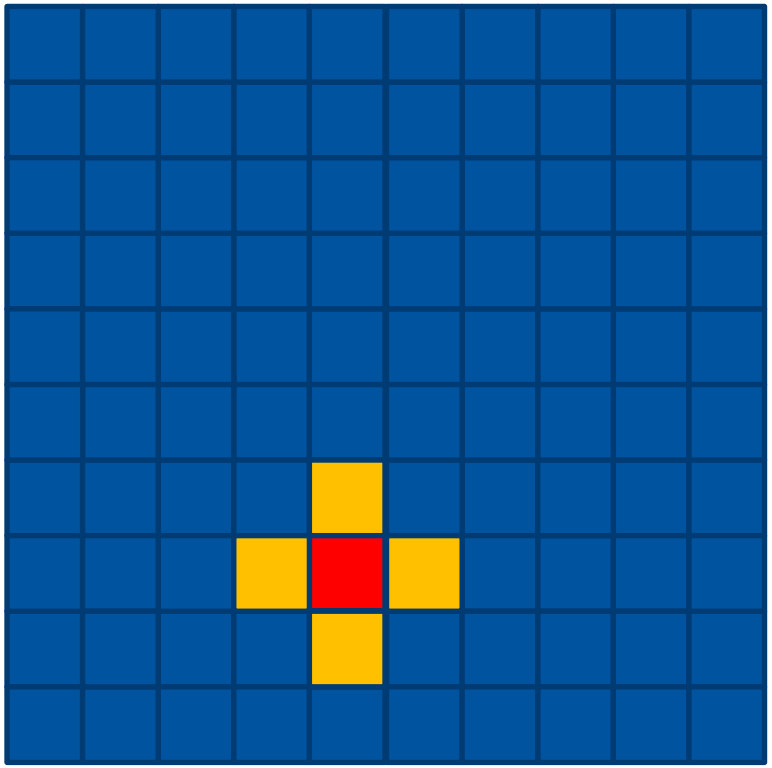
Amdahl's Law – The Ugly Truth™



Halos / Ghost Cells

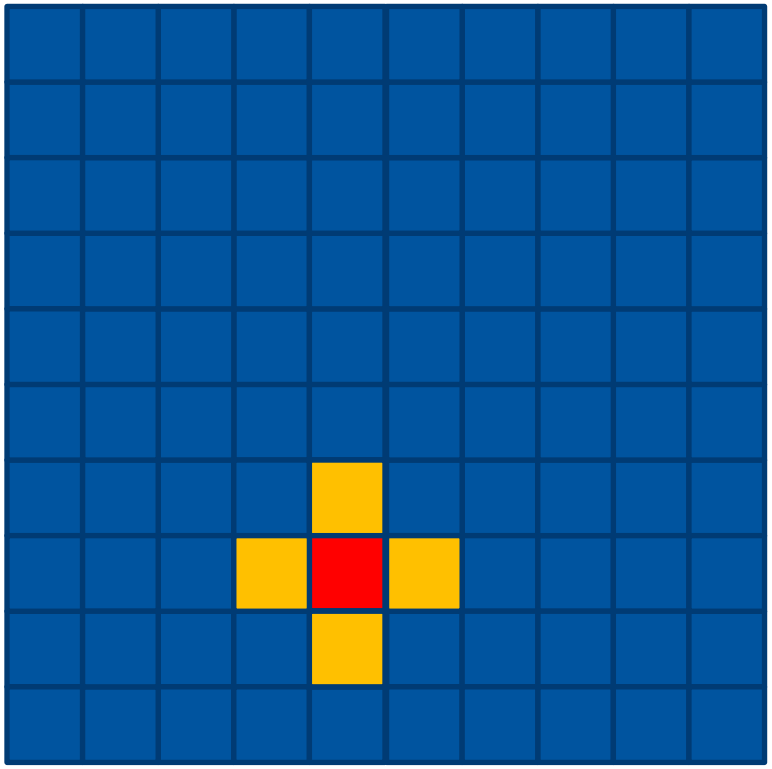
- **When decomposing a problem often interdependent data ends up in separate processes.**
- **Example: iterative matrix update in a PDE solver:**

$$cell_{i,j} = f\left(cell_{i,j}; cell_{i-1,j}, cell_{i+1,j}, cell_{i,j-1}, cell_{i,j+1}\right)$$



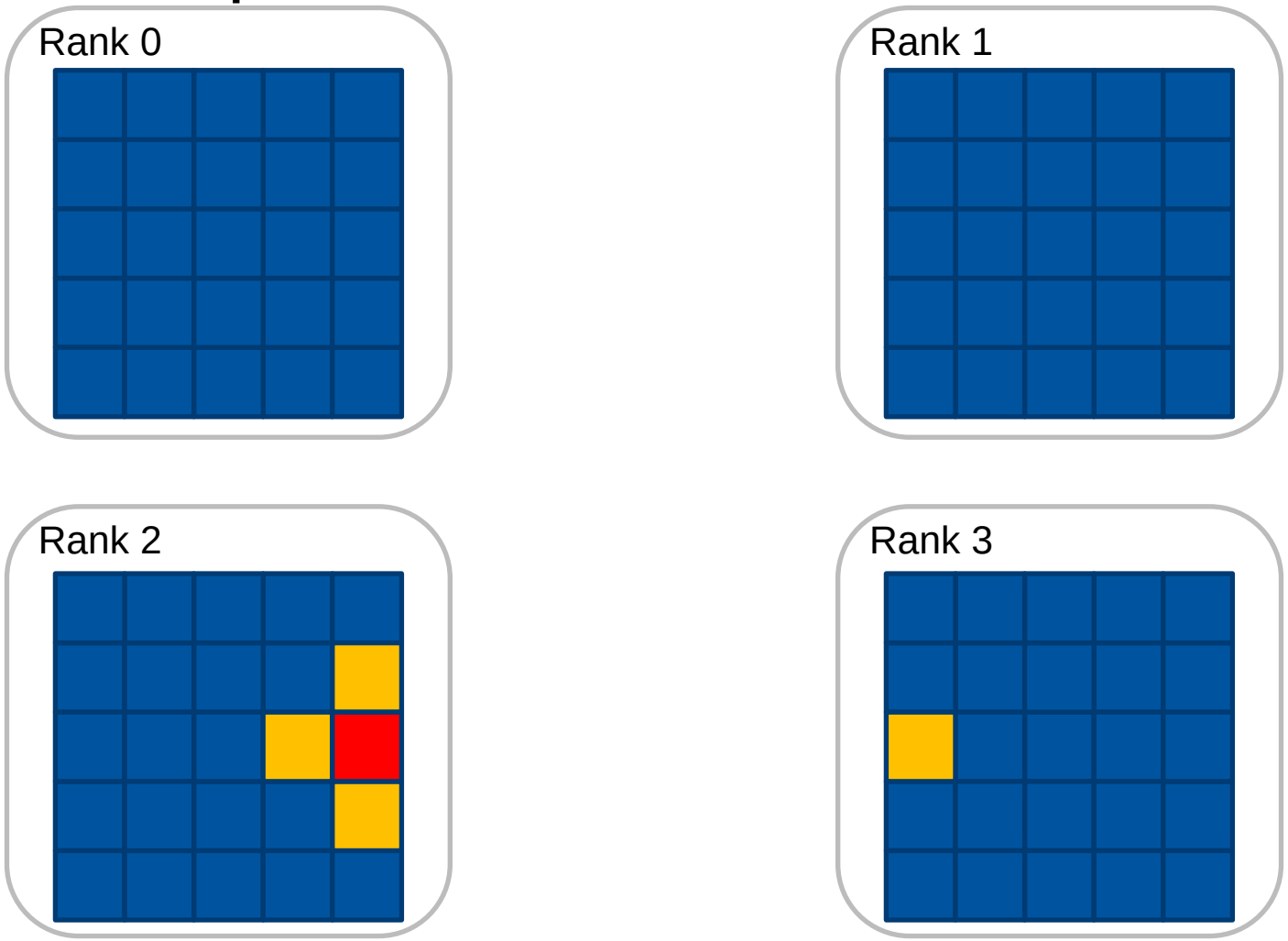
Halos / Ghost Cells

- **Domain decomposition strategy:**
  - → Partition the domain into parts.
  - → Each process works on one part only.



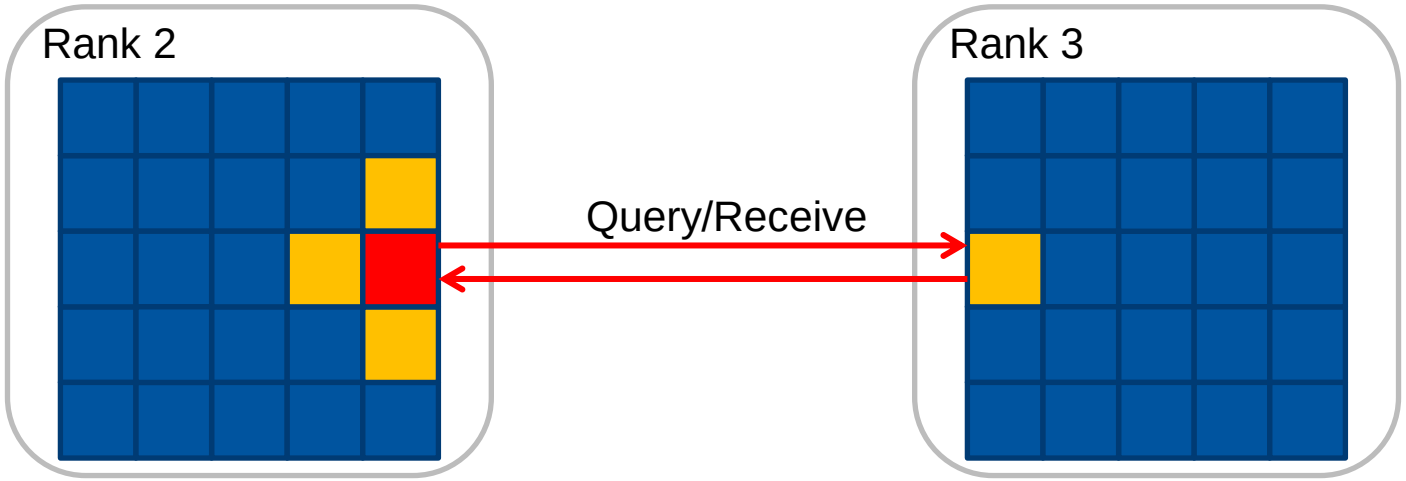
Halos / Ghost Cells

- **Domain decomposition**



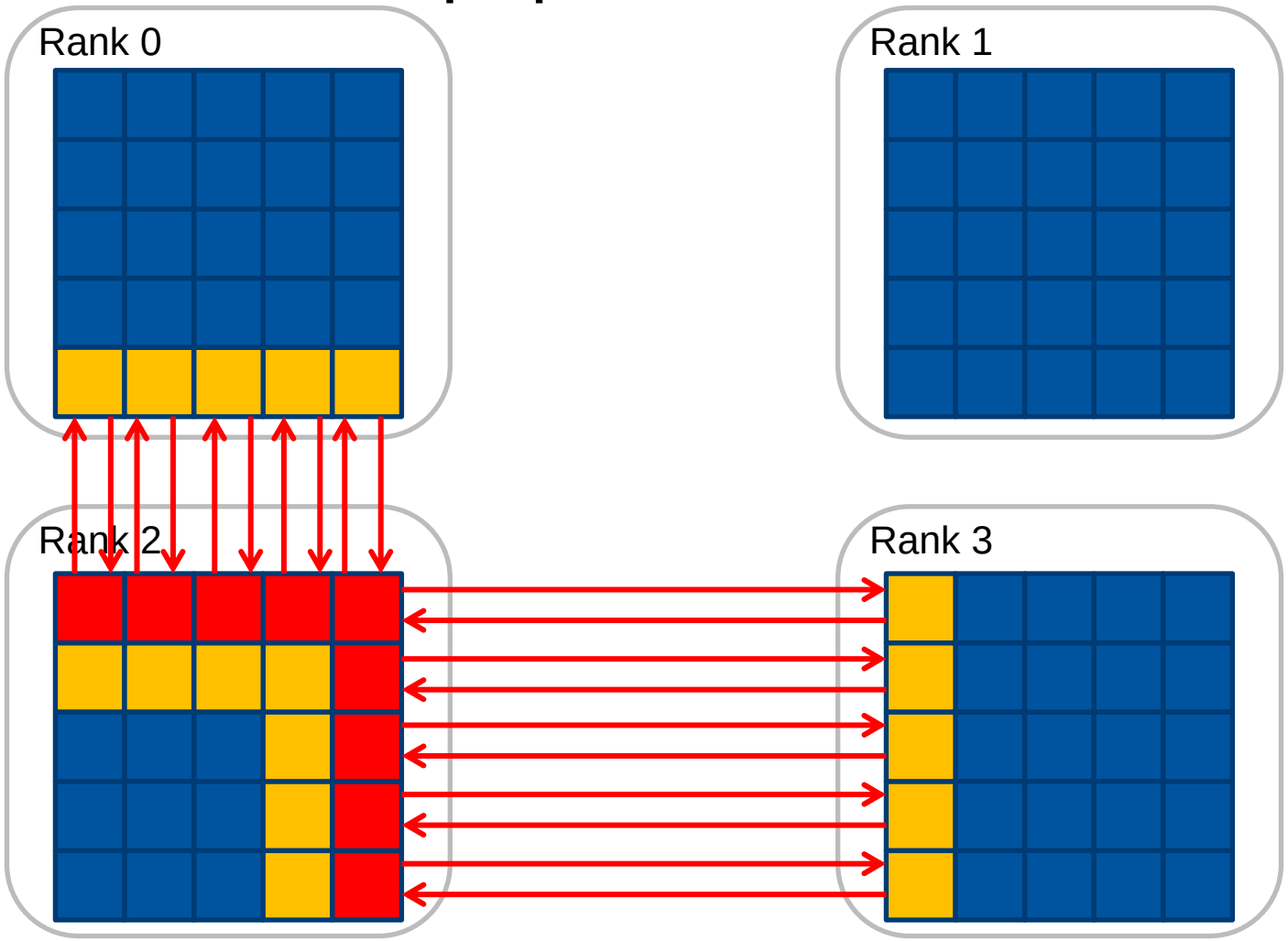
Halos / Ghost Cells

- **Communication**
  - → For every border cell communication with a neighbor rank is necessary
- **Problems:**
  - → Introduces synchronization on a very fine level
  - → Lots of communication calls – highly inefficient
  - → Performance can (and will) drastically suffer



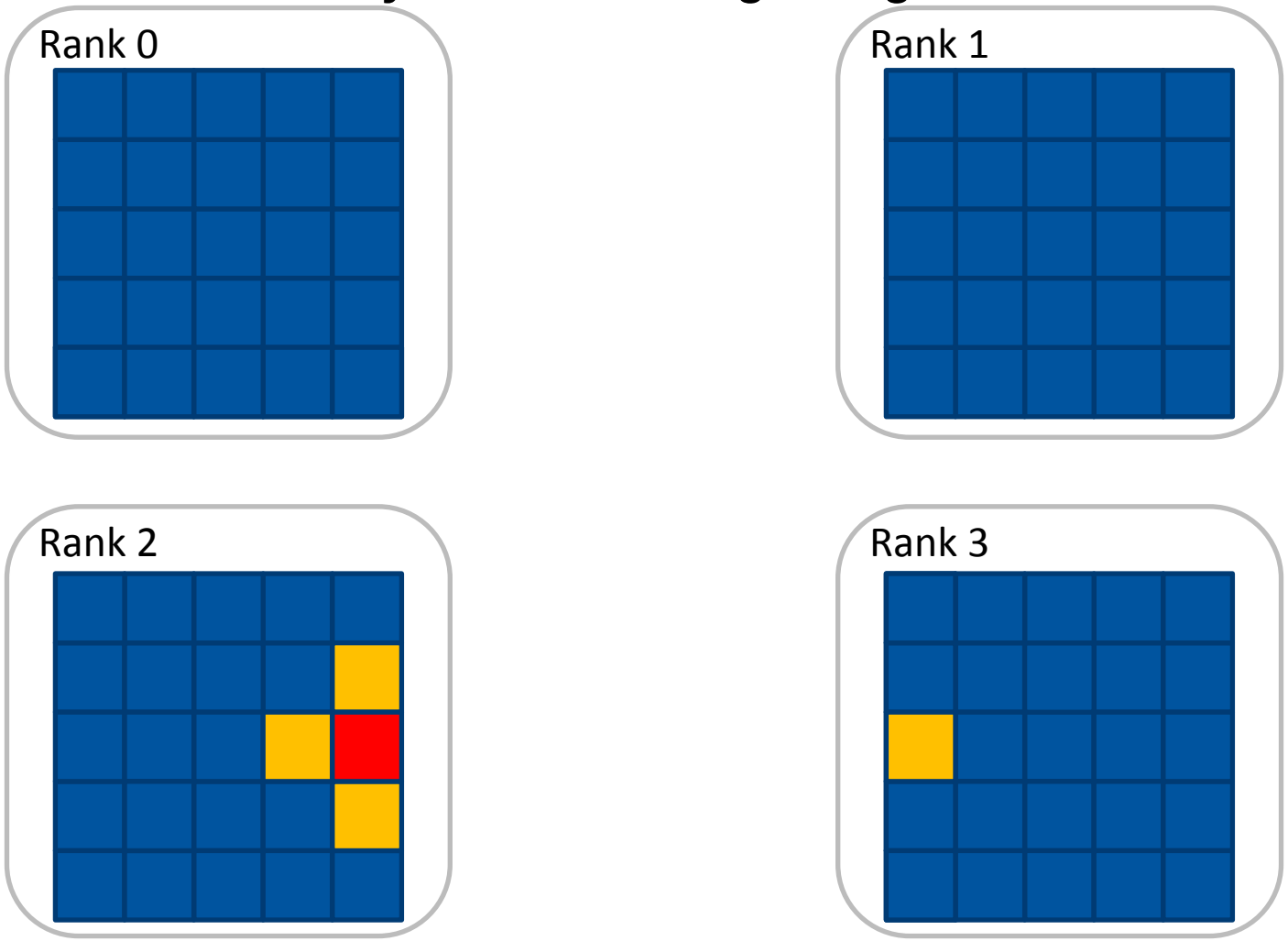
Halos / Ghost Cells

- **10 communication calls per process**



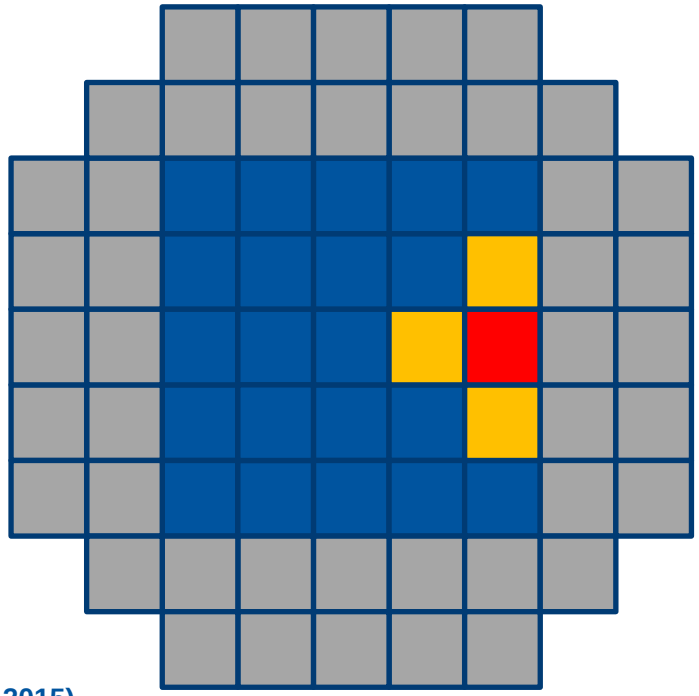
Halos / Ghost Cells

- **Copy all the necessary data at the beginning of each iteration**



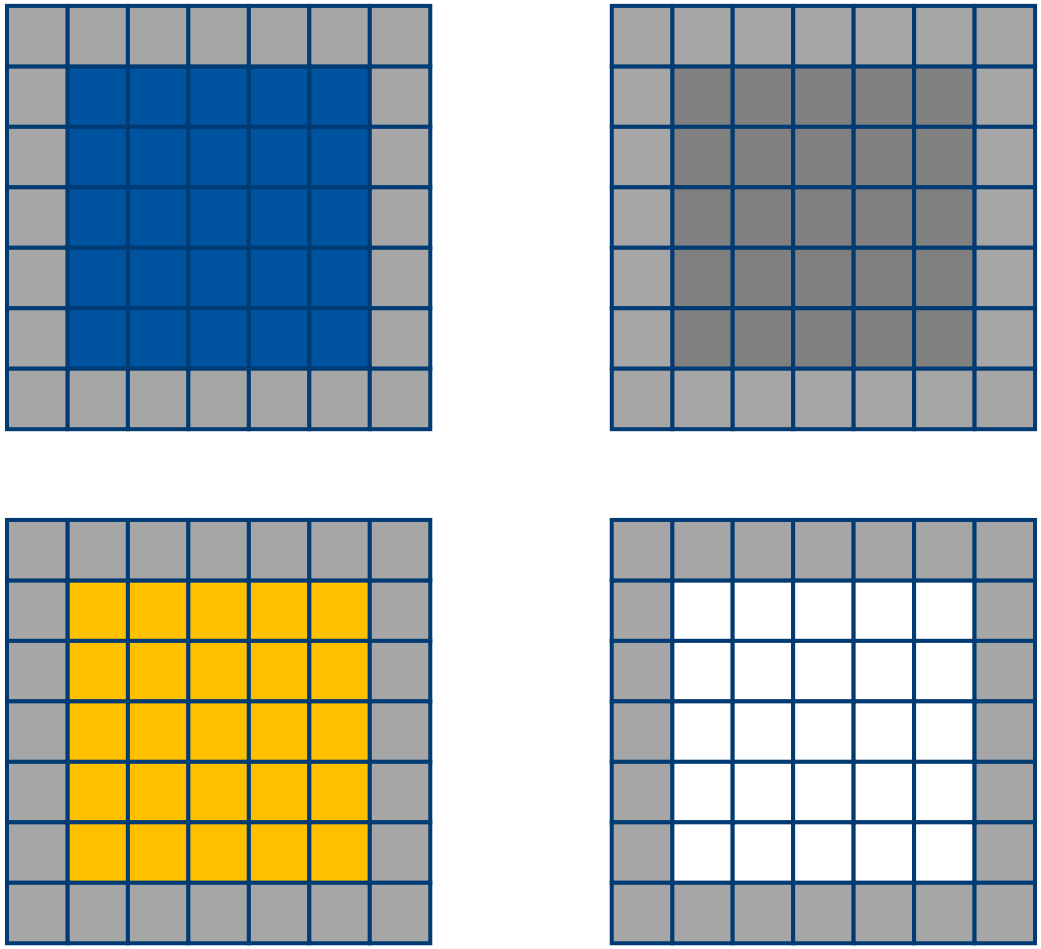
Halos / Ghost Cells

- **Copies around the host domain are called *halos* or *ghost cells***
- **Multi-level halos are also possible:**
  - → if required by the stencil
  - → reduce the number of communications



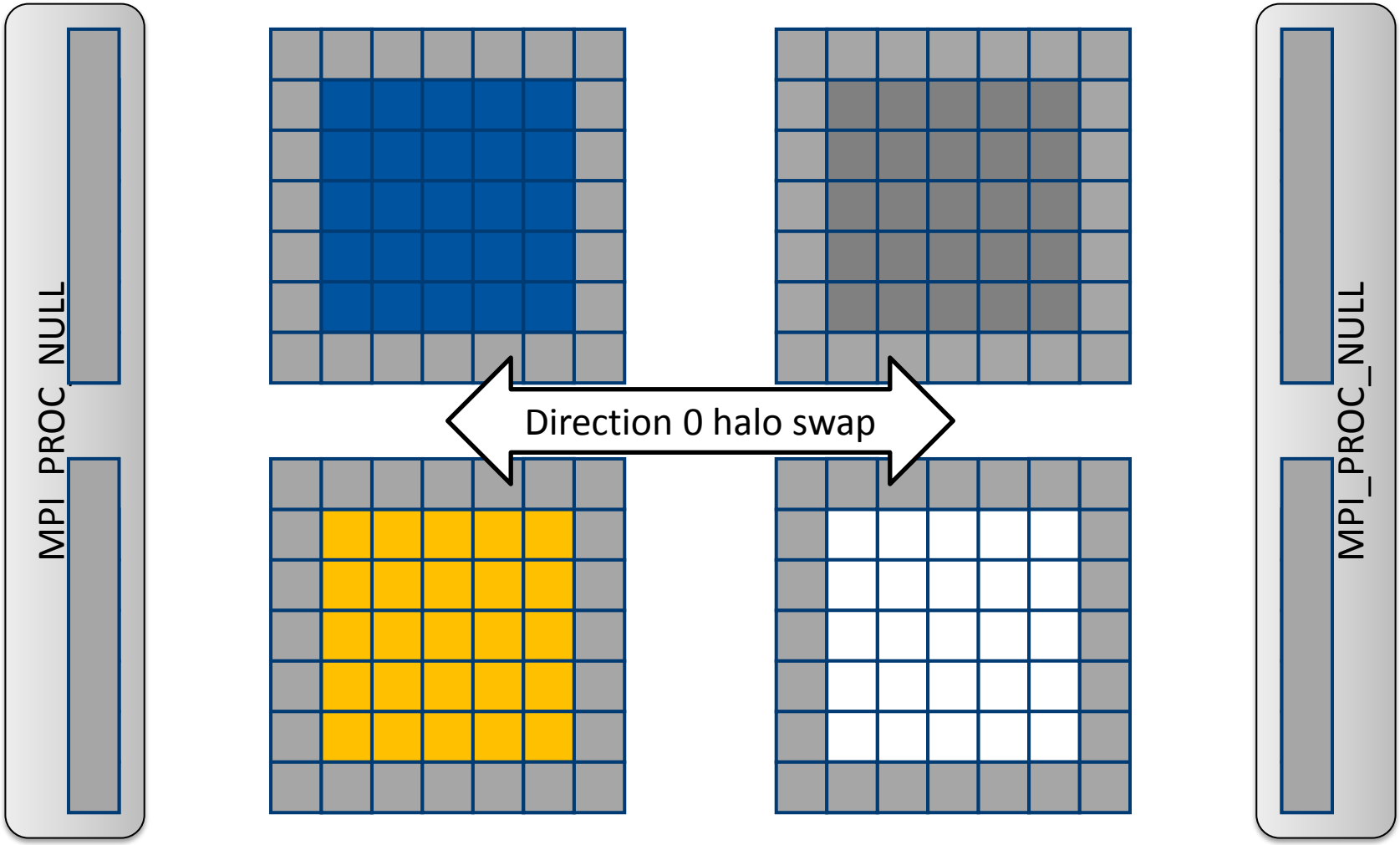
Halos / Ghost Cells

- **Halo Swap**



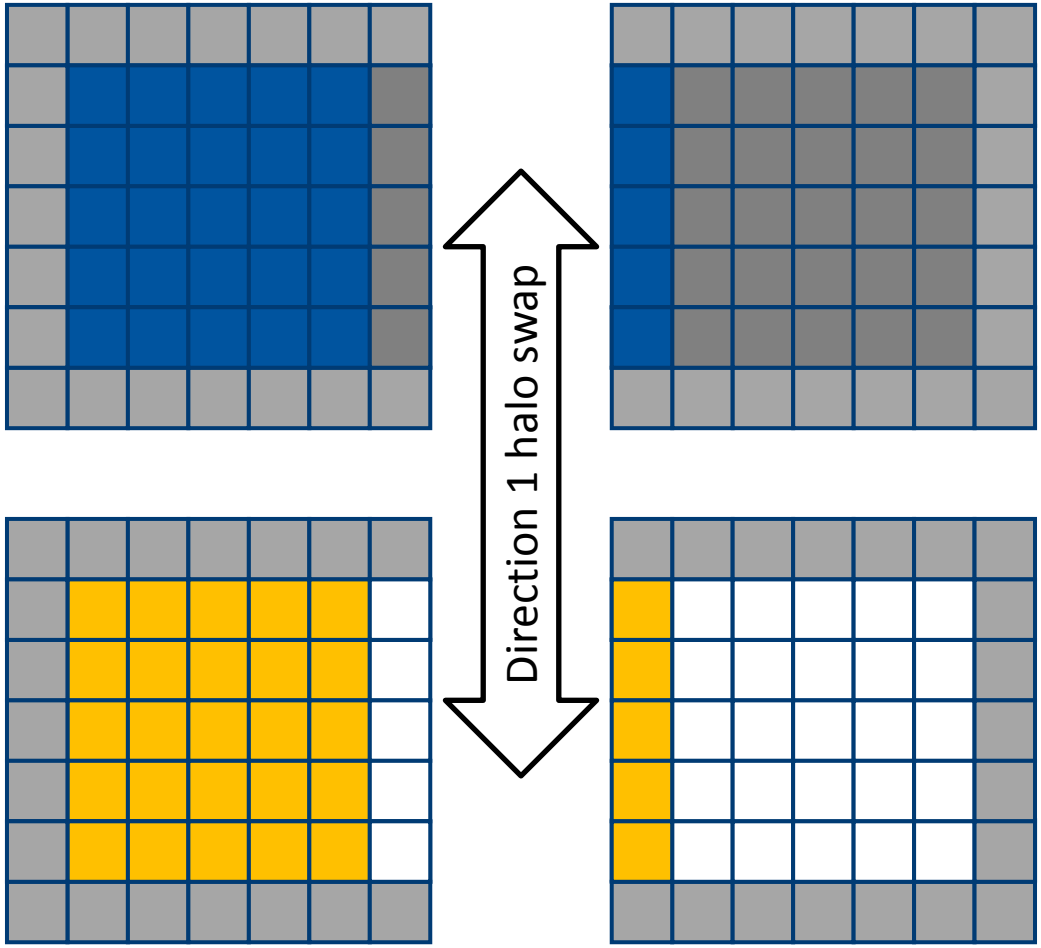
Halos / Ghost Cells

- **Halo Swap**



Halos / Ghost Cells

- **Halo Swap**



Halos / Ghost Cells

- **Halo Swaps are locally synchronous, but combined they make a globally synchronous operation:**
  - → initial process synchronization is critical for the performance;
  - → one late process delays all the other processes;
  - → sensitivity to OS jitter.

- **Pros:**
  - → idea comes naturally;
  - → simple to implement (two MPI calls per direction).

- **Cons:**
  - → not suitable for problems where load imbalance may occur.

Irregular Problems

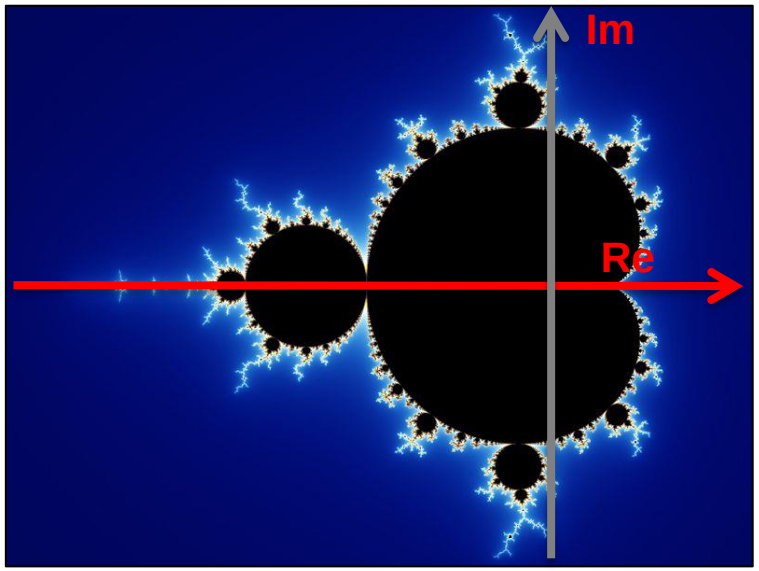
- **Escape time coloring algorithm for the Mandelbrot set**

```
For each image pixel (x, y):
// (x, y) – scaled pixel coords
c = x + i*y
z = 0
iteration = 0
maxIters = 1000
while (|z|² <= 2² &&
       iteration < maxIters)
{
   z = z² + c
   iteration = iteration + 1
}
if (iteration == maxIters)
   color = black
else
   color = iteration
plot(x, y, color)
```

Does the complex series

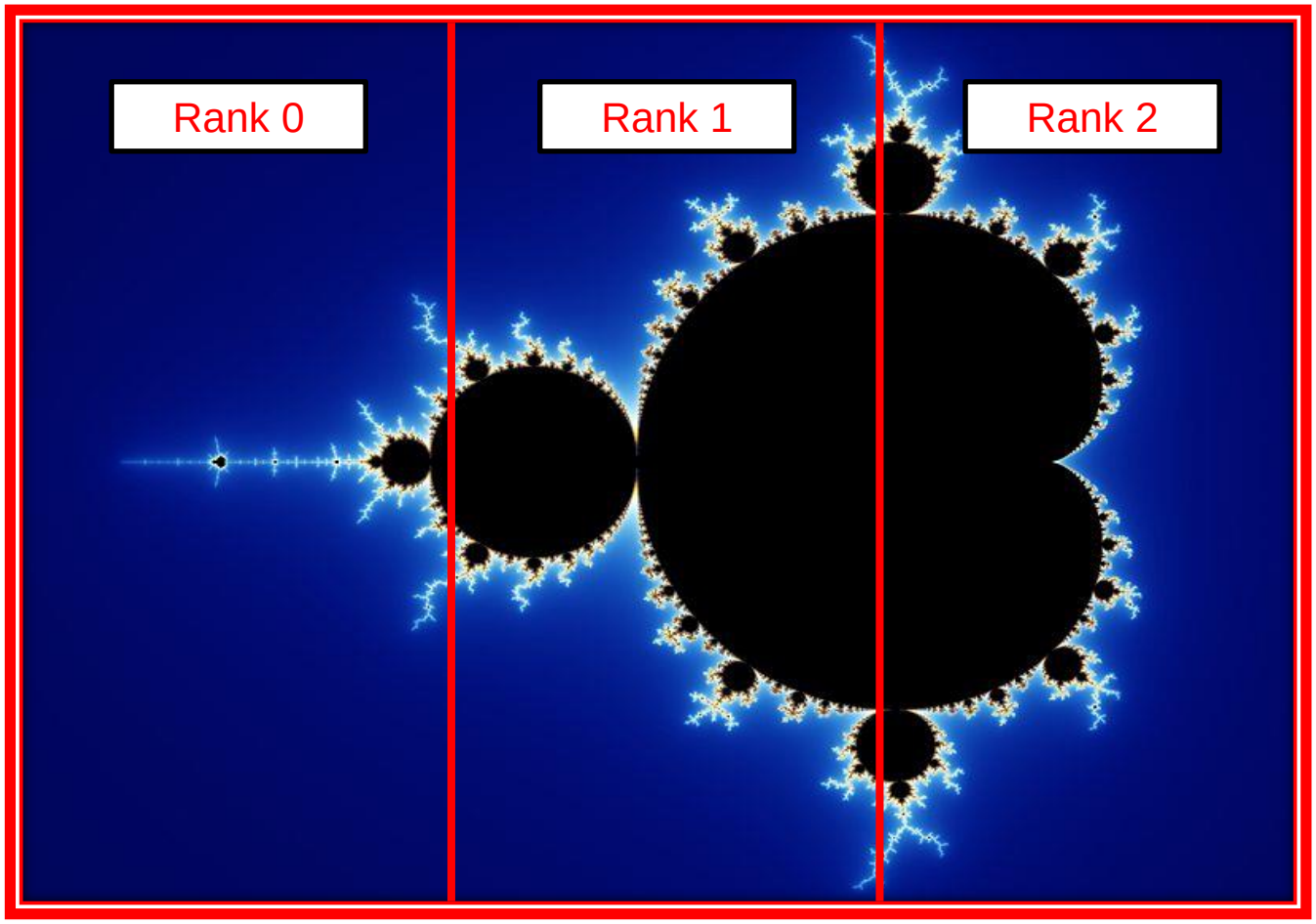
$$z_0 = 0;\ z_{n+1} = {z_n}^2 + c \quad z, c \in \mathbb{C}$$

remain bounded?



Irregular Problems

- **Static work distribution:**
  - → Every MPI rank works on 1/Nth of the problem (N – number of MPI ranks)

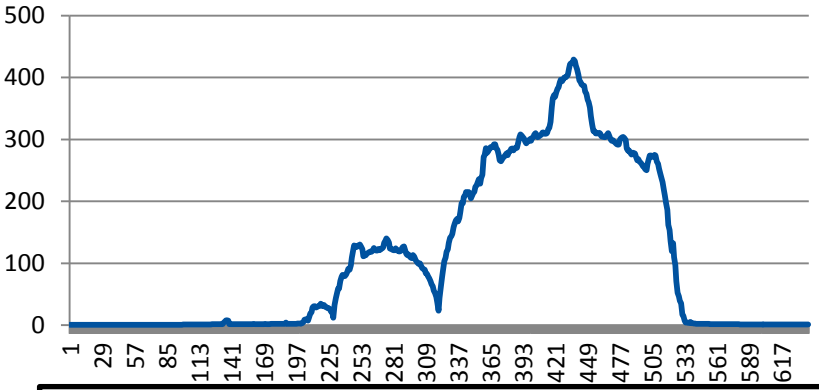


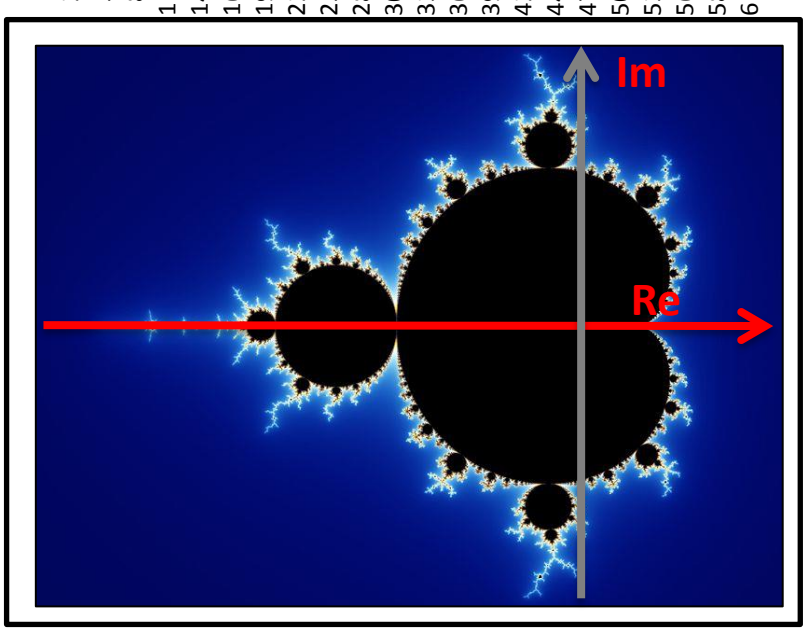
Irregular Problems

- **Mandelbrot set**

```
For each image pixel (x, y):
// (x, y) – scaled pixel coords
c = x + i*y
z = 0
iteration = 0
maxIters = 1000
while (|z|² <= 2² &&
       iteration < maxIters)
{
   z = z² + c
   iteration = iteration + 1
}
if (iteration == maxIters)
   color = black
else
   color = iteration
plot(x, y, color)
```

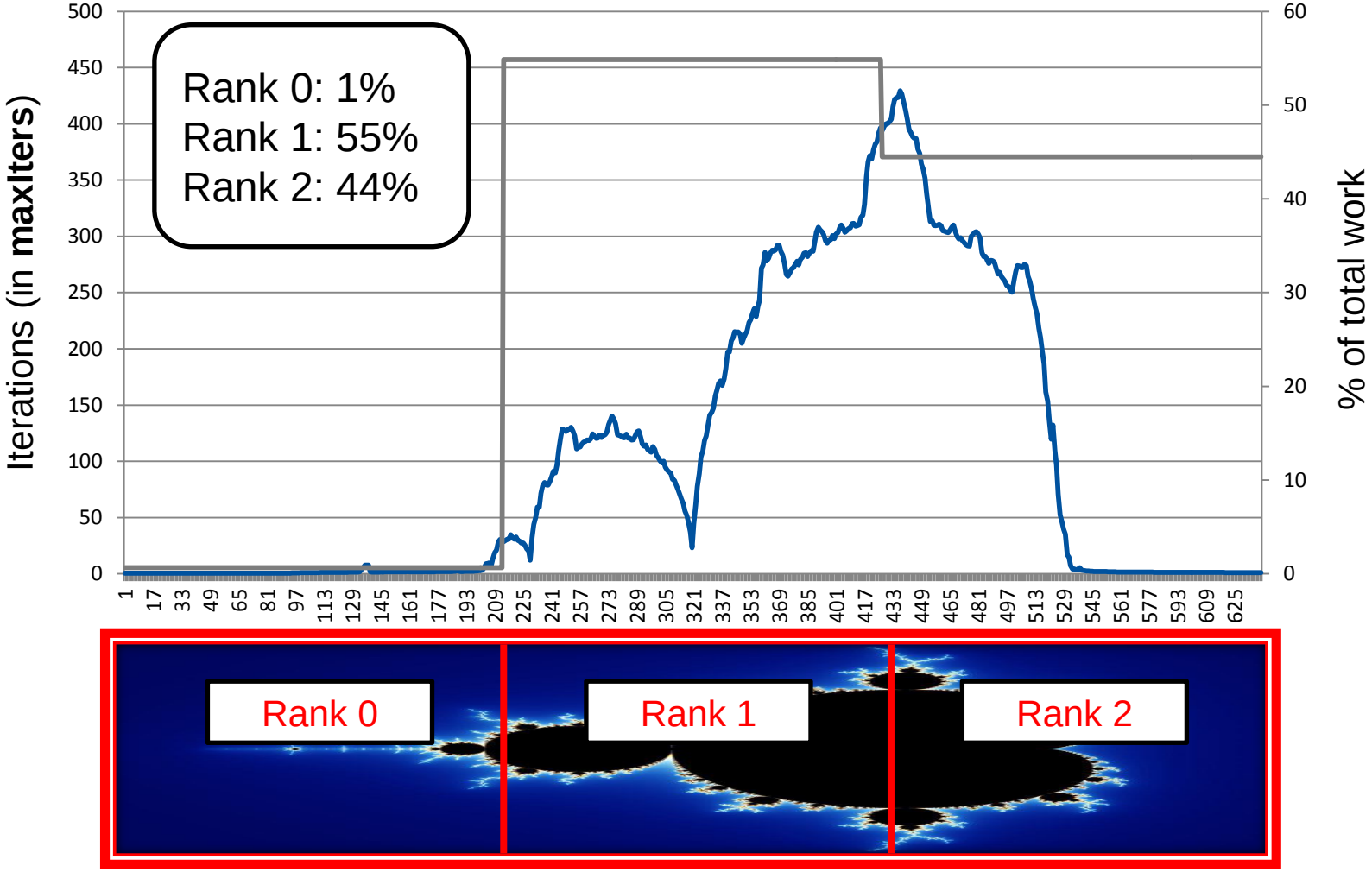
Iterations per image column (in **maxIters**)





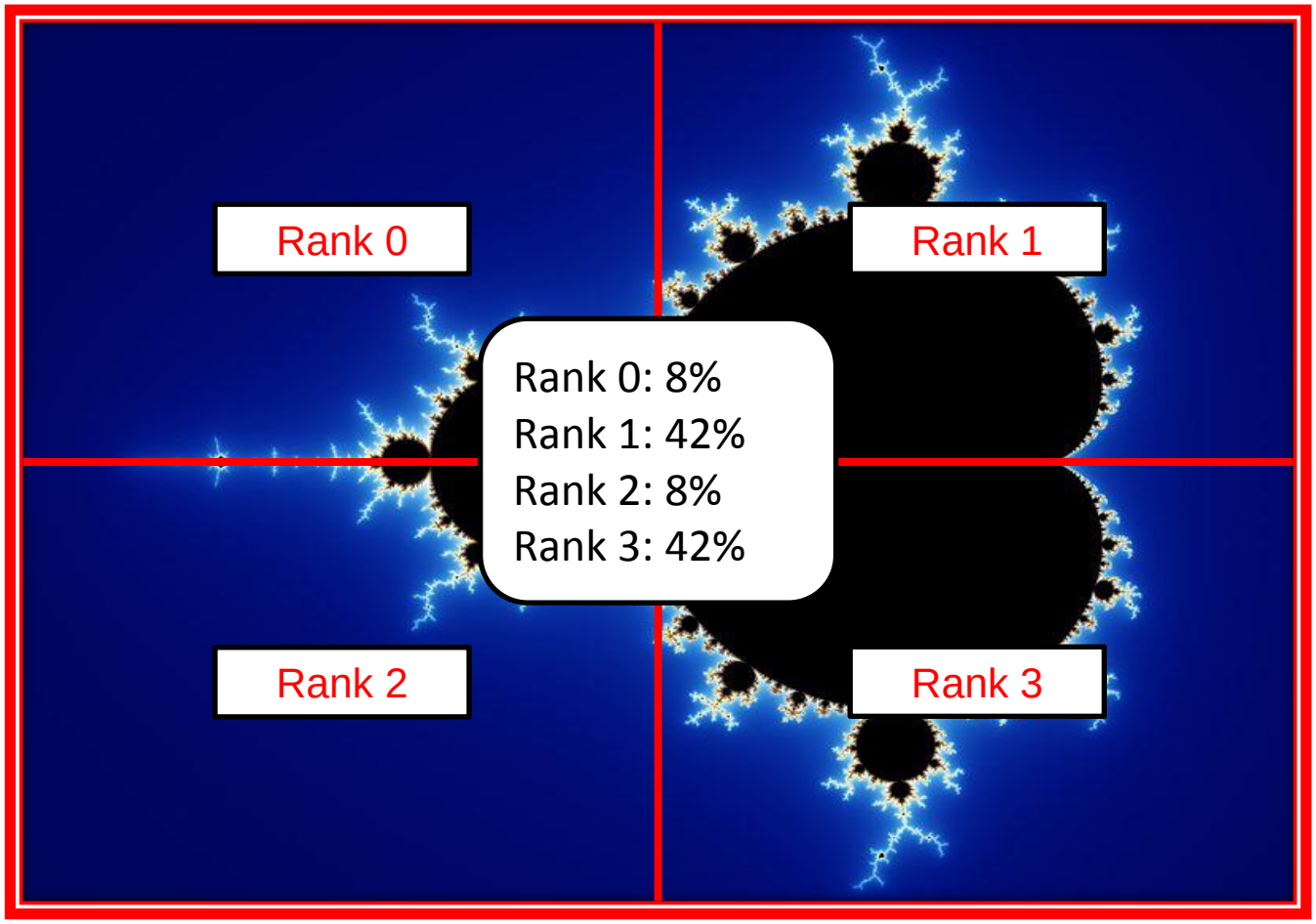
Work Imbalance

- **Computation complexity mapped to ranks**



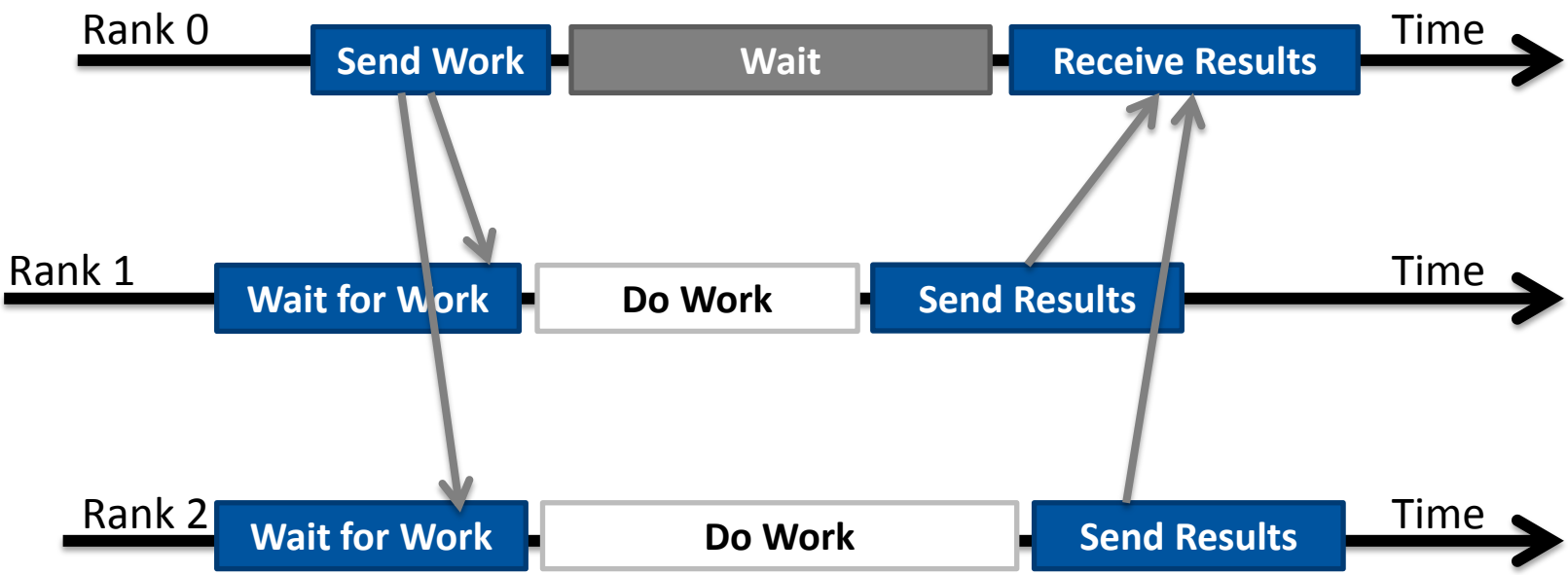
Work Imbalance

- **May be a different decomposition?**



Controller – Worker

- **One process (controller) manages the work.**
  - → Work is split into many relatively small work items.
- **Many other processes (workers) compute over the work items:**

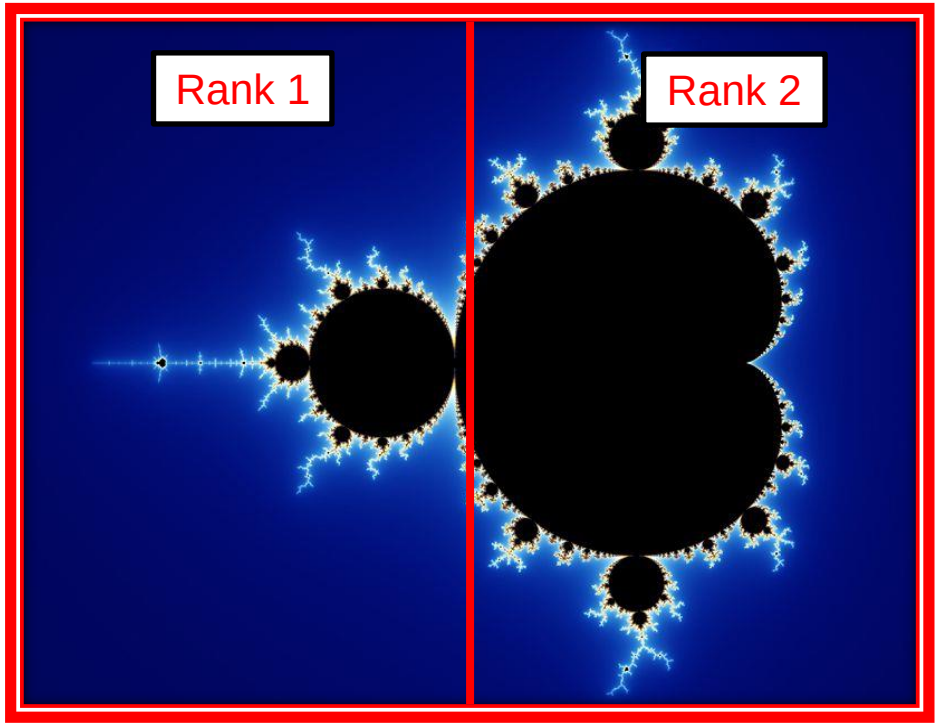


- **Above steps are repeated until all work items are processed.**
- **Sometimes called "bag of jobs" pattern.**

Controller – Worker

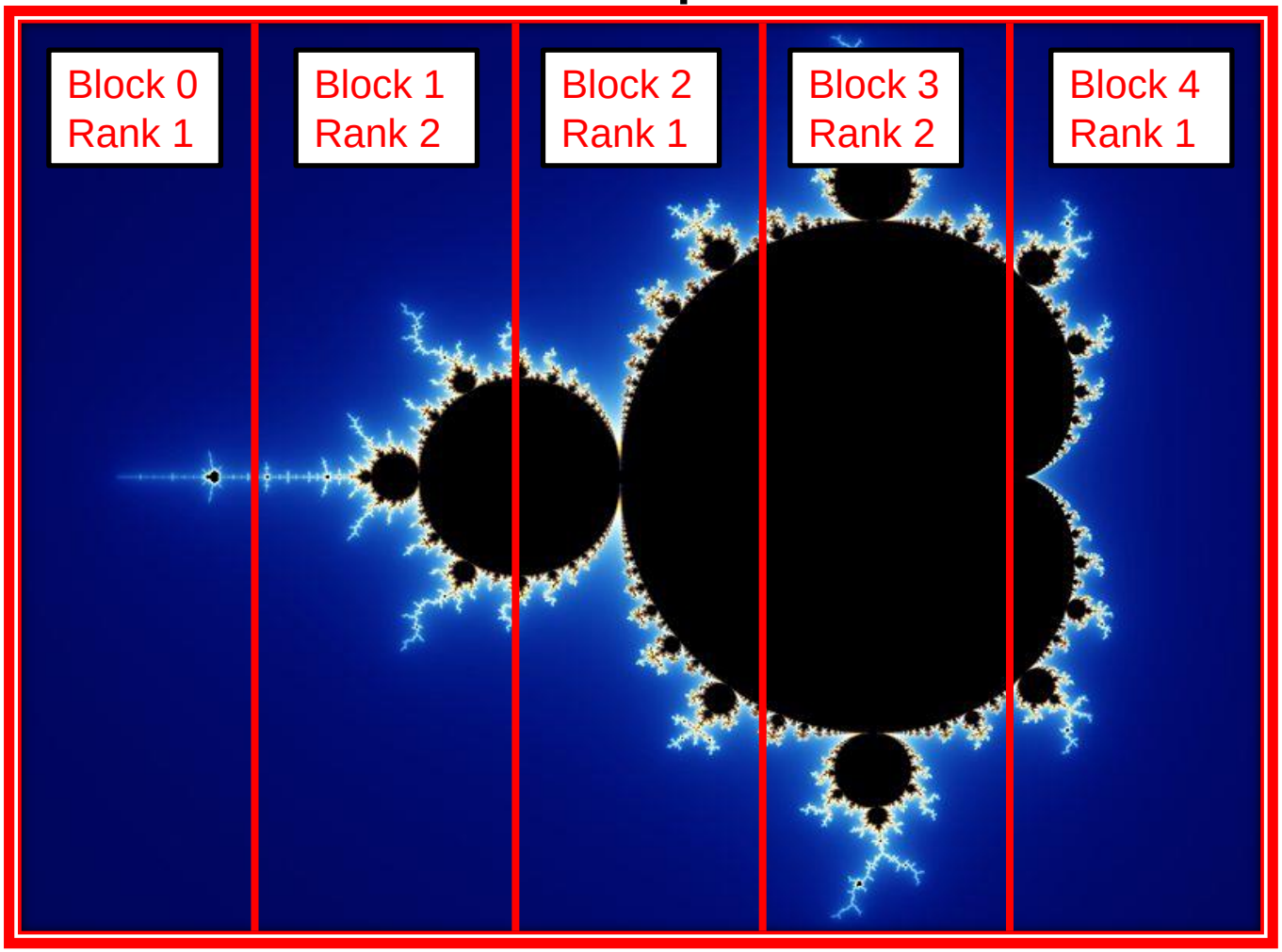
- **The algorithm:**

```
START
if (rank == 0)
{
    splitDomain;
    sendWorkItems;
    receiveItemResults;
    assembleResult;
    output;
}
else
{
    receiveWorkItems;
    processWorkItems;
    sendItemResults;
}
DONE
```



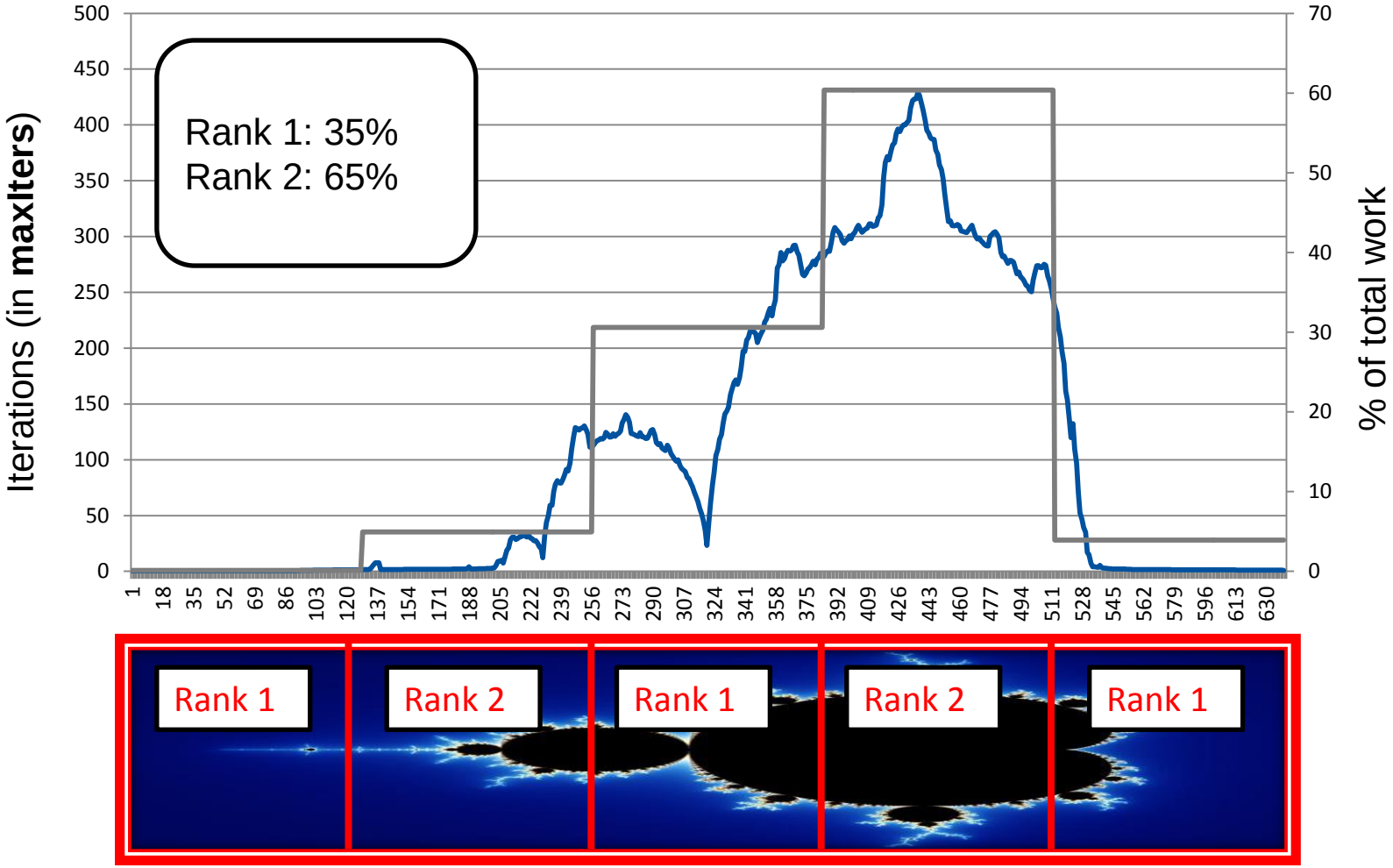
Controller – Worker

- **Controller – Worker with 3 worker processes**



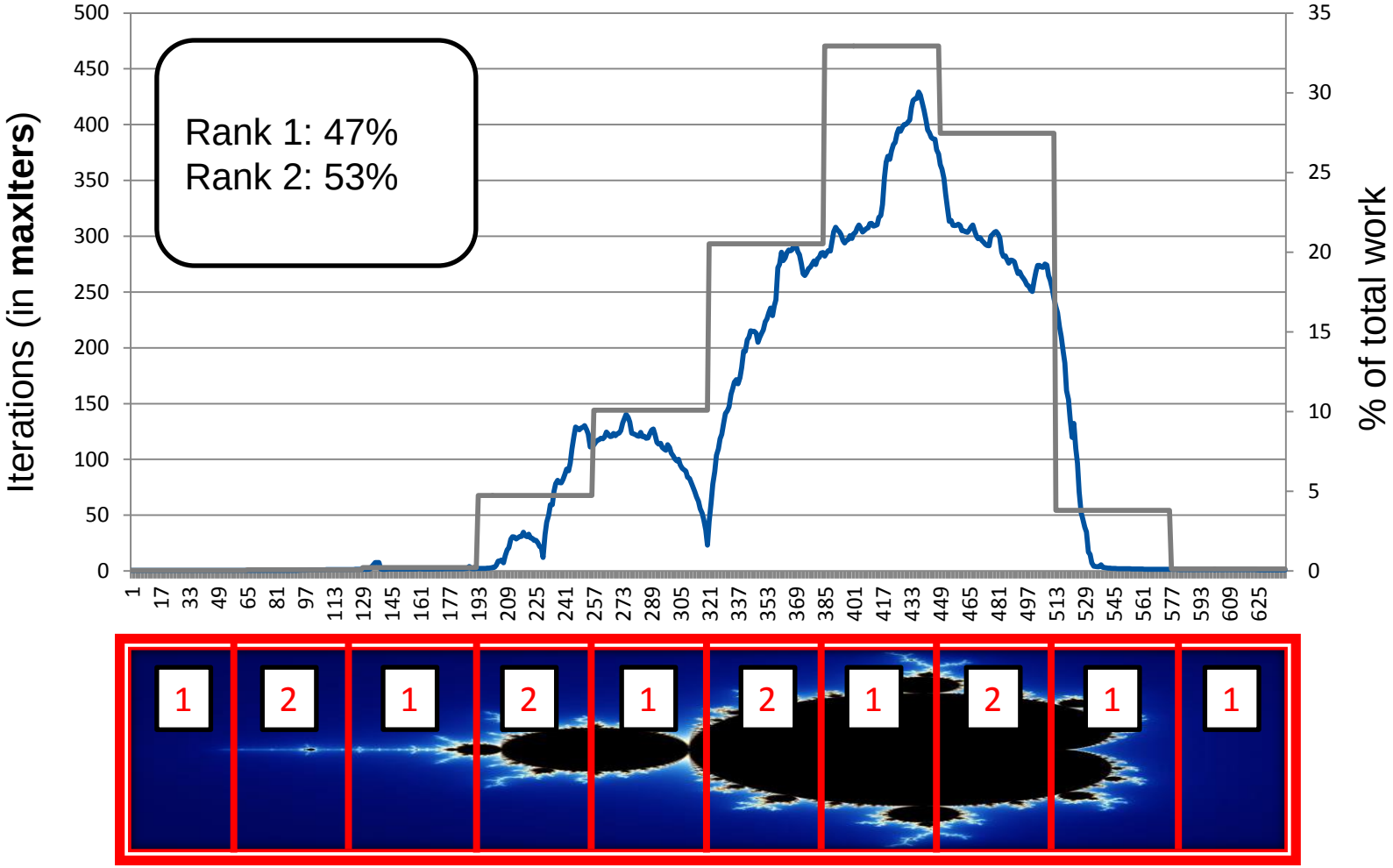
Work Balancing

- **Computational complexity mapped to ranks**



Work Balancing

- **Computational complexity mapped to ranks**



Work Balancing

- **Static work distribution:**
  - → Works best for regular problems
  - → Very simple to implement (e.g. nothing really to implement)
  - → Irregular problems result in work imbalance (e.g. Mandelbrot)
  - → Not usable for dynamic workspace problems (e.g. adaptive integration)

- **Controller – Worker approach:**
  - → Allows for great implementation flexibility
  - → Automatic work balancing if work units are properly sized
  - → Can be used for problems with dynamic workspaces
  - → Performs well on heterogeneous systems (e.g. CoWs)

Parallel Patterns

?

Thank you for your attention!