

Message Passing with MPI

Hristo Iliev

(Христо Илиев)

PPCES 2012, 2013

Christian Iwainsky

PPCES 2011

Agenda

■ Motivation

■ MPI – Part 1

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

■ MPI – Part 2

- Communicators
- Hybrid parallelization
- Common parallel patterns

■ Tools: Brief Introduction to VampirTrace and Vampir

Agenda

■ Motivation

■ MPI – Part 1

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

■ MPI – Part 2

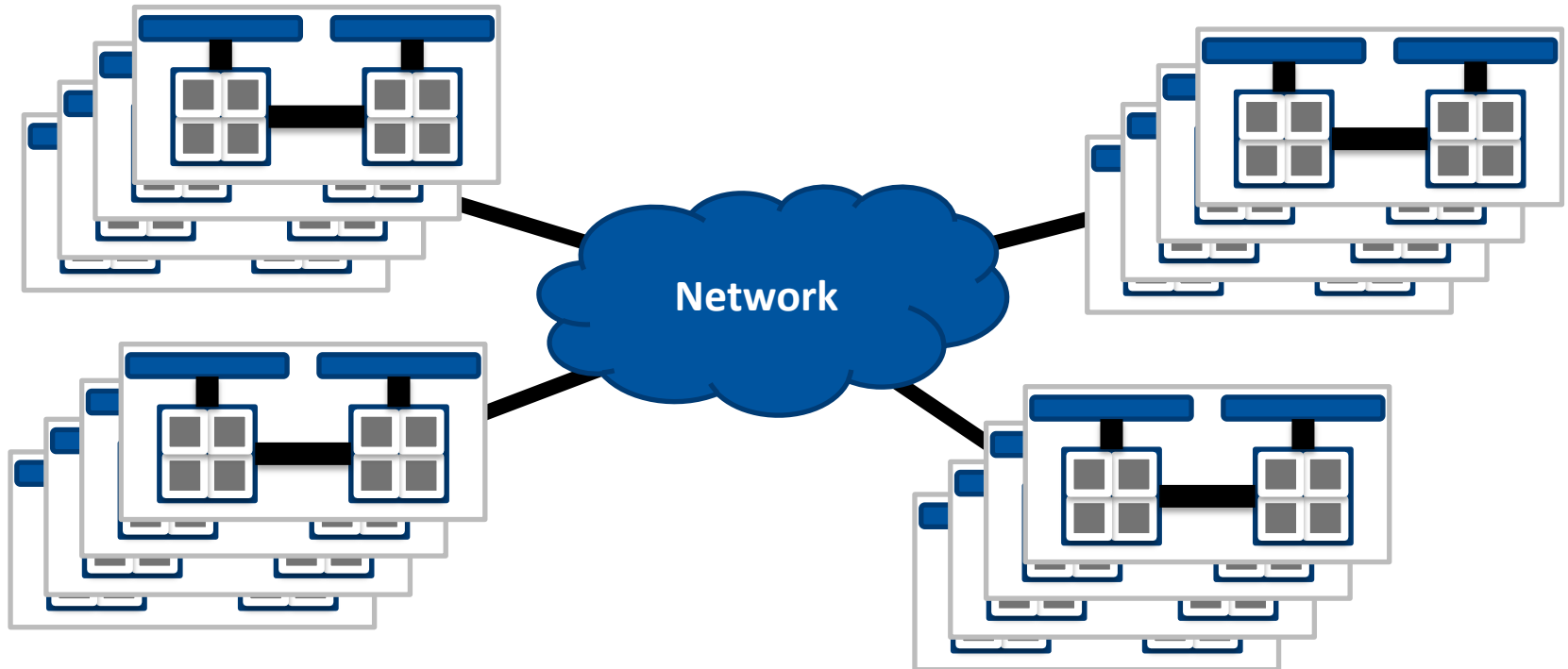
- Communicators
- Hybrid parallelization
- Common parallel patterns

■ Tools: Brief Introduction to VampirTrace and Vampir

Motivation: Distributed Memory Architecture

■ Clusters

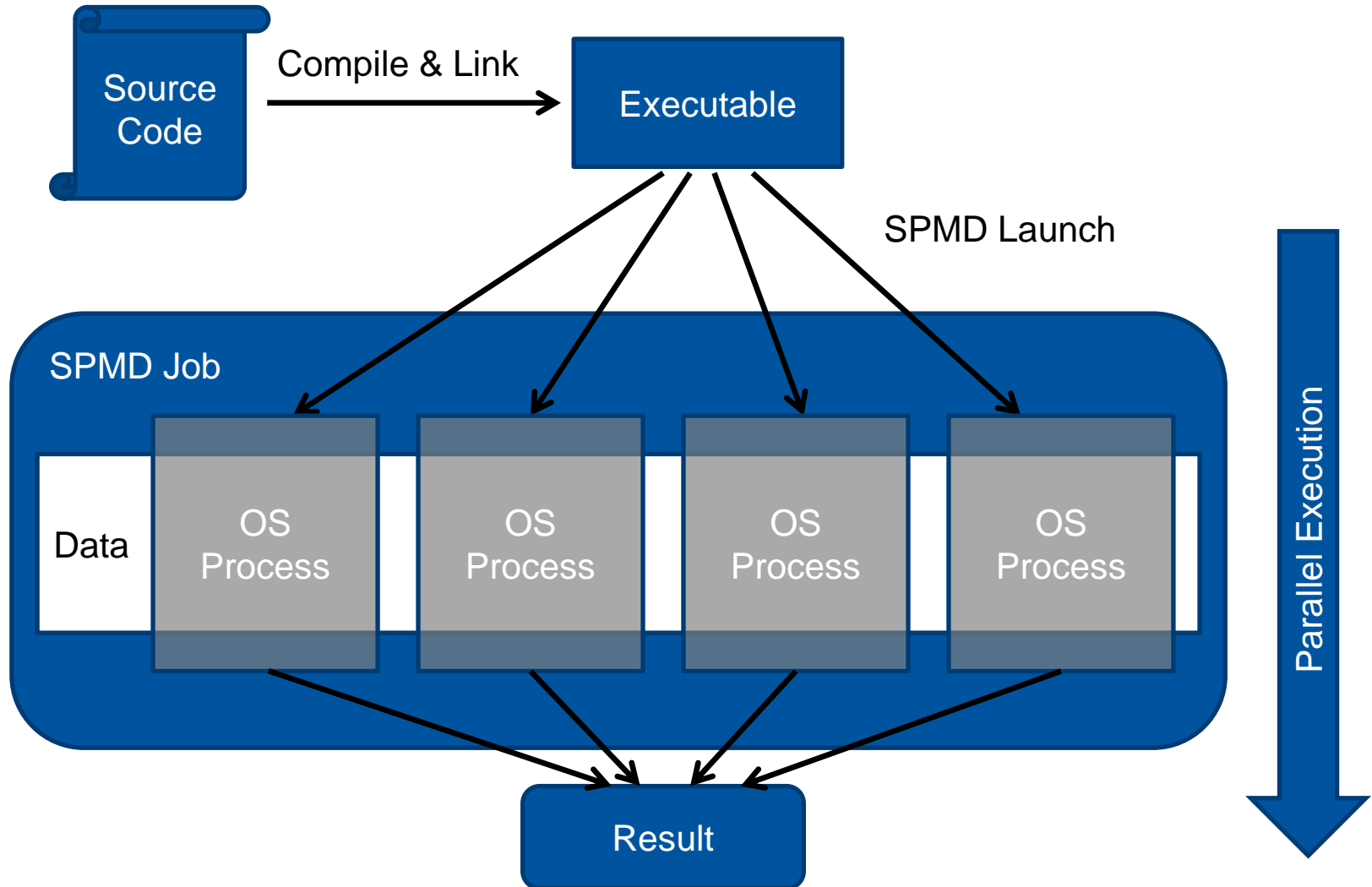
- HPC market is at large dominated by distributed memory *multicomputers* and *clusters*
- Nodes have no direct access to other nodes' memory and usually run their own (possibly stripped down) copy of the OS



- **Abstractions make programming and understanding easier**
 - MIMD, SIMD, SPMD, ...
- **Single Program Multiple Data (SPMD) model**
 - Multiple instances of a Single Program working on Multiple (parts of) Data
 - MPI is a concrete realization of the SPMD abstraction
 - Each program instance receives a unique ID – can be used for flow control

```
if (myID = specificID)  
{  
    do something  
}  
else  
{  
    do something different  
}
```

Motivation: SPMD Program Lifecycle



Motivation: 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

Motivation: Inter-Process Communication (IPC)

■ 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
 - **MPI**
 - BOINC (SETI@home, Einstein@home, *@home)
 - Globus Toolkit (Grid infrastructure)

Motivation:

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

- **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**

Agenda

- **Motivation**

- **MPI – Part 1**

- Concepts

- Point-to-point communication

- Non-blocking operations

- Collective operations

- **MPI – Part 2**

- Communicators

- Hybrid parallelization

- Common parallel patterns

- **Tools: Brief Introduction to VampirTrace and Vampir**

■ 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

■ 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

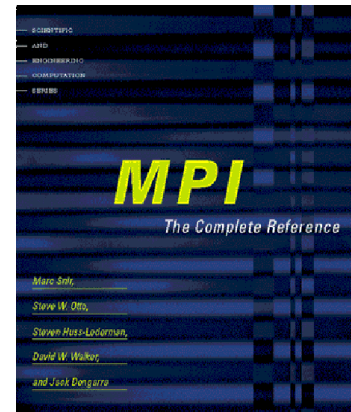
MPI History

- **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): Major enhancements**
 - One-sided communication
 - Parallel I/O
 - Dynamic process creation
 - Fortran 90 and C++ bindings
 - Thread safety
 - 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; C++ deleted from the standard

Literature about MPI (1/2)

■ 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

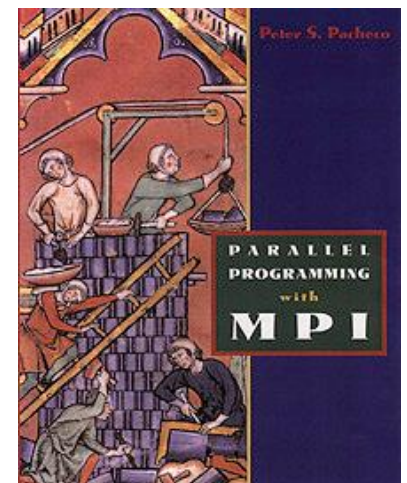
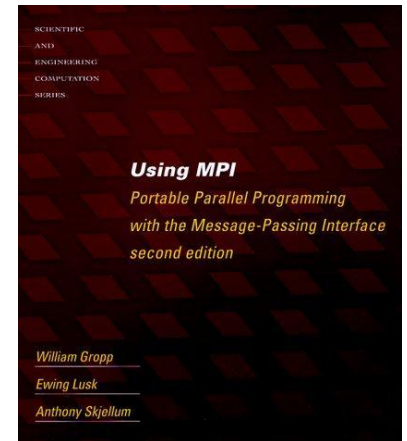
by William Gropp, Ewing Lusk, Rajeev Thakur

The MIT Press, Cambridge/London, 2000

■ Parallel Programming with MPI

by Peter Pacheco

Morgan Kaufmann Publishers, 1996



- **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 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

MPI Basics:

General MPI Program Structure

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

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

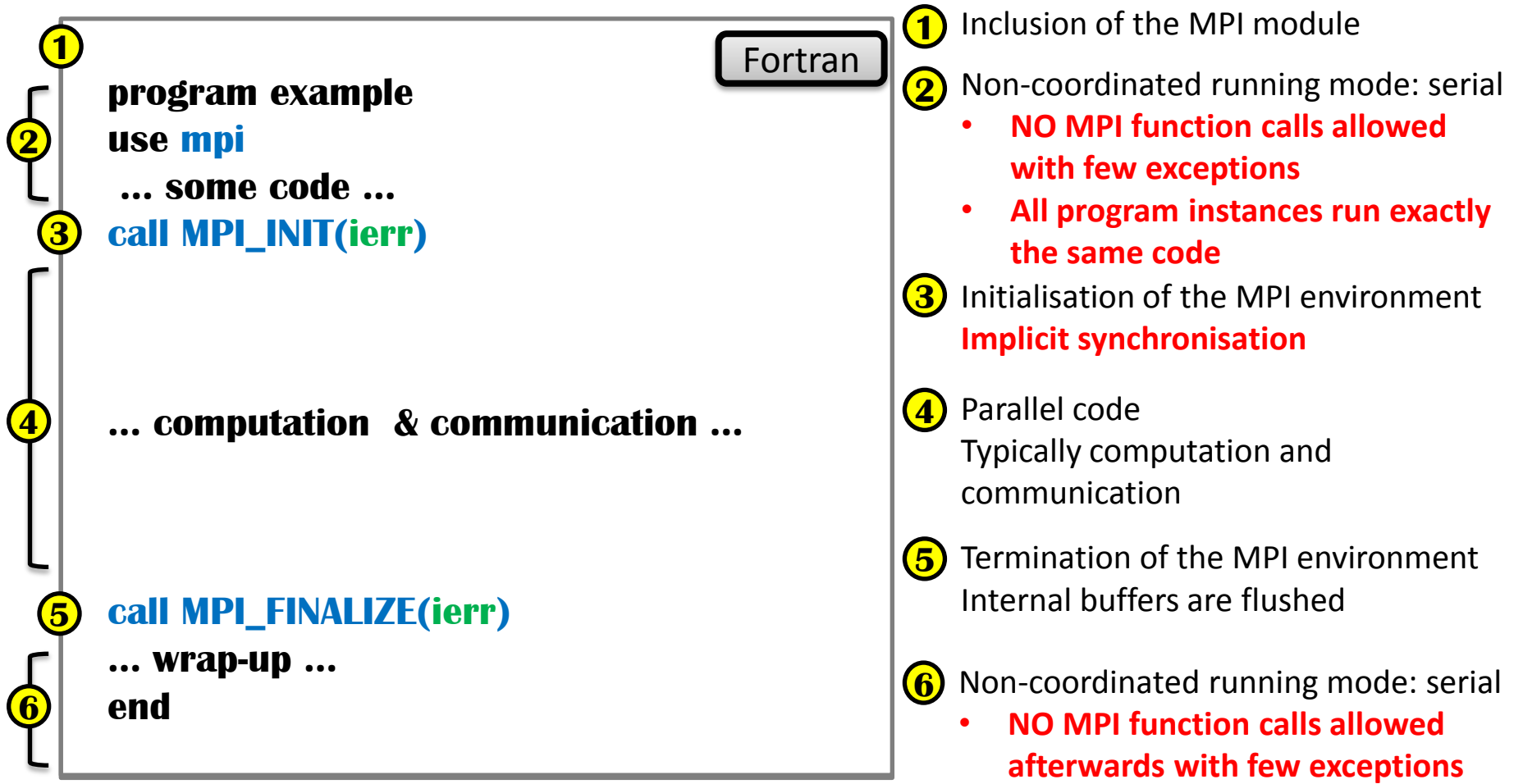
C/C++

- 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**

MPI Basics:

General MPI Program Structure

■ Startup, initialization, finalization and shutdown – Fortran



MPI Basics:

General MPI Program Structure

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

```
#include <mpi.h>
int main(int argc, char **argv)
{
    ... some code ...
    MPI_Init(&argc, &argv);
    ... other code ...
    MPI_Comm_size(MPI_COMM_WORLD,
                  &numberOfProcs);
    MPI_Comm_rank(MPI_COMM_WORLD,
                  &rank);
    ... computation & communication ...
    MPI_Finalize();
    ... wrapup ...
    return 0;
}
```

C/C++

- 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

MPI Basics:

General MPI Program Structure

- 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

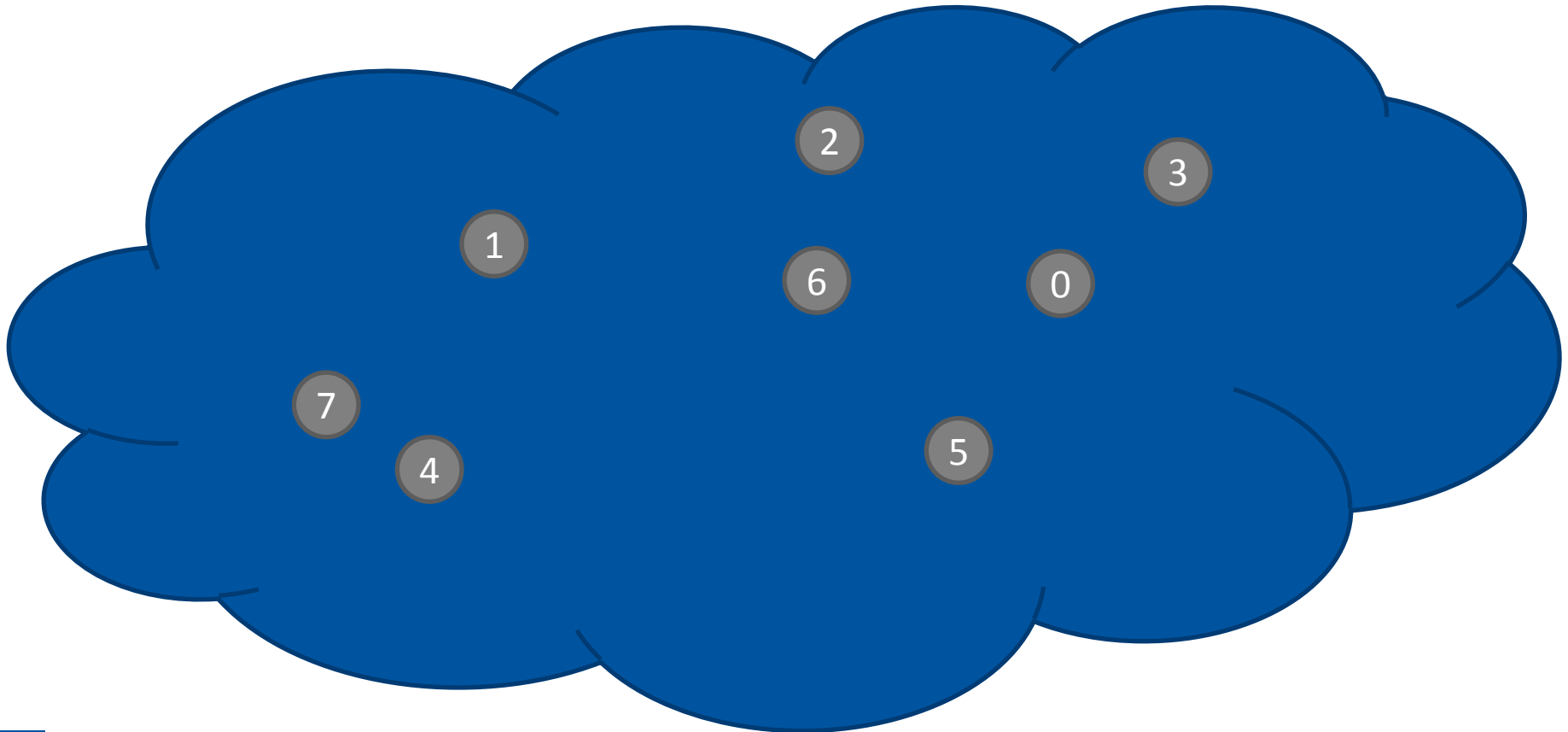
E.g. if there are 2 processes running then **numberOfProcs** will contain 2 after the call

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

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





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

■ MPI – Part 1

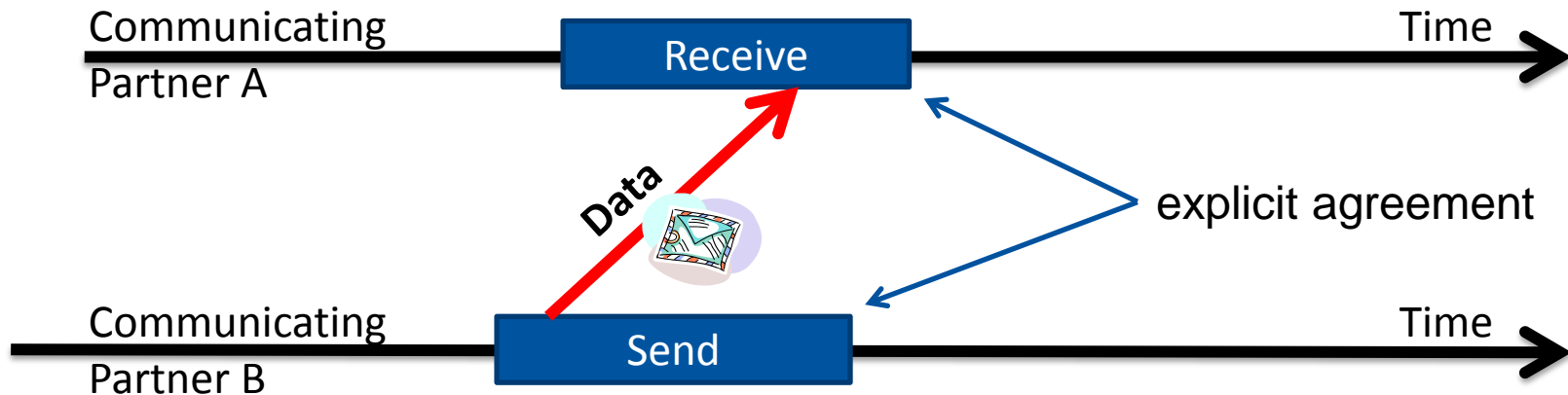
- Concepts
- Point-to-point communication
- Non-blocking operations
- Collective operations

■ MPI – Part 2

- Communicators
- Hybrid parallelization
- Common parallel patterns

■ Tools: Brief Introduction to VampirTrace and Vampir

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



- **Required:**

- Send and Receive primitives (operations)
- Identification of both the sender and the receiver
- Specification of what has to be send/received

Message Passing: Sending Data

- To send a single message:

What?

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

Where?

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 not less than 32767)
- **comm:** communication context (communicator)

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

Fortran

■ To receive a single message:

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

C/C++

- **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 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

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

- Fortran

- C/C++

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

- User-defined data types

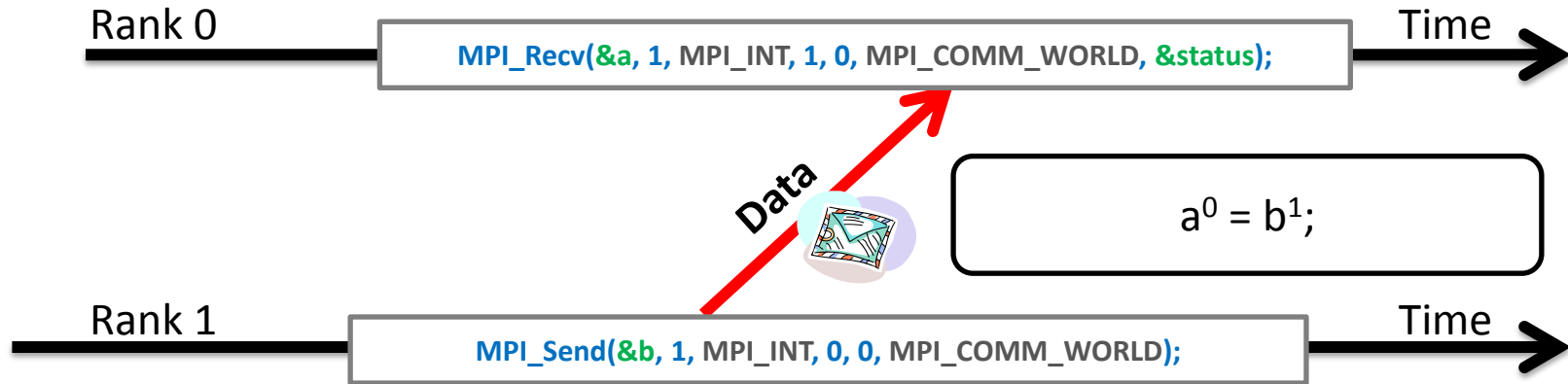
- **MPI is a library – it cannot infer on the type of elements in the supplied buffer at run time and that's why it has to be told what it is**
 - C++ wrapper APIs usually infer the MPI datatype from the arguments type
- **MPI datatypes tell MPI how to:**
 - read actual memory values from the send buffer
 - write actual memory values in 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 Basics:

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 (always last in the argument list) 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 in MPI is explicit:



- These two calls transfer the value of the *b* variable in rank 1 into the *a* variable in rank 0.
- For now assume that *comm* is fixed as `MPI_COMM_WORLD`. We will talk about communicators later on.

■ 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?

Message Passing: A Full Example

C/C++

```
#include <mpi.h>
int main(int argc, char **argv)
{
    int numProcs, rank, data;
    MPI_Status status;
    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD,
                  &numProcs);
    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

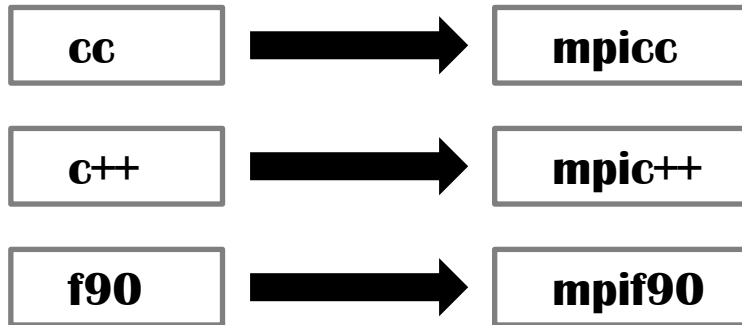
③

④

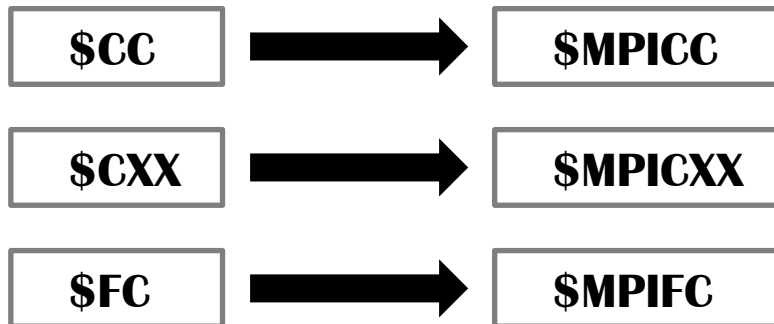
⑤

- ① Initialize the MPI library
- ② Get entity identification
- ③ Do smth different in each/a rank
- ④ Communicate
- ⑤ Clean up the MPI library

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



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



- MPI is a typical library – header files, object code archives, etc.
- For convenience vendors provide specialized 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
-lidl
-Wl,--export-dynamic
-lnsi
-lutil
```

■ RWTH Aachen

Compiler module):

- 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> ...
```


■ 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 Passing: Message Envelope and Matching

- Reception of MPI messages is done by matching their envelope
- Recall: MPI_Send

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

C/C++

- 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

- Recall: MPI_Recv

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

C/C++

Message Passing: 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)
```

C/C++

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

C/C++

- 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 w.r.t. the message**

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

- **The receive buffer must be able to hold the whole message**
 - send count \leq 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**

- **The receive buffer must be able to hold the whole message**
 - send count \leq 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

■ Blocks until a matching message becomes available

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

C/C++

→ 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)

C/C++

■ Status inquiry:

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

C/C++

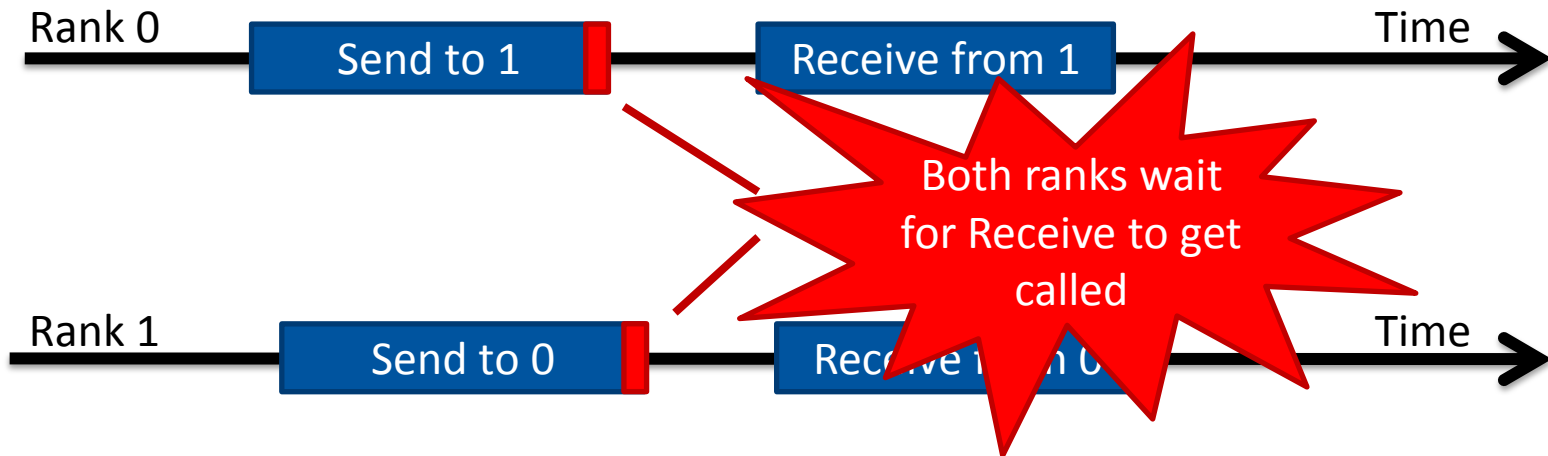
→ Computes how many integral **datatype** elements can be extracted from the message referenced by **status**

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

■ Both MPI_Send and MPI_Recv calls are blocking:

- Blocking calls only return once the operation has completed
- The receive operation only returns after a matching message has arrived
- The send operation might be buffered and return before the message is sent

■ Deadlock in a typical data exchange scenario:



Message Passing: 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)
```

C/C++

- 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

Message Passing: 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)
```

C/C++

- First sends one message and then receives one message.
- **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)
```

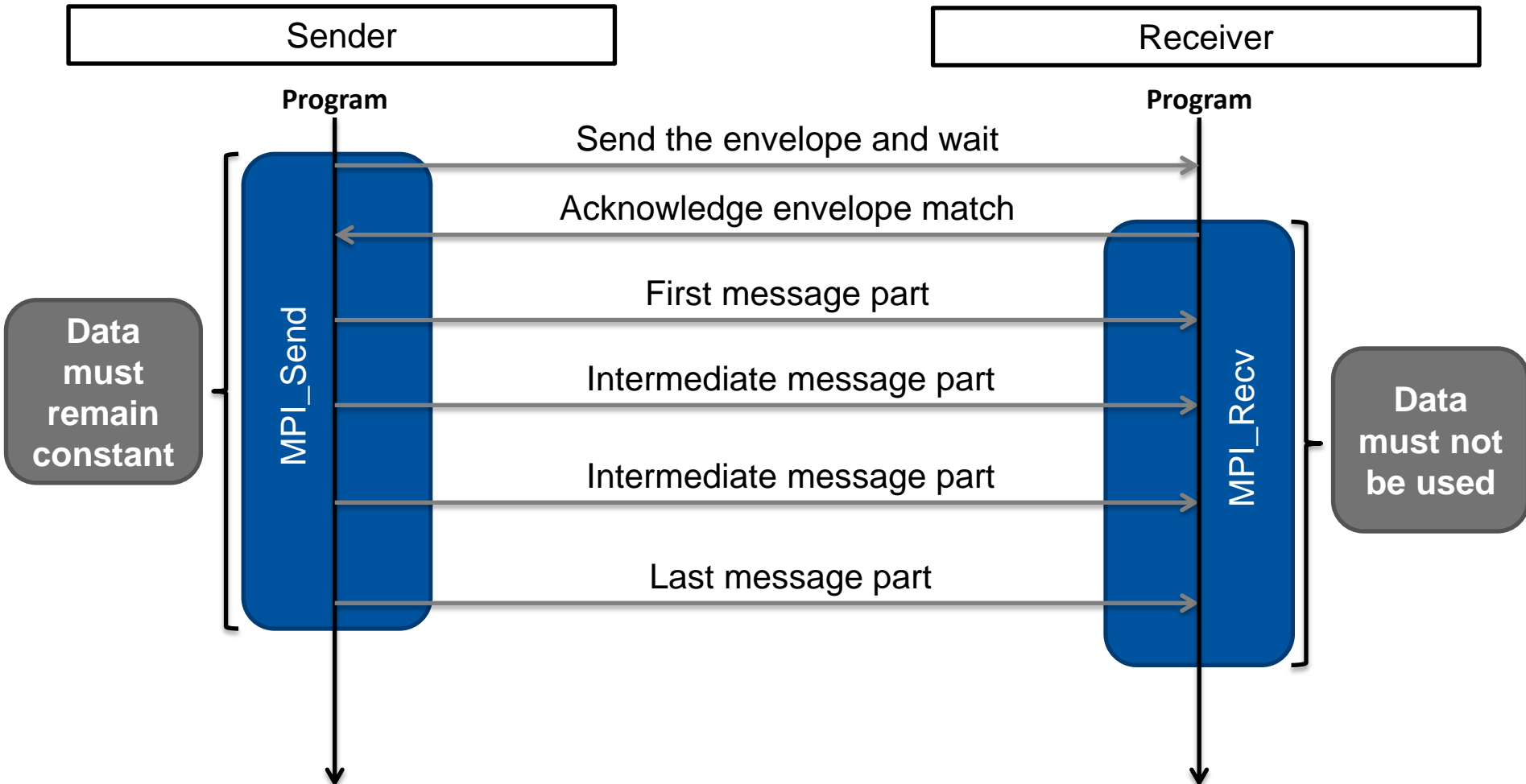
C/C++

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

Agenda

- Motivation
- **MPI – Part 1**
 - Concepts
 - Point-to-point communication
 - Non-blocking operations
 - Collective operations
- **MPI – Part 2**
 - Communicators
 - Hybrid parallelization
 - Common parallel patterns
- **Tools: Brief Introduction to VampirTrace and Vampir**

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



■ 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)**

C/C++

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

C/C++

→ **request:** on success set to the handle of the initiated operation

■ Blocking wait for the completion of an asynchronous operation:

MPI_Wait (MPI_Request *request, MPI_Status *status)

C/C++

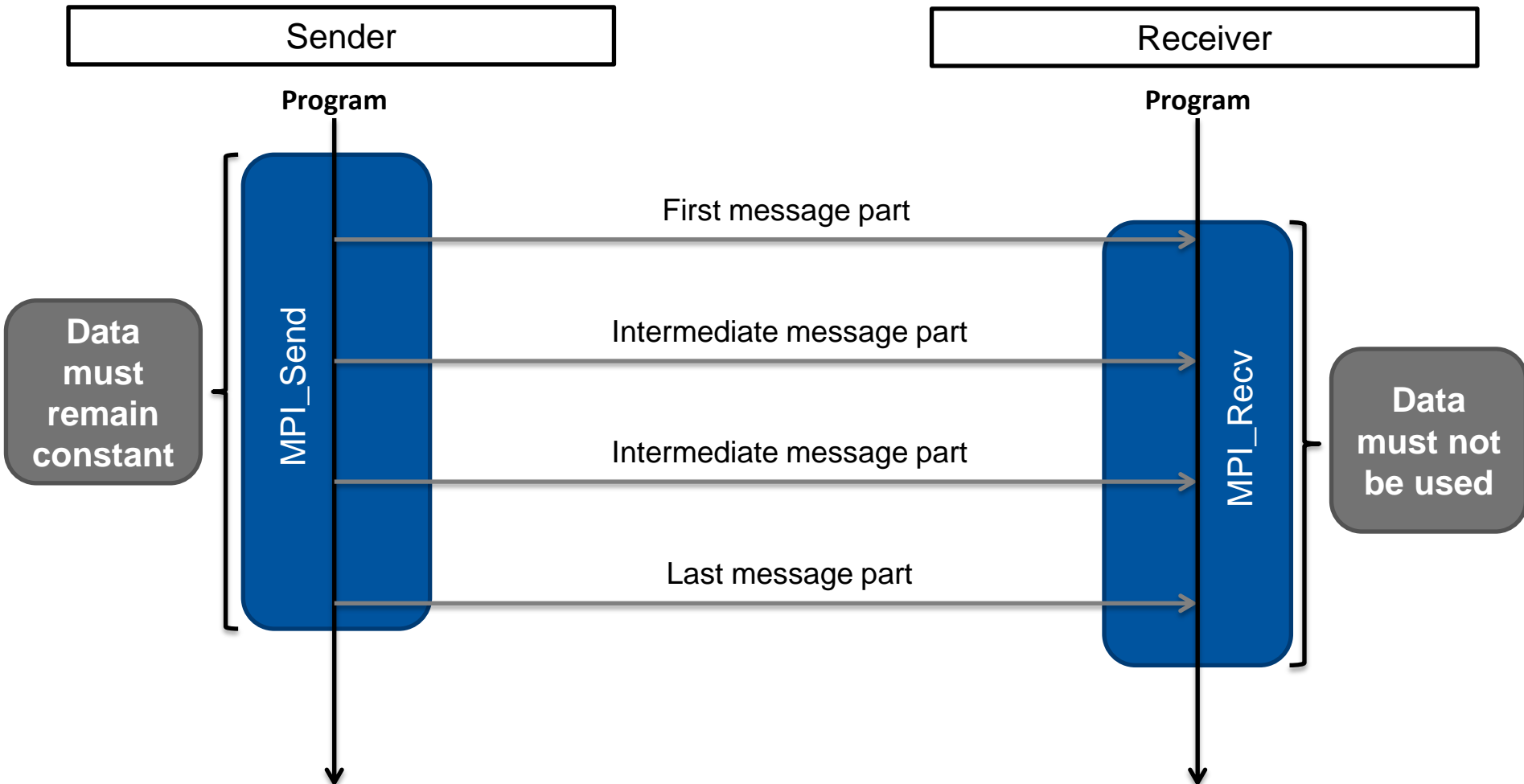
→ **request:** handle for an active asynchronous operation
freed and set to **MPI_REQUEST_NULL** on successful return

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

- **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 testing and waiting MPI calls.**
- **Blocking MPI calls are equivalent to initiating 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.**

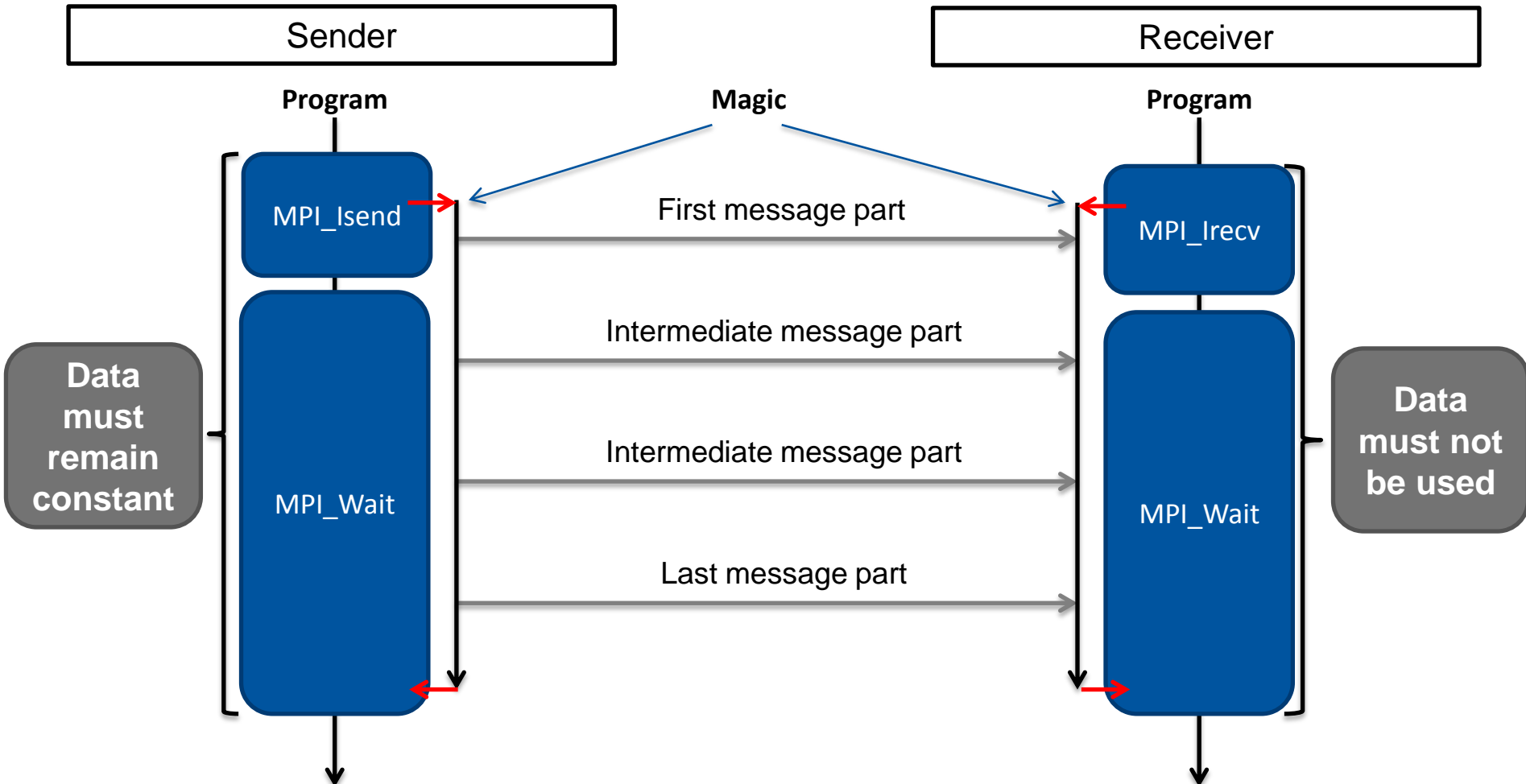
Non-Blocking Operations: Example

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



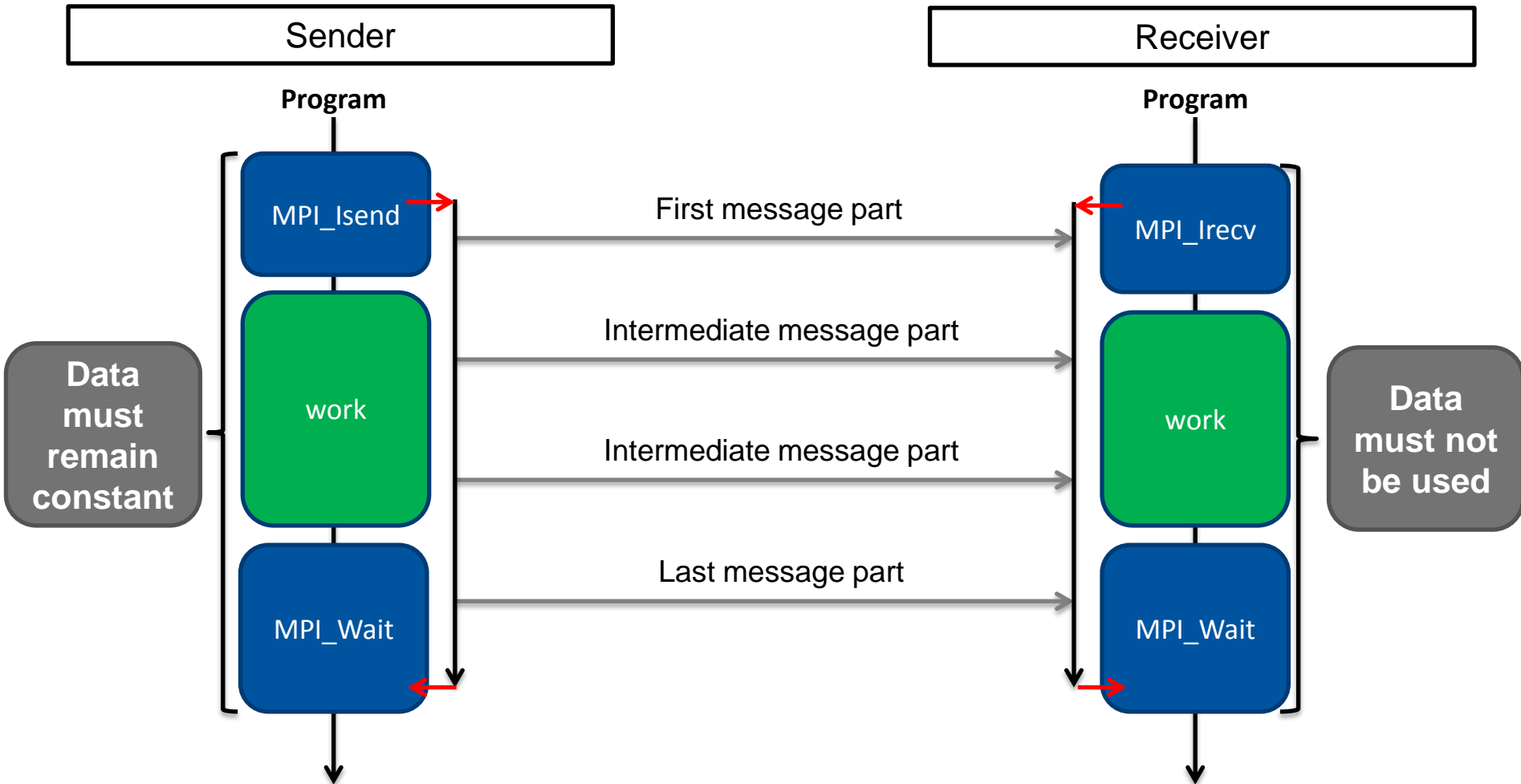
Non-Blocking Operations: Example

- Equivalent with non-blocking send and receive operations:



Non-Blocking Operations: Interleave Computation/Communication

- Other work can be done in between:



Non-Blocking Operations: Non-Blocking Request Testing

■ Test if given operation has completed:

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

```
C/C++
```

- **flag**: **true** once the operation has completed, otherwise **false**
- **status**: status of the operation (if completed), only set if **flag** is **true**
- 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**

Non-Blocking Operations: 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

■ **If not interested in the statuses, pass MPI_STATUSES_IGNORE.**

- **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

■ Standard mode

→ The call blocks until the message has either been transferred or 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 only if a matching receive has been posted already. Behaves as standard send in every other aspect.

■ 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**)

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

MPI_Abort (MPI_Comm comm, int errorcode)

C/C++

→ **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 ()

C/C++

→ 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)

C/C++

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

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

MPI Basics:

MPI Lifetime Management Functions

- MPI can only be initialized once and finalized once 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)`

C/C++

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

- Test if MPI is already finalized:

`MPI_Finalized (int *flag)`

C/C++

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

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



Agenda

■ Motivation

■ MPI – Part 1

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

■ MPI – Part 2

- Communicators
- Hybrid parallelization
- Common parallel patterns

■ Tools: Brief Introduction to VampirTrace and Vampir

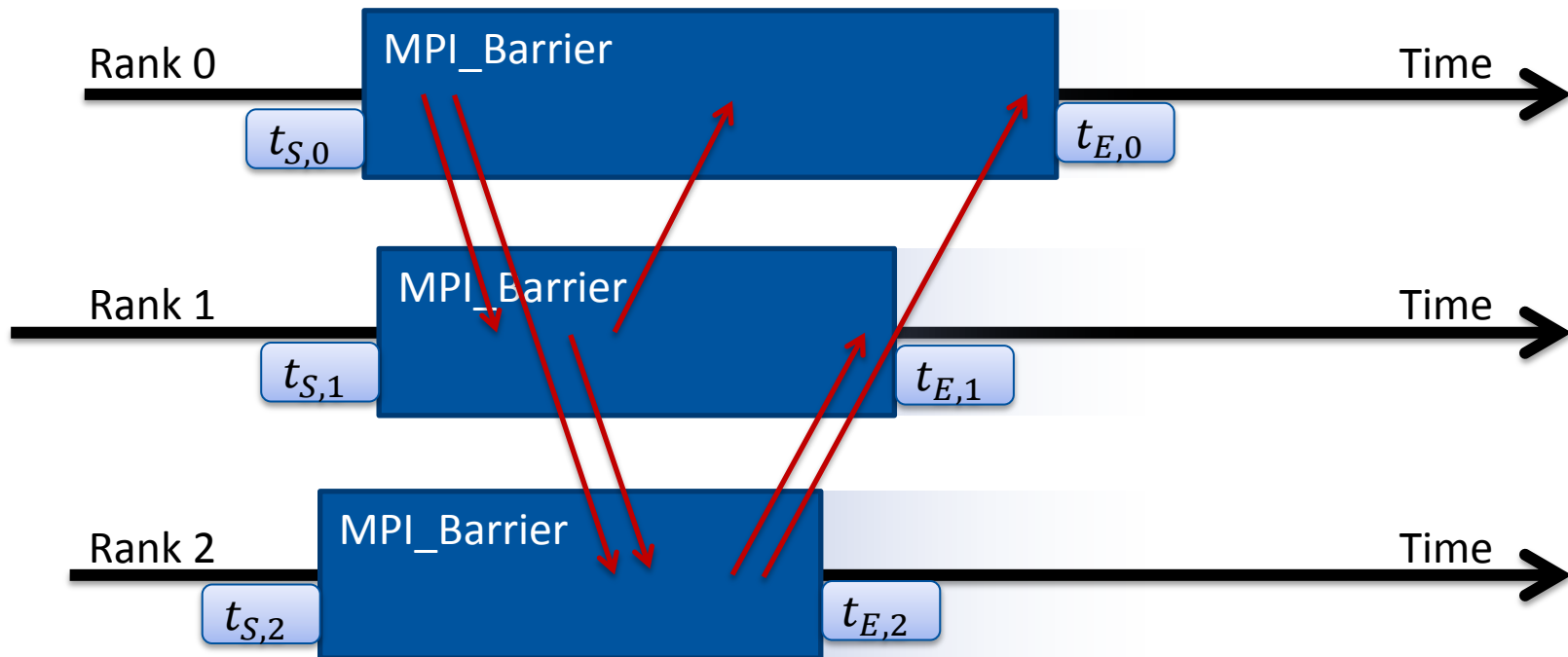
- **MPI collective operations involve all ranks in a given communication context (communicator) at the same time.**
- **All ranks in the communicator must make the same MPI call for the operation to succeed. There should be one call per MPI rank (e.g. not per thread).**
- **Some collective operations 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.

Collective Operations: Barrier Synchronization

- The only synchronization primitive that MPI provides

`MPI_Barrier (MPI_Comm comm)`

C/C++



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

- 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);
    }
}
```

■ MPI broadcast operation:

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

C/C++

- **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:

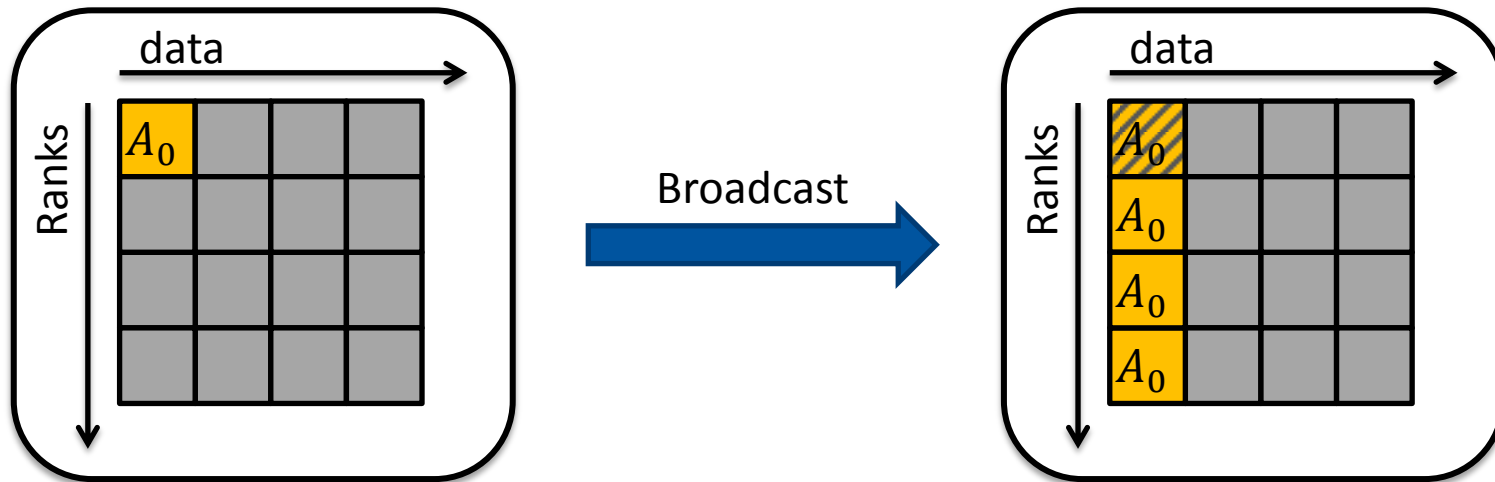
- in all ranks but **root**, **data** is an output argument
- in rank **root**, **data** is an input argument
- **MPI_Bcast** completes only after all ranks in **comm** have made the call

Collective Operations: Data Broadcast

■ MPI broadcast operation:

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

C/C++



- 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)
```

C/C++

- **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

Significant at root
rank only

- 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)
```

C/C++

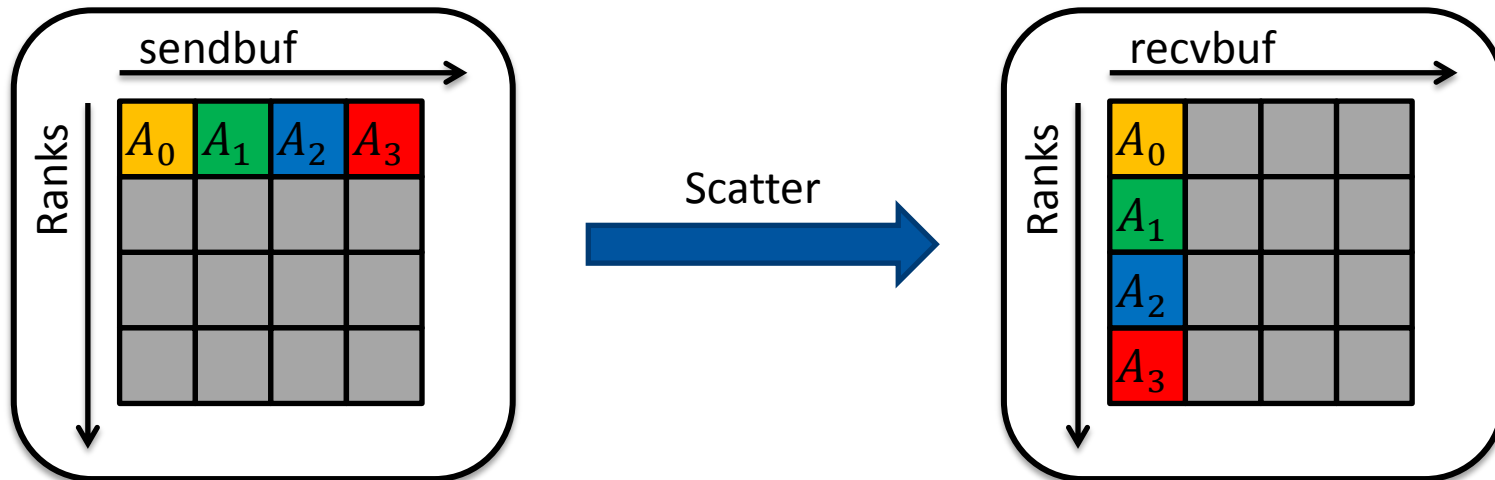
- Notes:

- **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

- 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**)

C/C++



- 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)
```

C/C++

- The opposite operation of MPI_Scatter:

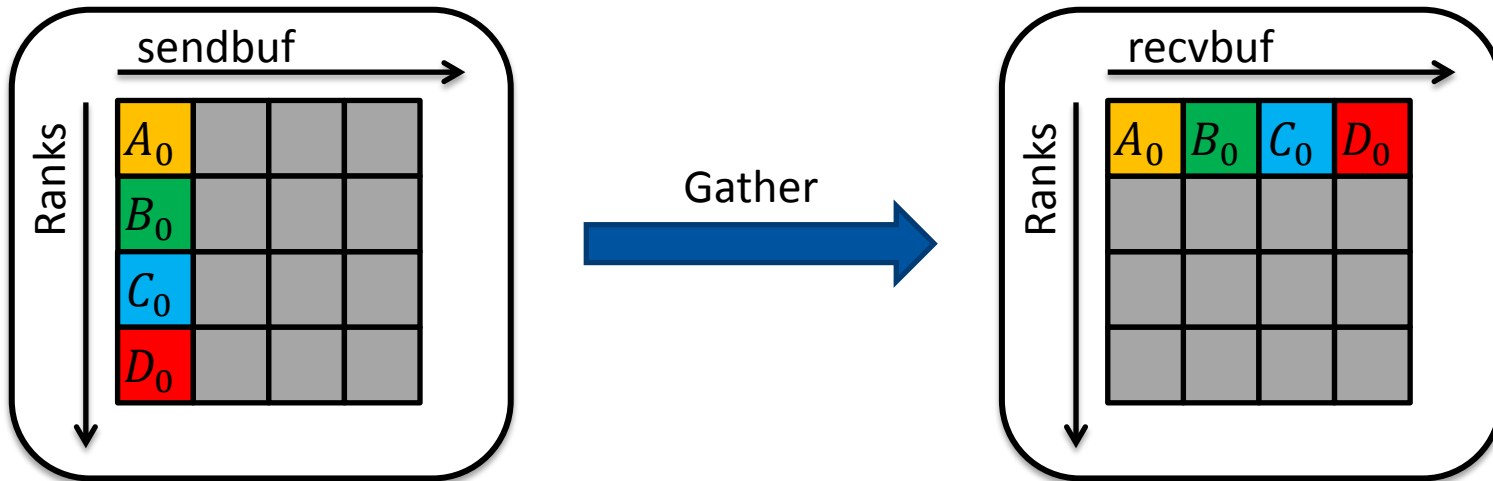
- **recvbuf** must be large enough to hold **recvcount** elements from each rank
- root also receives one data chunk from itself
- data chunks are stored in increasing order of receiver's rank
- for each chunk the receive size must match the amount of data sent

Significant at root
rank only

- 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)
```

C/C++



- 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 recvtpe, MPI_Comm comm)
```

C/C++

- Note:

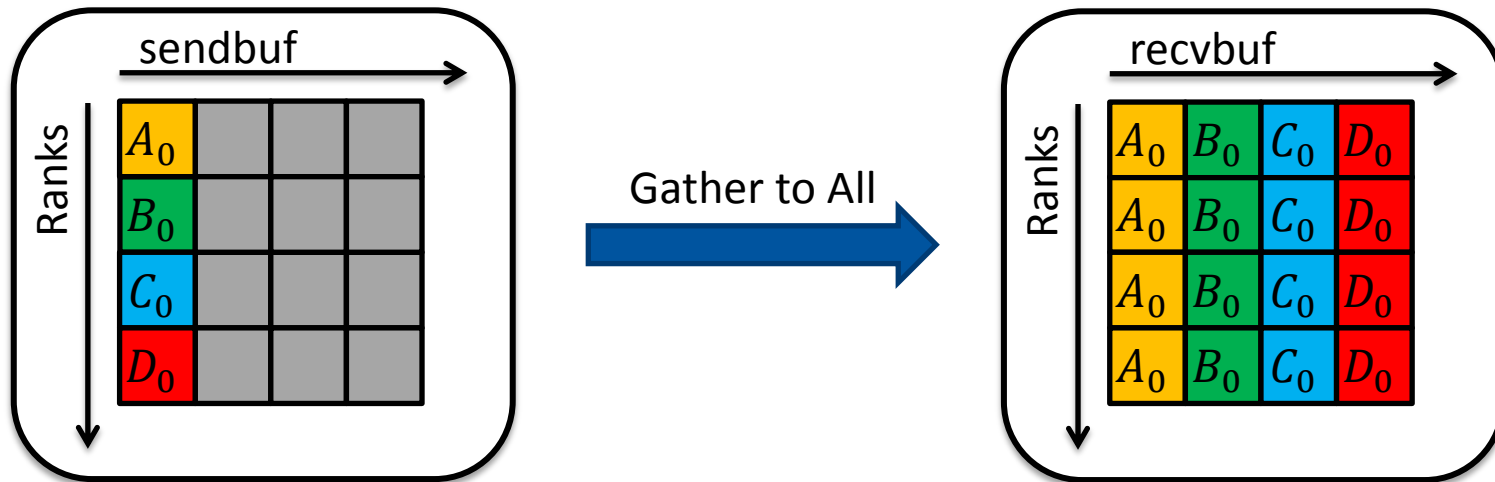
- rootless operation – all ranks receive a copy of the gathered data
- each rank also receives one data chunk from itself
- data chunks are stored in increasing order of sender's rank
- for each chunk the receive size must match the amount of data sent
- equivalent to **MPI_Gather + MPI_Bcast**, but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

Collective Operations: 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**)

C/C++



■ Combined scatter and gather operation:

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

C/C++

■ 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)

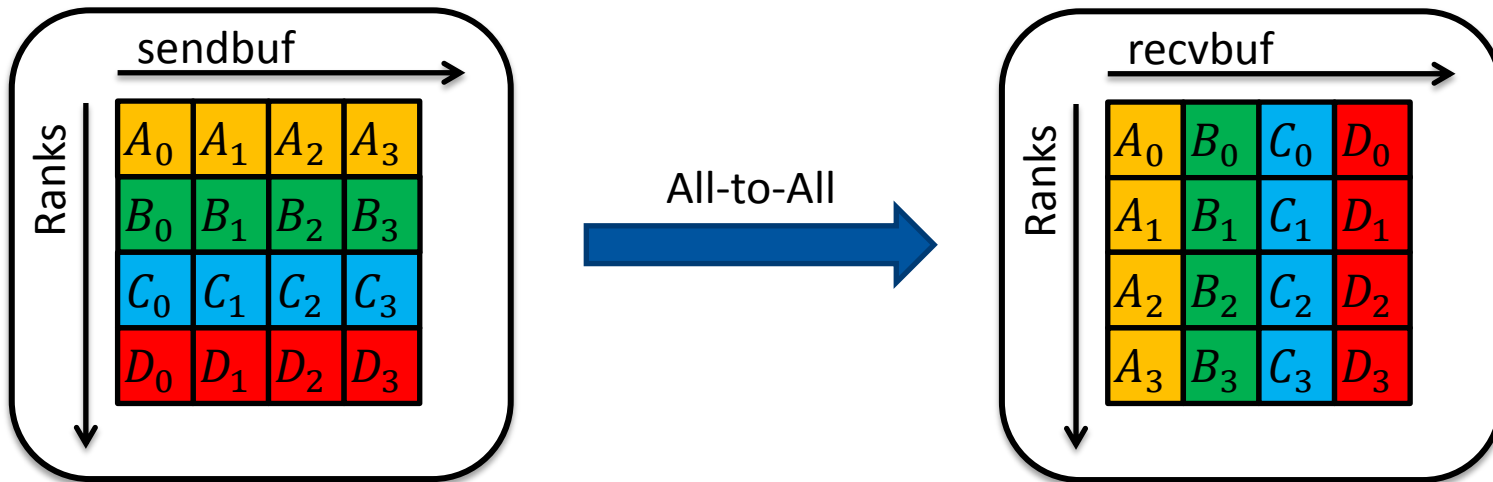
Collective Operations: All-to-All

■ Combined scatter and gather operation:

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

C/C++

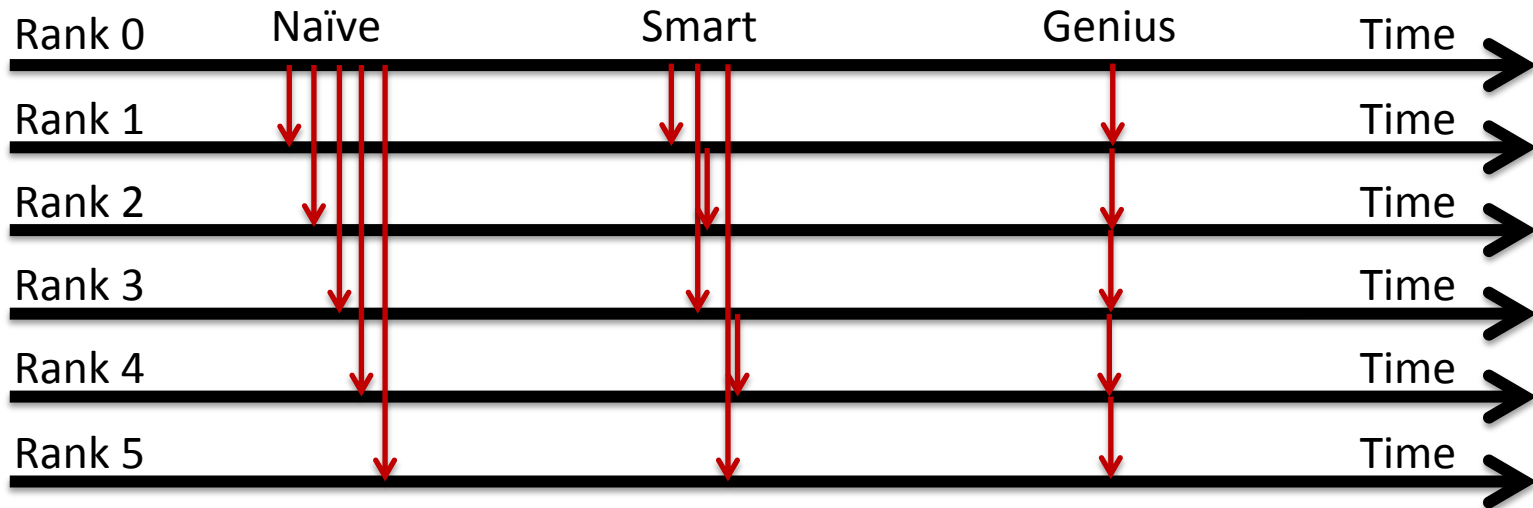
■ Note: a kind of global chunked transpose



- **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**

- Collective operations implement portably common SPMD patterns
- Implementation-specific magic, but standard behavior
- Example: Broadcast

- Naïve: root sends separate message to every other rank, $O(\#\text{ranks})$
- Smart: tree-based hierarchical communication, $O(\log(\#\text{ranks}))$
- Genius: pipeline-based segmented transport, $O(1)$



- 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)**

C/C++

- **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

$$\text{recvbuf}[i] = \text{sendbuf}_0[i] \text{ op } \text{sendbuf}_1[i] \text{ op } \dots \text{ op } \text{sendbuf}_{\text{n ranks}-1}[i]$$

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

- **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.**

- 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)
```

C/C++

- 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**)

■ Versions with variable chunk size:

→ **MPI_Gatherv**

→ **MPI_Scatterv**

→ **MPI_Allgatherv**

→ **MPI_Alltoallv**

■ Advanced reduction operations:

→ **MPI_Scan**

→ **MPI_Reduce_scatter**

→ **MPI_Reduce_scatter_block**

→ ...

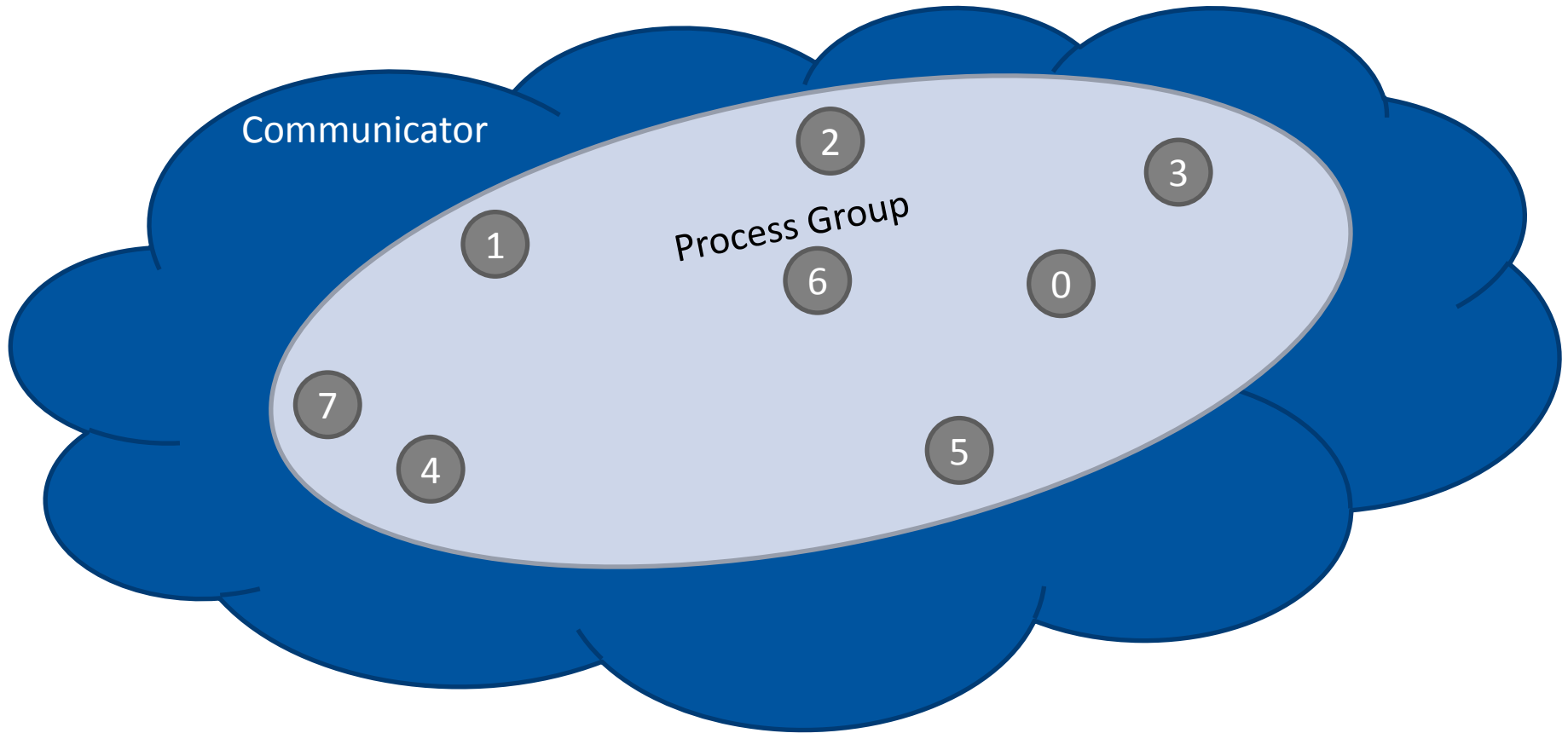


Agenda

- Motivation
- MPI – Part 1
 - Concepts
 - Point-to-point communication
 - Non-blocking operations
 - Collective operations
- **MPI – Part 2**
 - Communicators
 - Hybrid parallelization
 - Common parallel patterns
- **Tools: Brief Introduction to VampirTrace and Vampir**

- **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.**

- Communicator – process group – ranks



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

MPI_Comm_size (MPI_Comm comm, int *size)

C/C++

→ 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)

C/C++

- 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

■ Recall: message envelope

	Sender	Receiver
Source	Implicit	Explicit, wildcard possible (<code>MPI_ANY_SOURCE</code>)
Destination	Explicit	Implicit
Tag	Explicit	Explicit, wildcard possible (<code>MPI_ANY_TAG</code>)
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

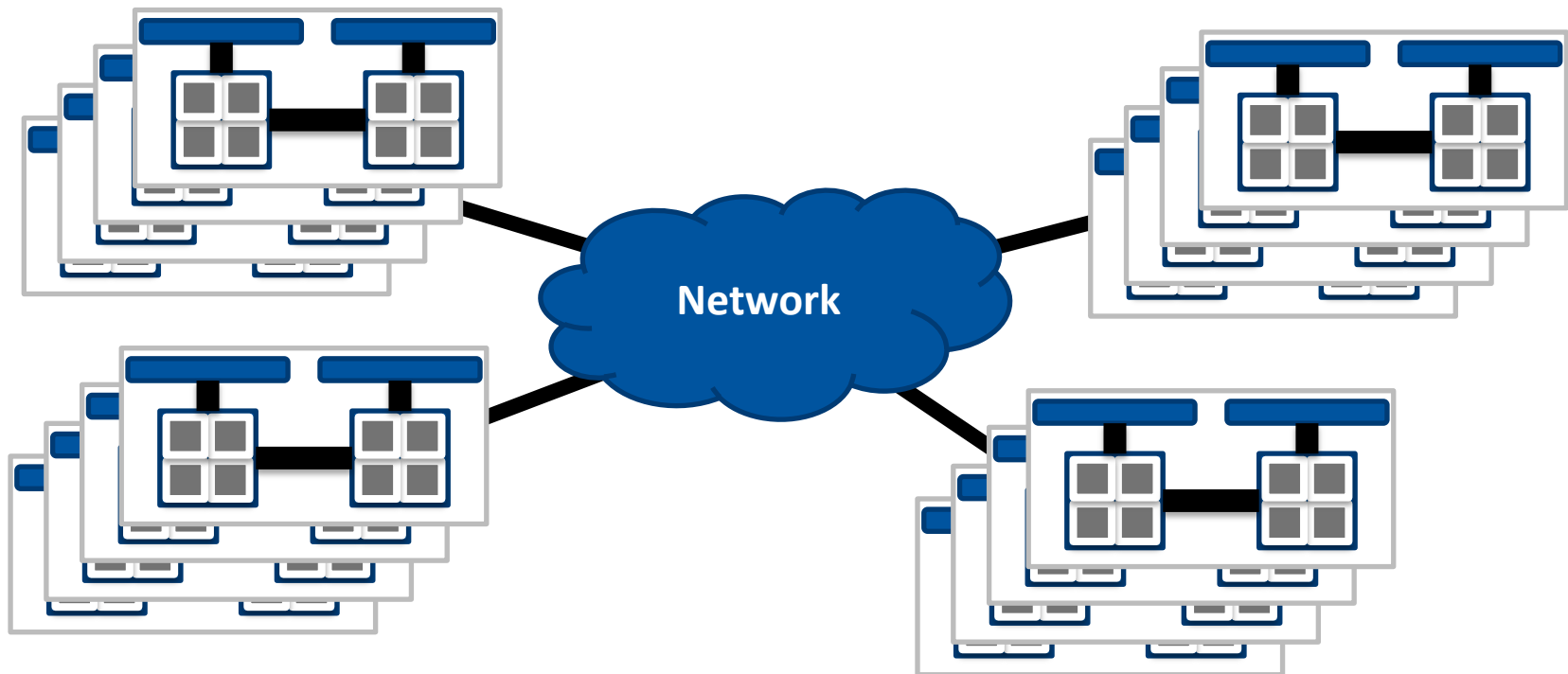
Agenda

- Motivation
- MPI – Part 1
 - Concepts
 - Point-to-point communication
 - Non-blocking operations
 - Collective operations
- **MPI – Part 2**
 - Communicators
 - Hybrid parallelization
 - Common parallel patterns
- **Tools: Brief Introduction to VampirTrace and Vampir**

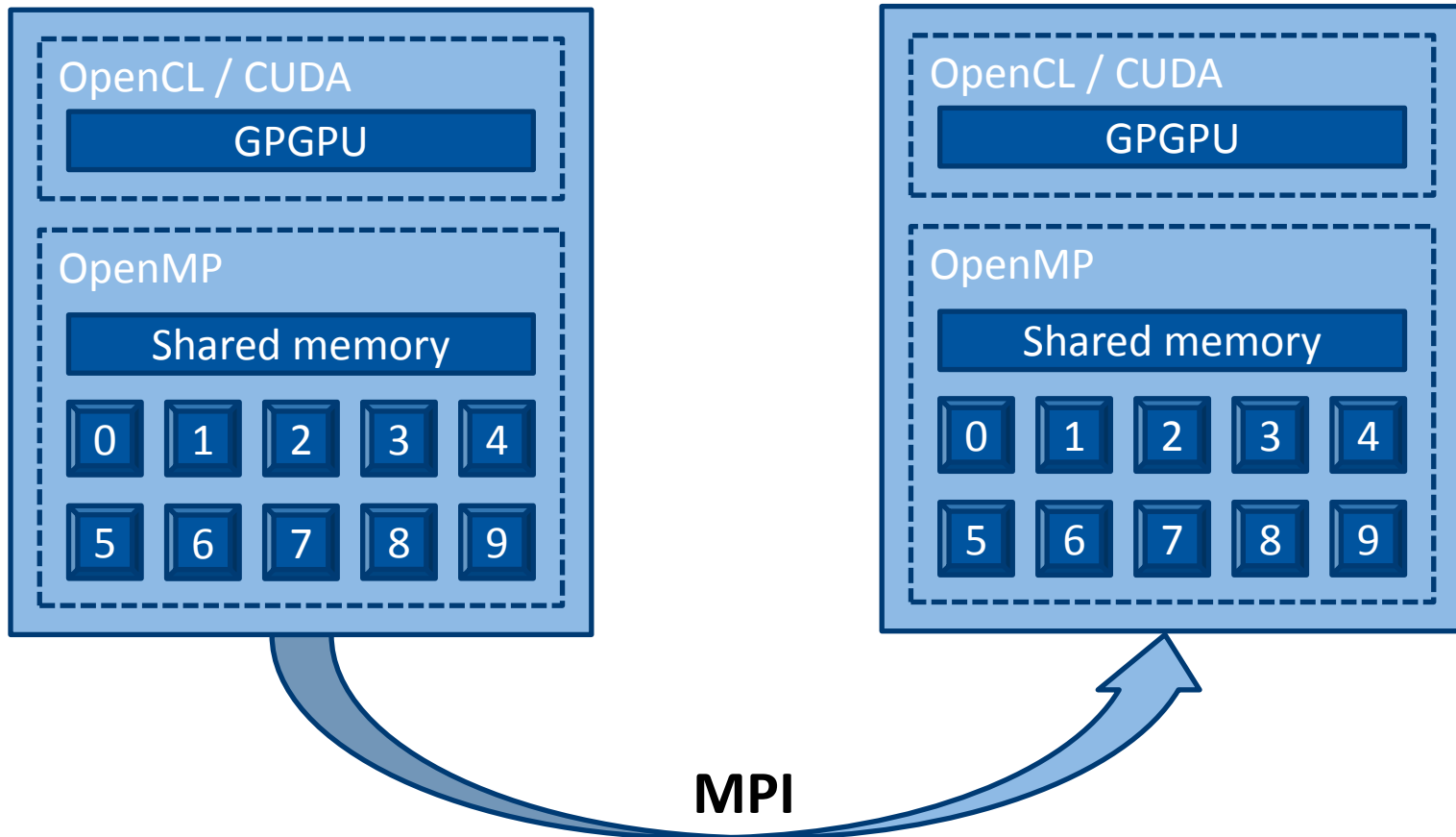
- **MPI is sufficiently abstract so it runs perfectly fine on a single node:**
 - it doesn't care where processes are located as long as they can communicate
 - message passing implemented using shared memory and IPC
 - all details hidden by the MPI implementation;
 - usually faster than sending messages 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

■ 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



- Hierarchical mixing of different programming paradigms



- Most MPI implementations are threaded (e.g. for non-blocking requests) but not thread-safe.
- Four levels of threading support in increasing order:

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

- All implementations support `MPI_THREAD_SINGLE`, but some do not support `MPI_THREAD_MULTIPLE`.

■ Initialize MPI with thread support:

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

C/C++

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

Fortran

→ **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.

■ Obtain the provided level of thread support:

MPI_Query_thread (int *provided)

C/C++

- 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)

C/C++

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

■ 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/1.6.4mt**

→ The InfiniBand BTL does not support **MPI_THREAD_MULTIPLE!!!**

■ Intel MPI

→ Compile with **-openmp**, **-parallel** or **-mt_mpi**

- **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.**

Now What?

- **Here is a list of important MPI topic that this course does not cover:**

- User-defined MPI datatypes
- Virtual topologies
- Parallel I/O

- **And a list of more exotic topics:**

- Dynamic process control
- Client/server relations
- One-sided operations



Agenda

- **Motivation**

- **MPI – Part 1**

- Concepts

- Point-to-point communication

- Non-blocking operations

- Collective operations

- **MPI – Part 2**

- Communicators

- Hybrid parallelization

- Common parallel patterns

- **Tools: Brief Introduction to VampirTrace and Vampir**

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

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

■ 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: Parallel Computing – 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 $E_n > 1$ – superlinear speedup

■ Communication overhead increases with the number of processes:

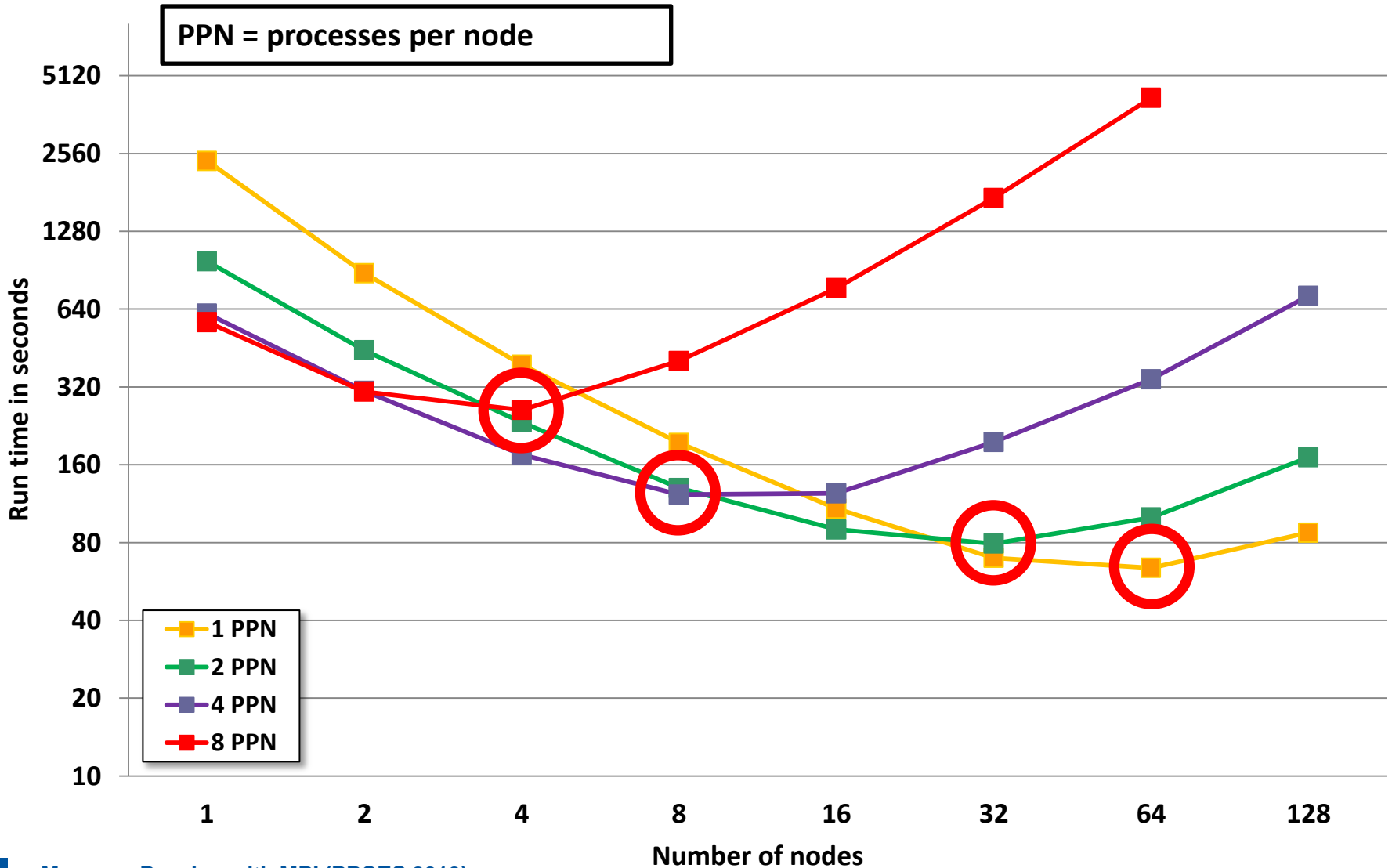
→ more processes → more messages (depends on the communication pattern)
especially true for the collective operations
(*you did not reimplement them, did you?*)

→ more messages → more network latency → more serial time

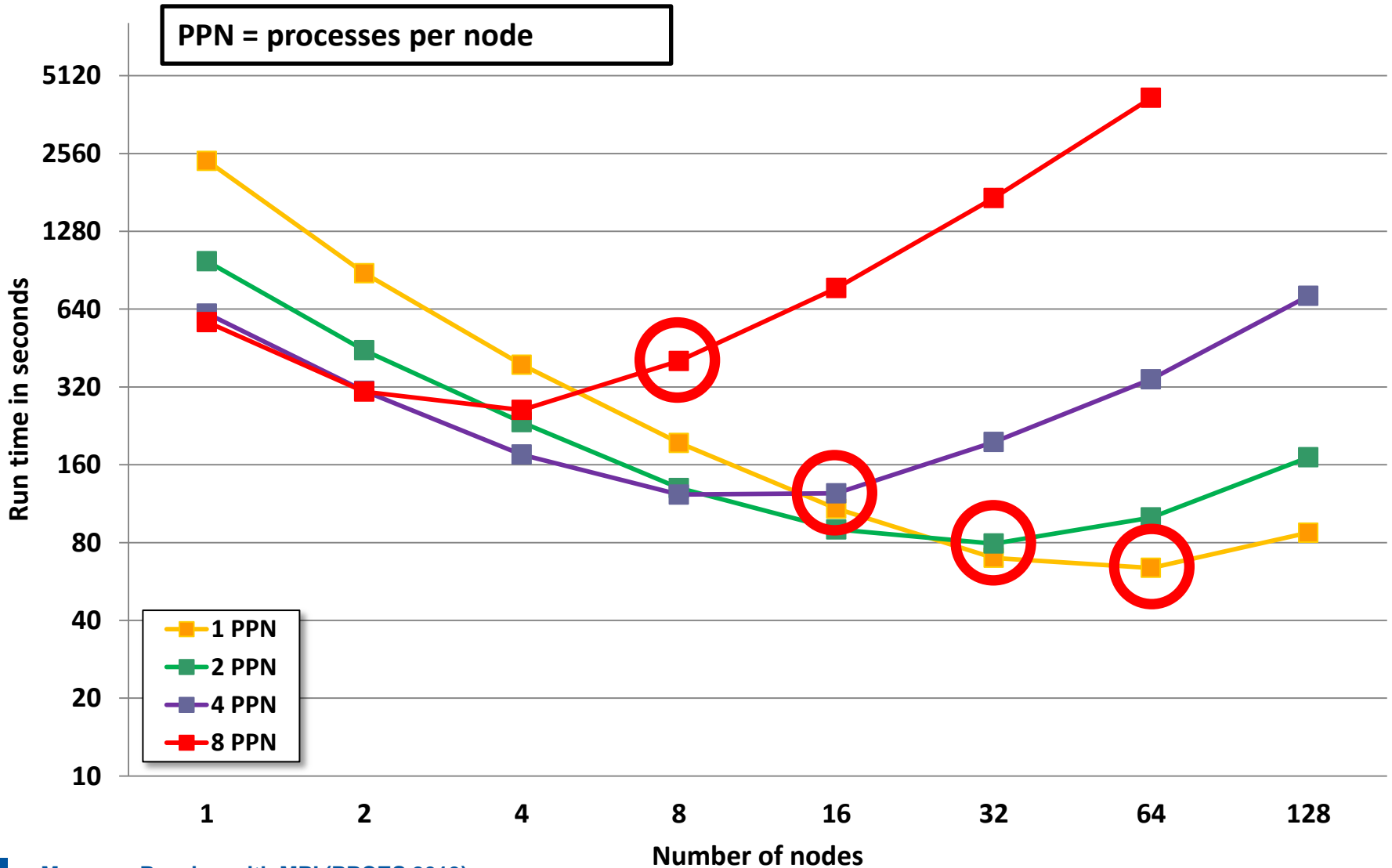
→ more serial time → lower parallel efficiency

→ with large process counts the overhead could negate the parallelization gain

Amdahl's Law: The Ugly Truth™

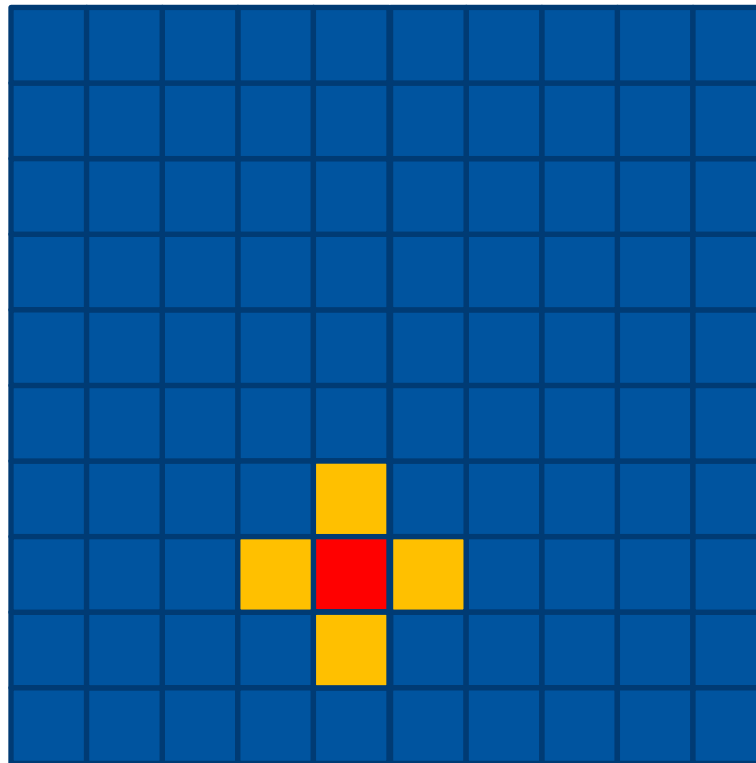


Amdahl's Law: The Ugly Truth™



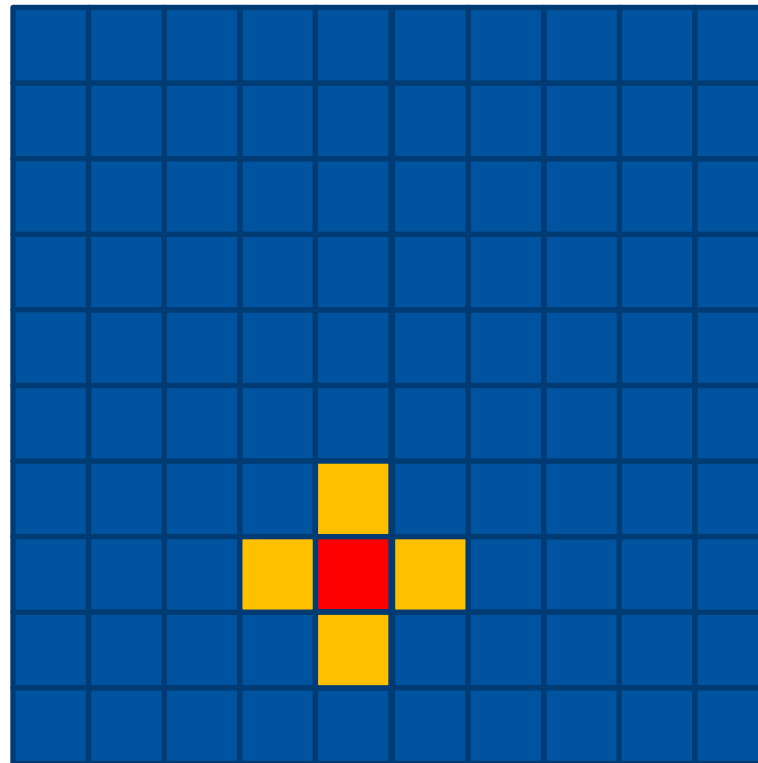
- When decomposing a problem often interdependent data ends up in separate processes.
- Example: iterative matrix update in a PDE solver:

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



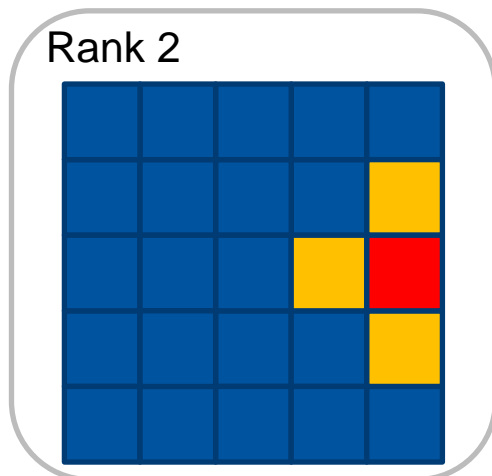
Parallel Patterns: Halos / Ghost Cells

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



Parallel Patterns: Halos / Ghost Cells

■ Domain decomposition



■ Communication

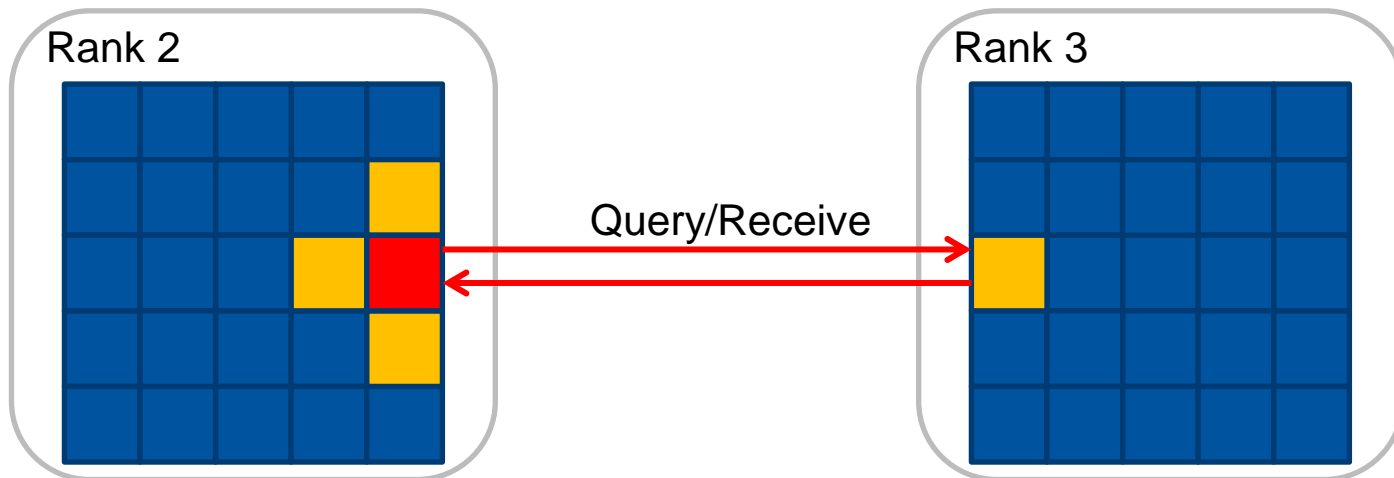
→ For every border cell communication with a neighbor rank is necessary

■ Problems:

→ Introduces synchronization on a very fine level

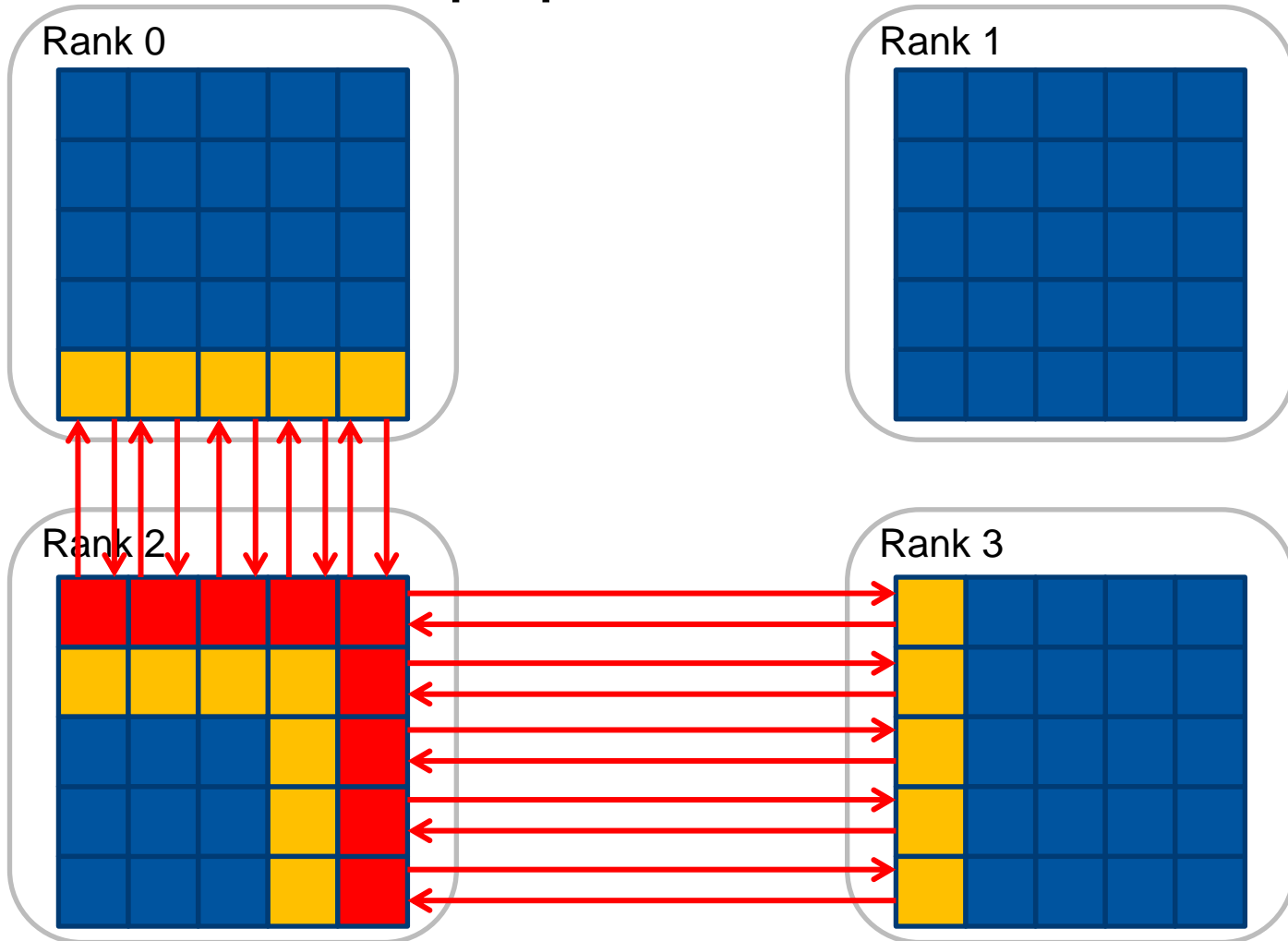
→ Lots of communication calls – highly inefficient

→ Performance can (and will) drastically suffer

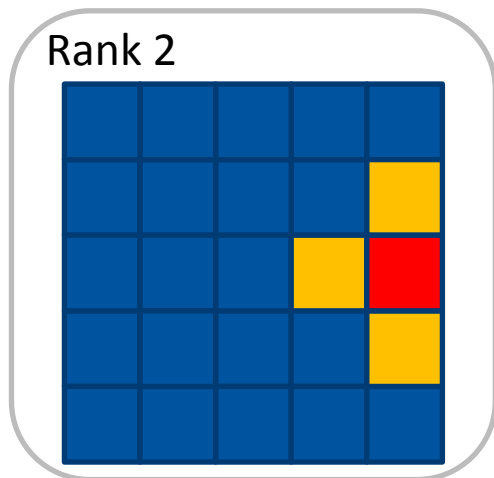


Parallel Patterns: Halos / Ghost Cells

- 10 communication calls per process

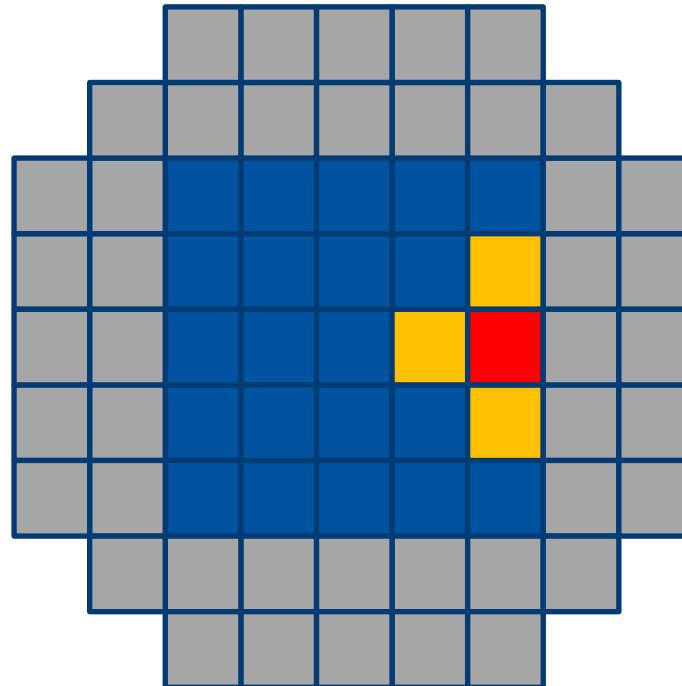


- Copy all the necessary data at the beginning of each iteration

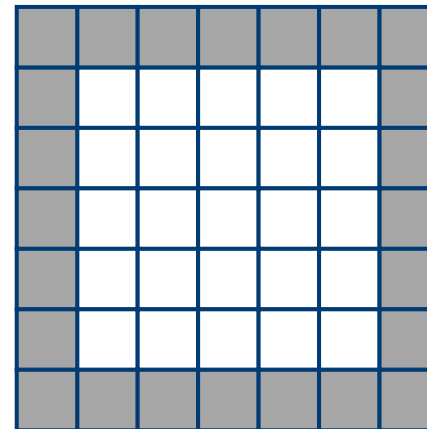
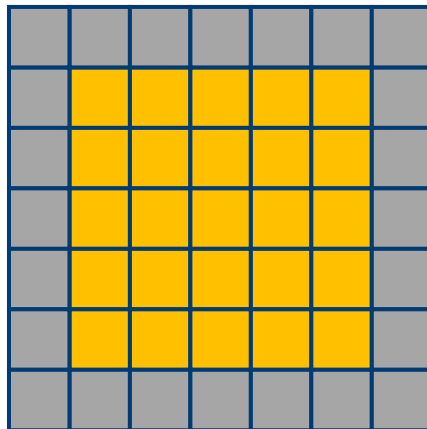
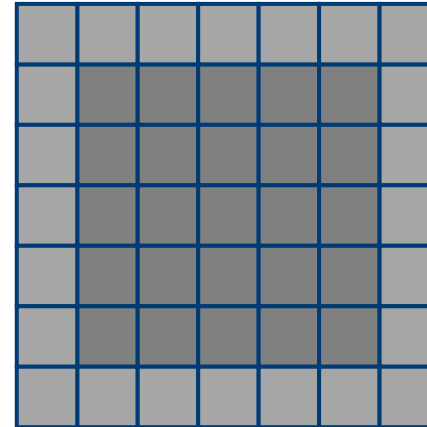
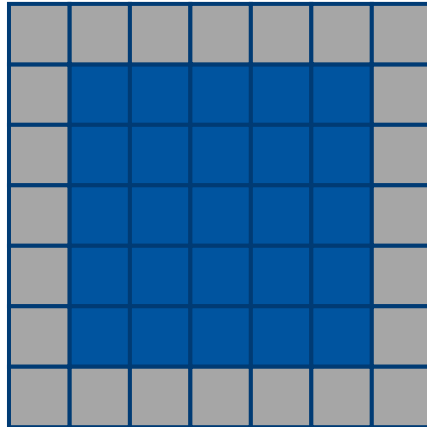


Parallel Patterns: Halos / Ghost Cells

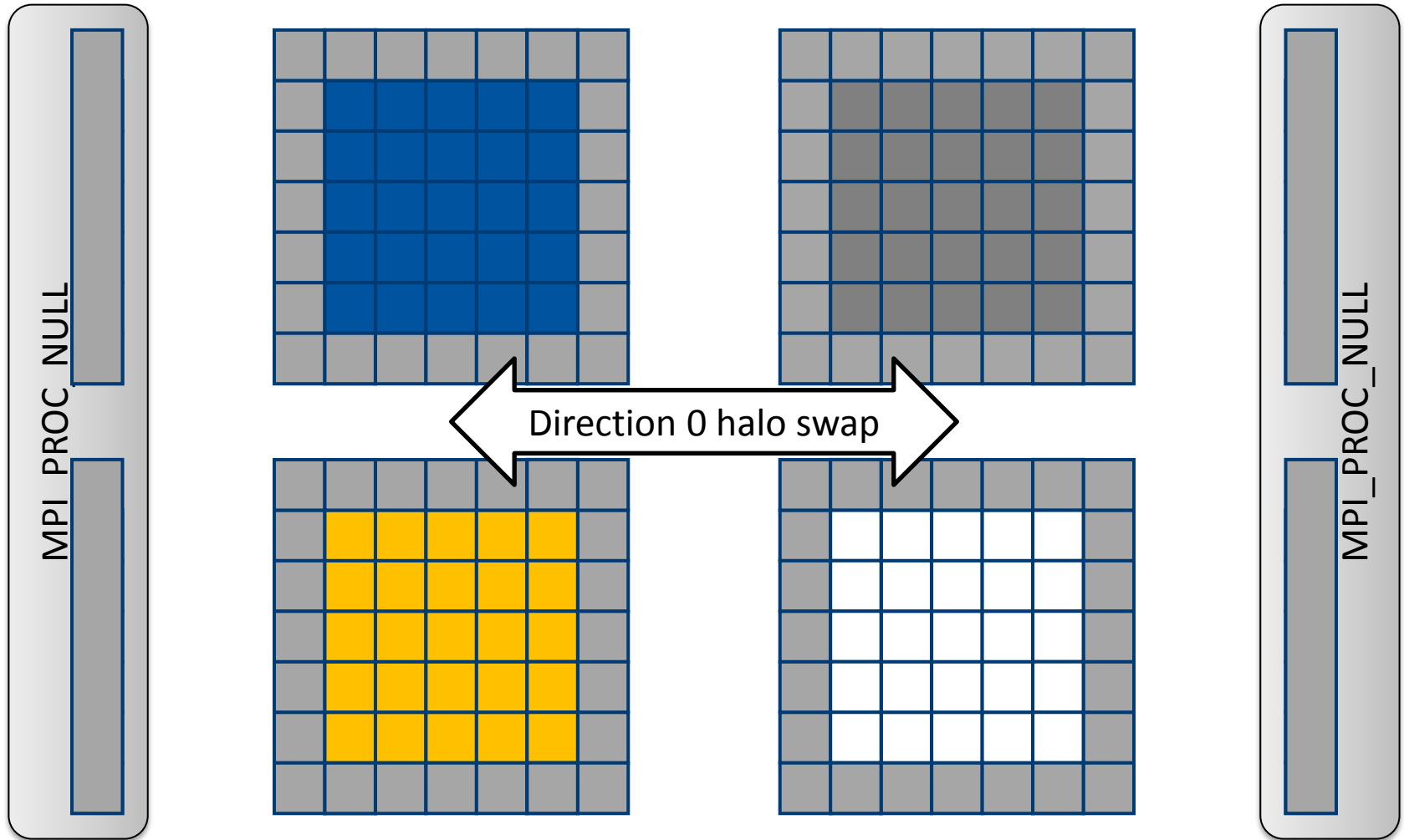
- Copies around the host domain are called *halos* or *ghost cells*
- Multi-level halos are also possible:
 - if required by the stencil
 - reduce the number of communications



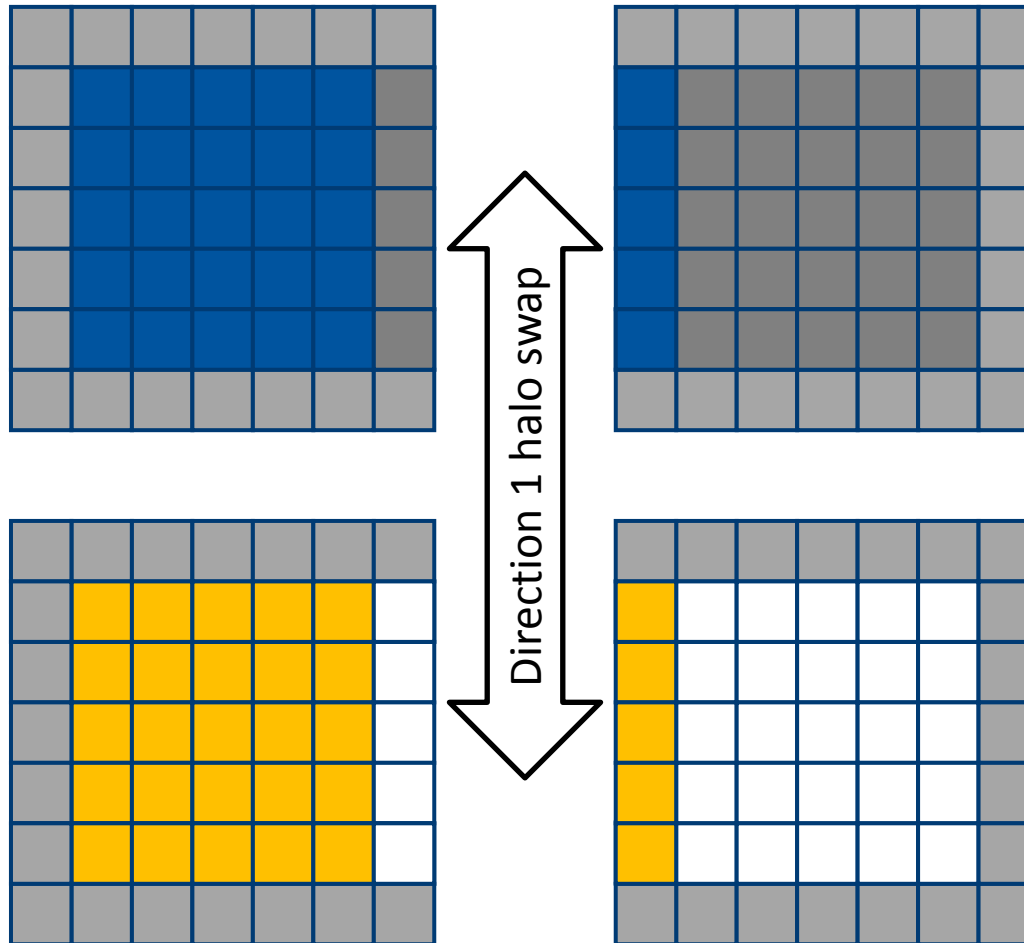
■ Halo Swap



■ Halo Swap



■ Halo Swap



- **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.

■ 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

maxlters = 1000

while ($|z|^2 \leq 2^2$ && iteration < maxlters)

{

$z = z^2 + c$

iteration = iteration + 1

}

if (iteration == maxlters)

color = black

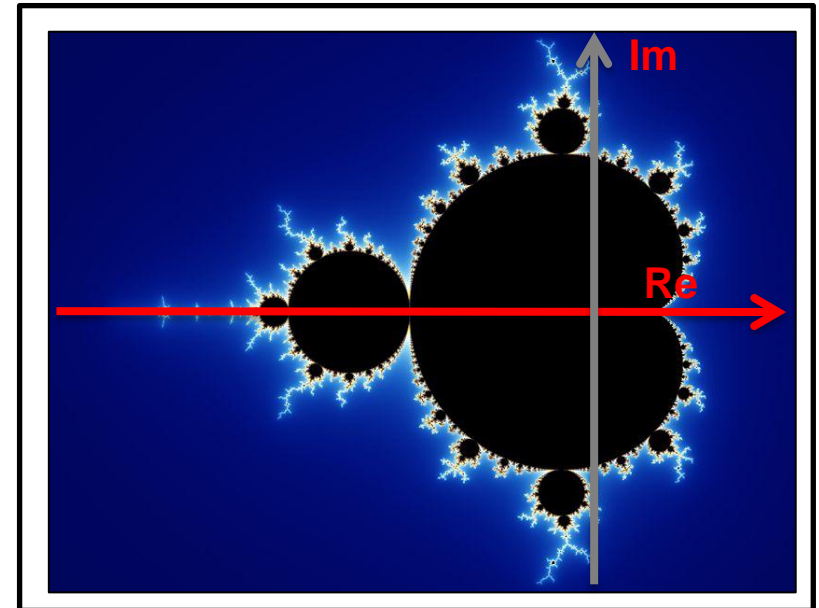
else

color = iteration

plot(x, y, color)

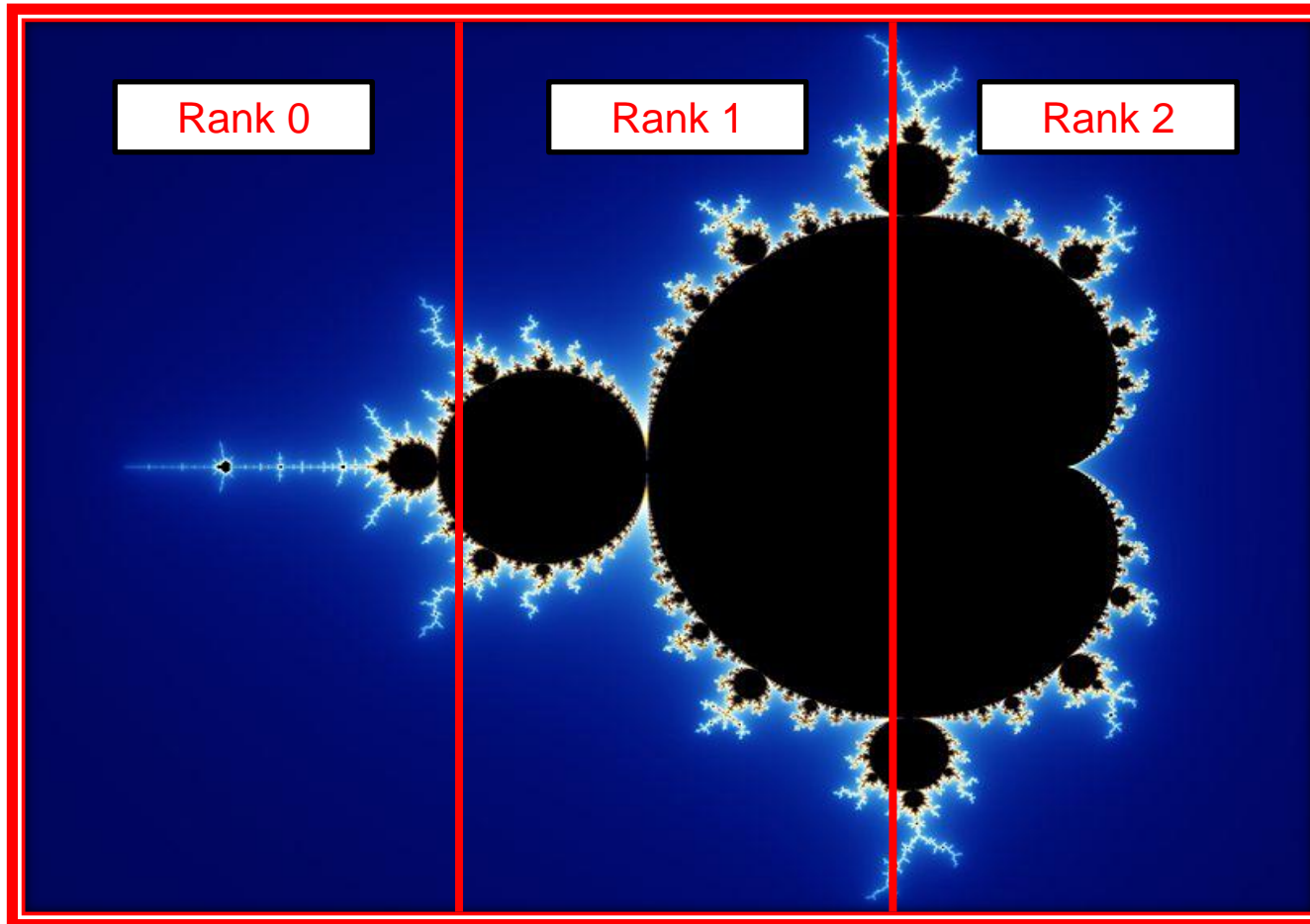
Does the complex series

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



■ Static work distribution:

→ Every MPI rank works on 1/Nth of the problem (N – number of MPI ranks)



■ Mandelbrot set

For each image pixel (x, y) :

// (x, y) – scaled pixel coords

$c = x + i*y$

$z = 0$

iteration = 0

maxlters = 1000

while ($|z|^2 \leq 2^2$ && iteration < maxlters)

{

$z = z^2 + c$

iteration = iteration + 1

}

if (iteration == maxlters)

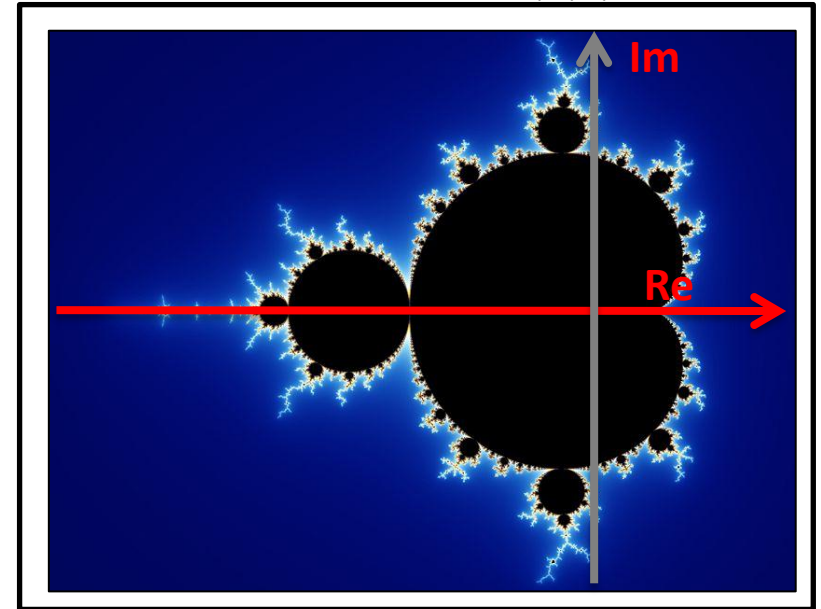
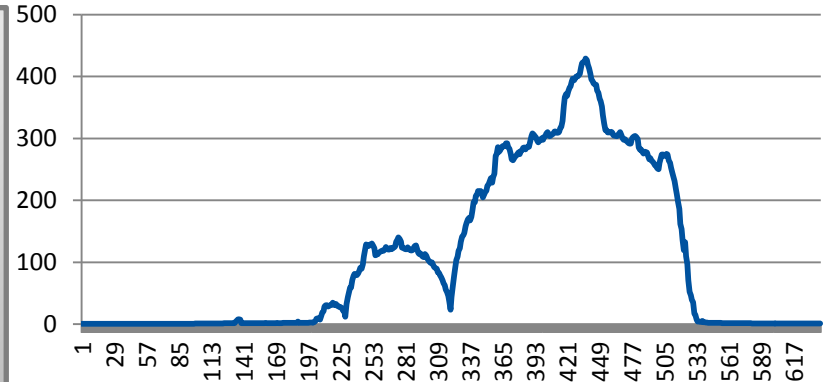
color = black

else

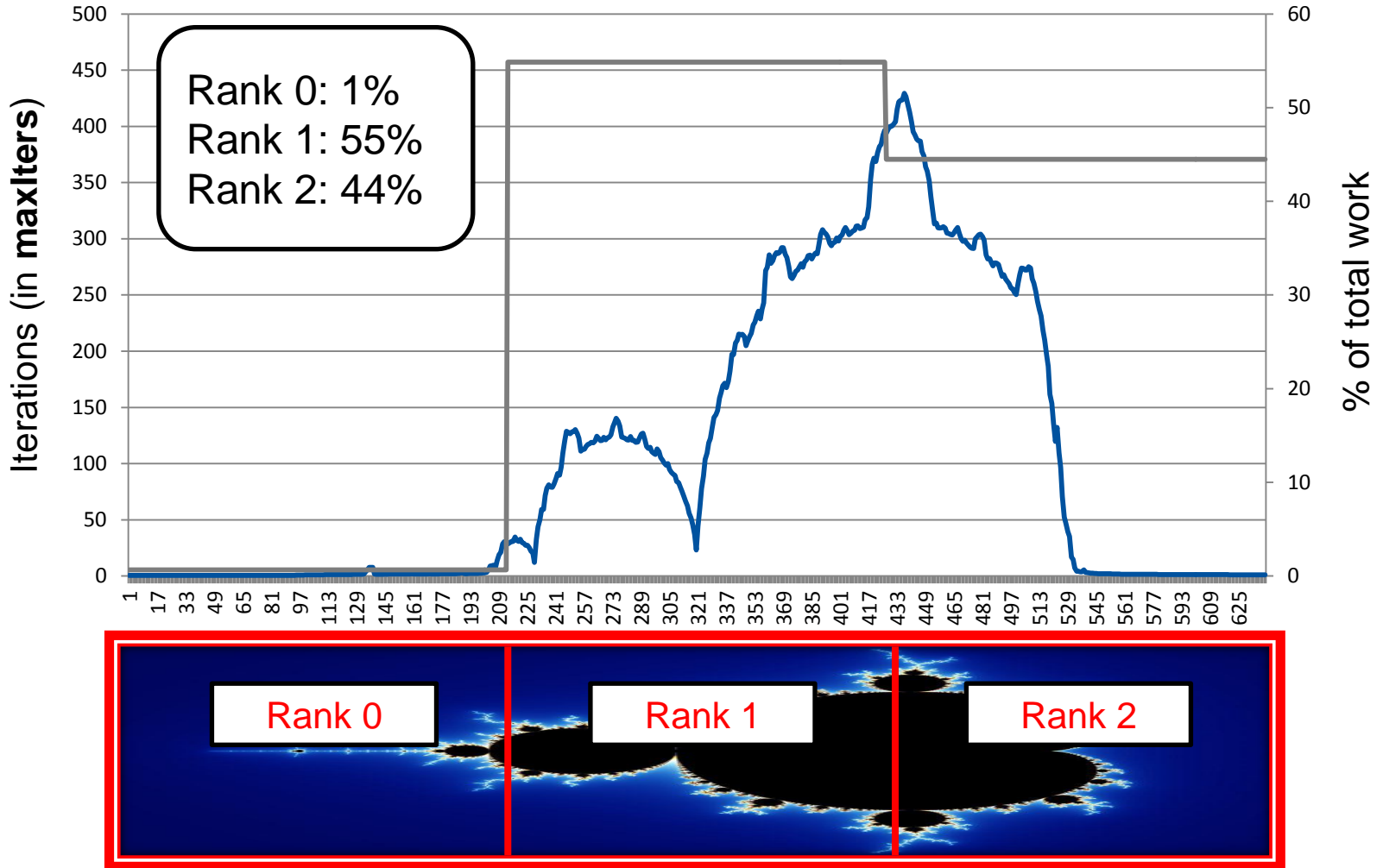
color = iteration

plot(x, y, color)

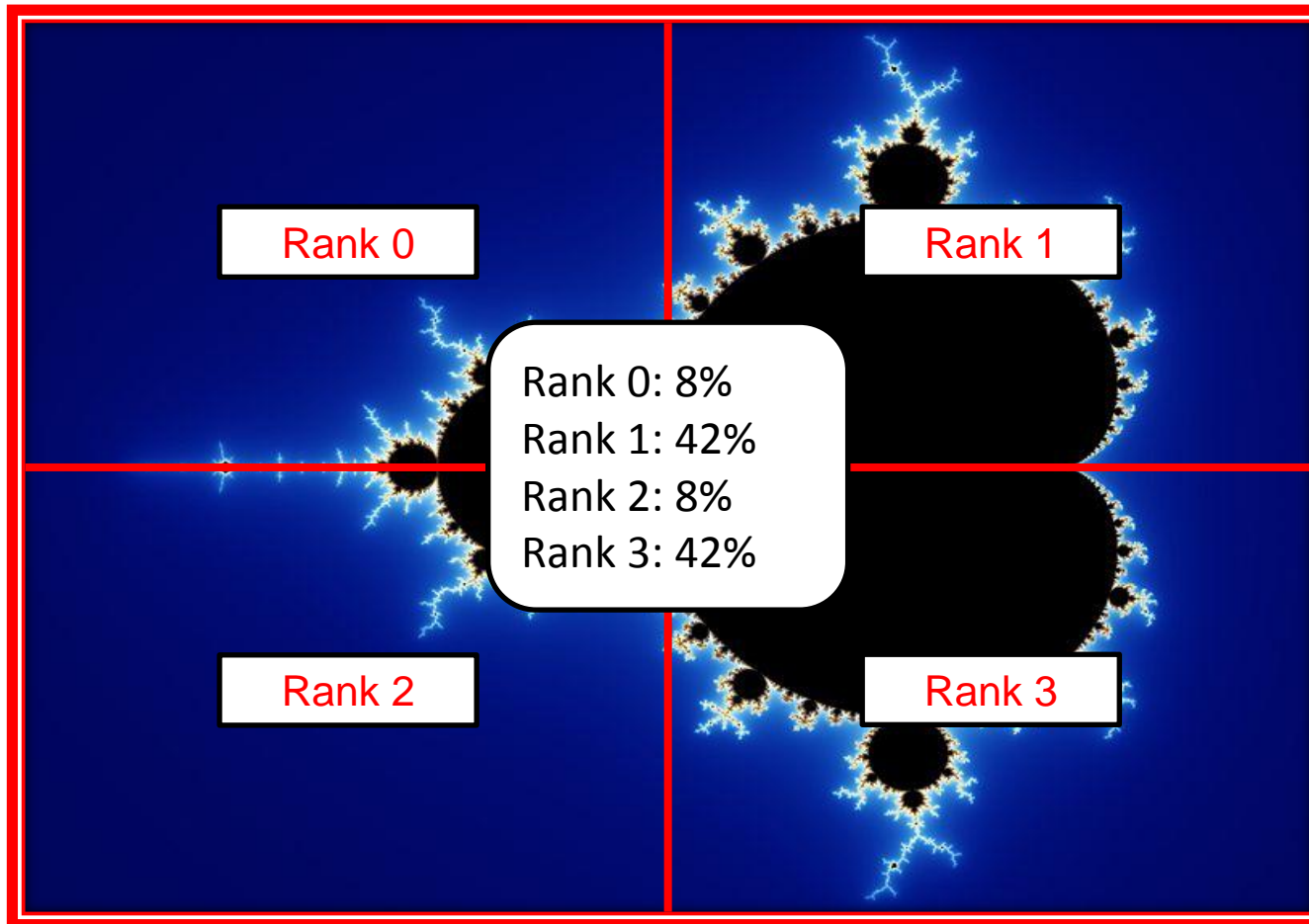
Iterations per image column (in maxlters)



■ Computation complexity mapped to ranks

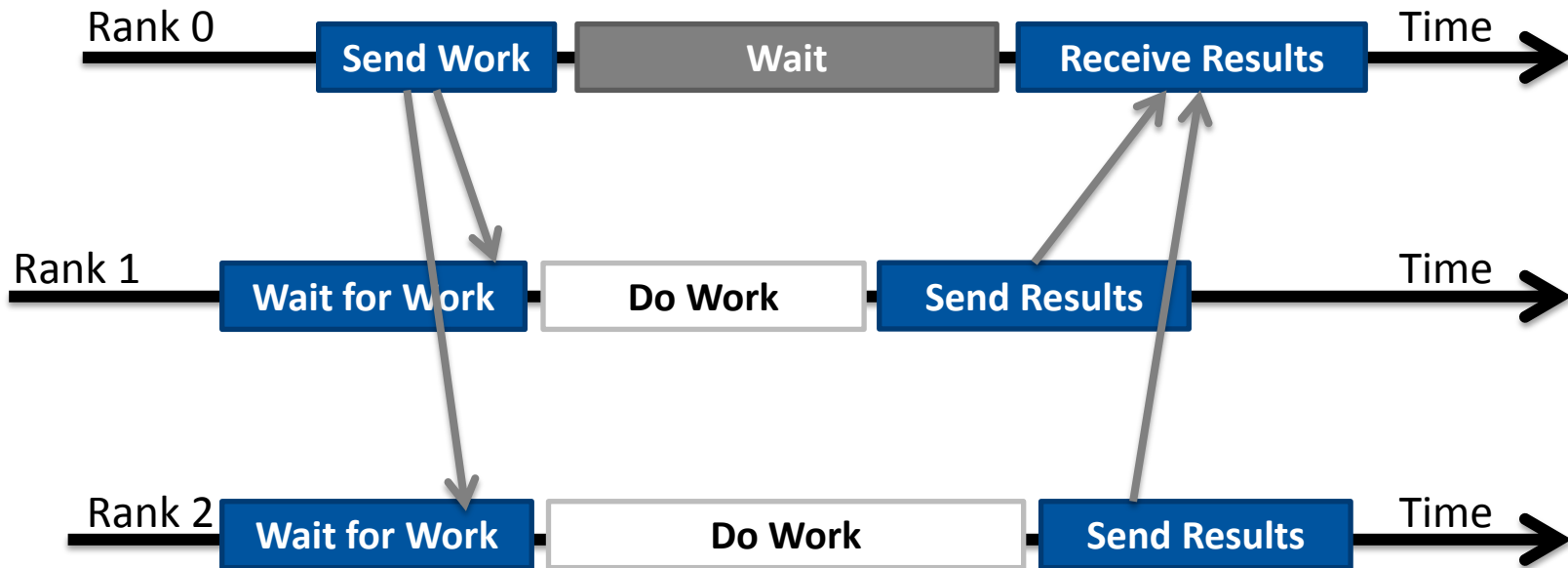


- May be a different decomposition?



Parallel Patterns: Controller – Worker

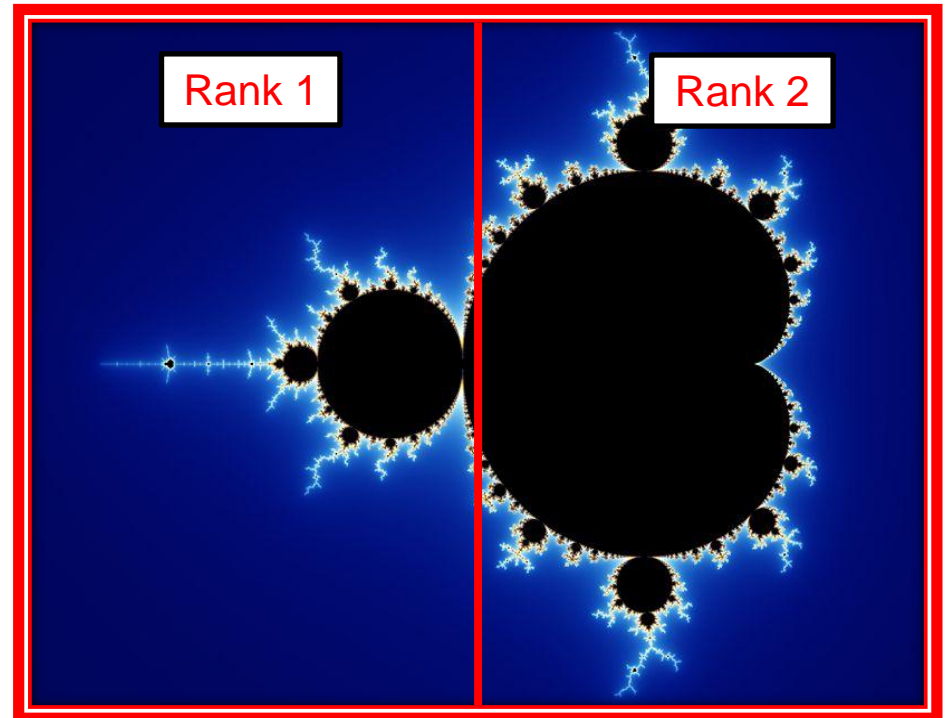
- One process (controller) manages the work.
 - Work is split into many relatively small work items.
- Many other processes (workers) compute over the work items:



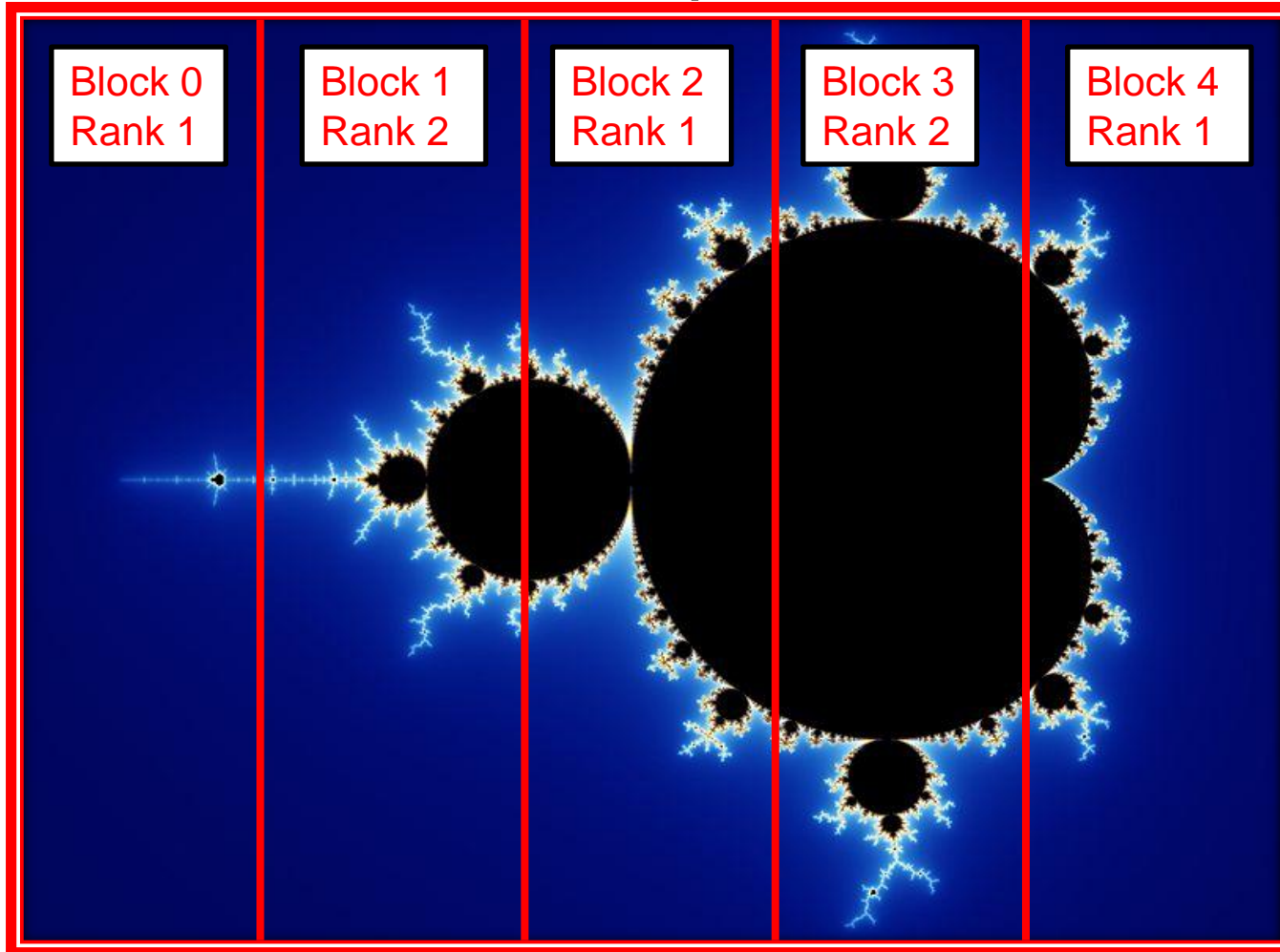
- Above steps are repeated until all work items are processed.
- Sometimes called “bag of jobs” pattern.

■ The algorithm:

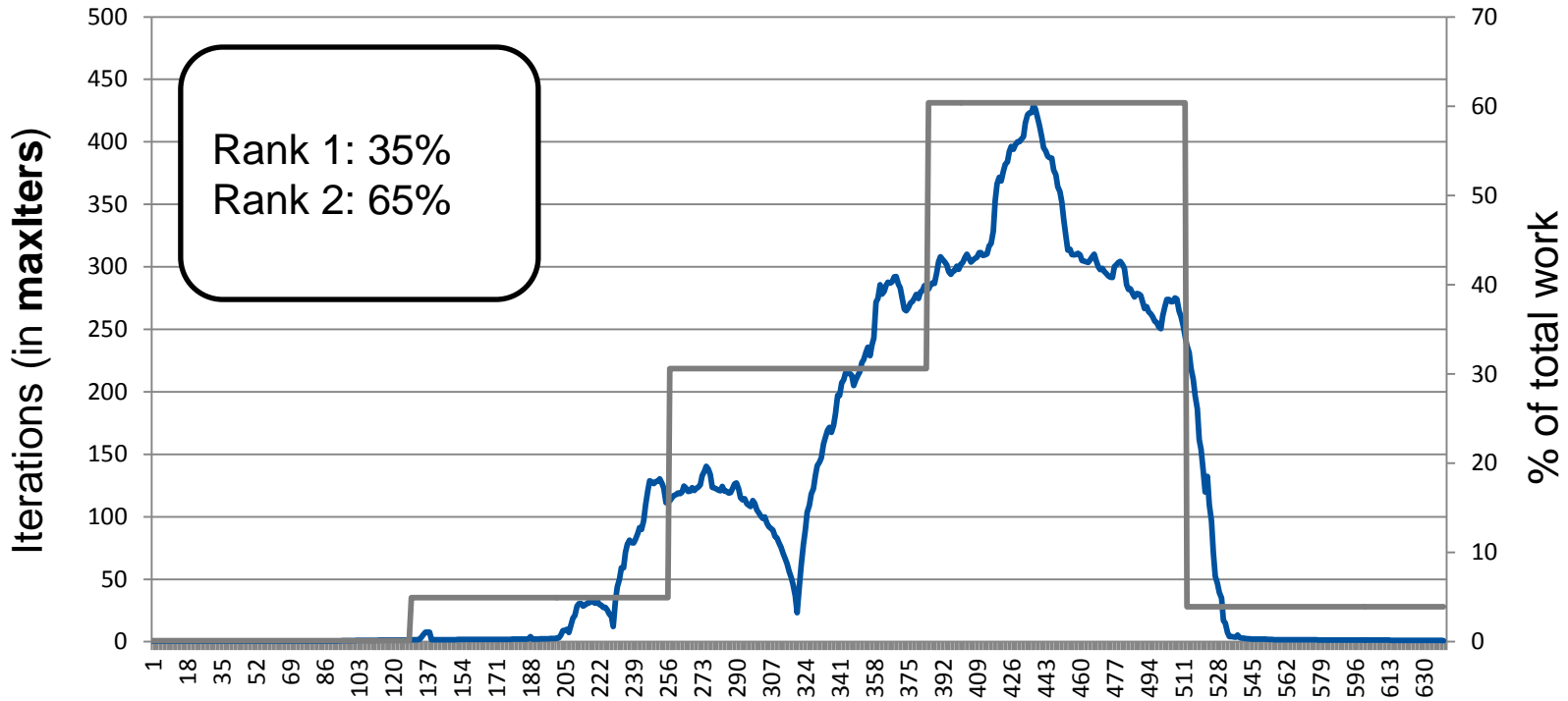
```
START
if (rank == 0)
{
    splitDomain;
    sendWorkItems;
    receivePartialResults;
    assembleResult;
    output;
}
else
{
    receiveWorkItems;
    processWorkItems;
    sendPartialResults;
}
DONE
```



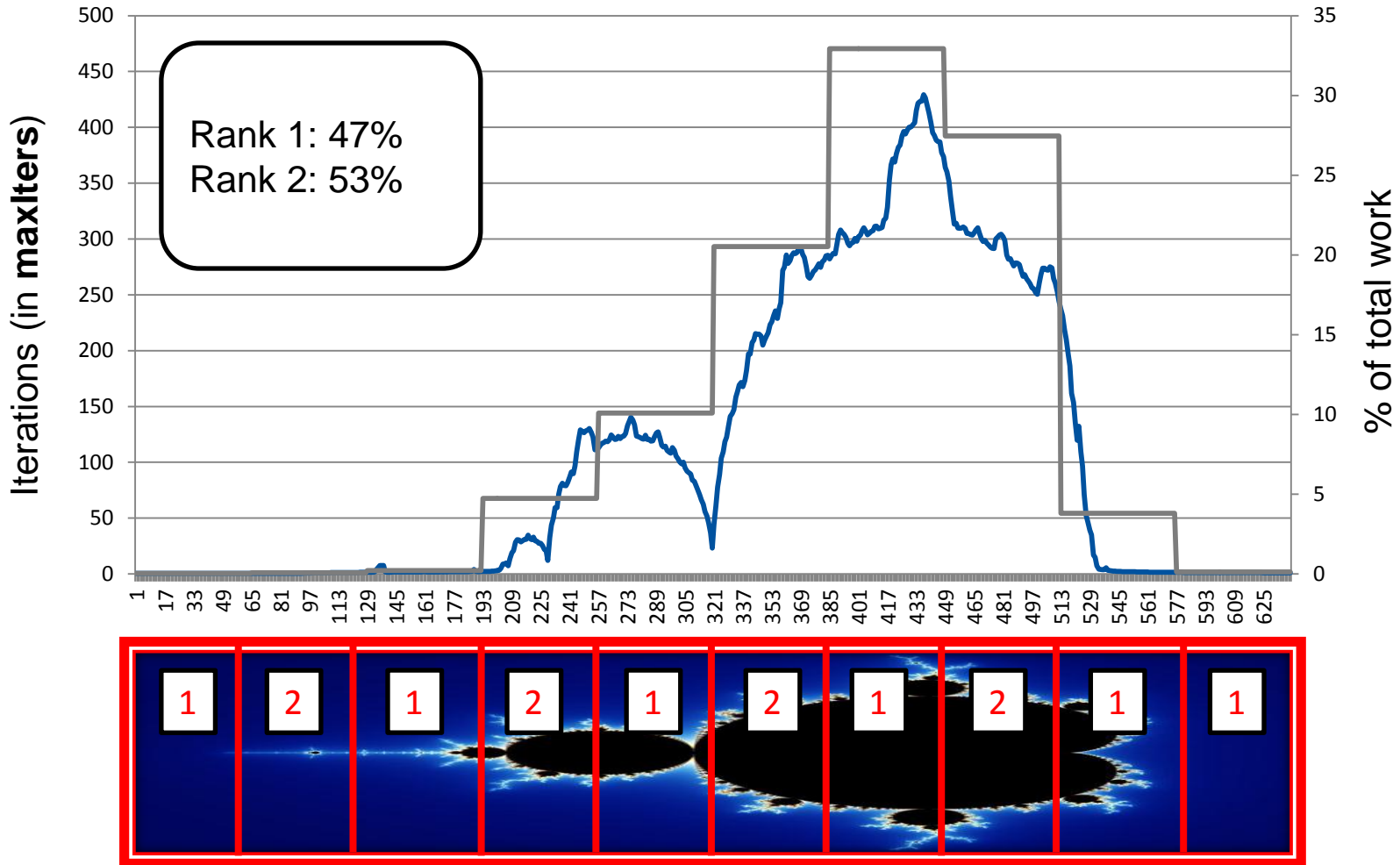
■ Controller – Worker with 3 worker processes



■ Computational complexity mapped to ranks



■ Computational complexity mapped to ranks



■ **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)



Agenda

- Motivation
- MPI – Part 1
 - Concepts
 - Point-to-point communication
 - Non-blocking operations
 - Collective operations
- MPI – Part 2
 - Communicators
 - Hybrid parallelization
 - Common parallel patterns
- **Tools: Brief Introduction to VampirTrace and Vampir**

Serial	Oracle Performance Analyzer
	Intel VTune
	GNU gprof
	Acumem
OpenMP / Shared Memory	Oracle Performance Analyzer
	Intel Threading Tools
	Intel Parallel Studio
	Vampir
MPI	Oracle Performance Analyzer
	Intel Trace Collector and Analyzer
	Microsoft Event Tracing for MS-MPI
	Vampir
	Scalasca

■ VampirTrace / Vampir tool suite:

→ VampirTrace

→ instrumentation tool

→ tracing library

→ writes OTF trace files

→ open-source code

→ Vampir

→ visualization and analysis GUI tool

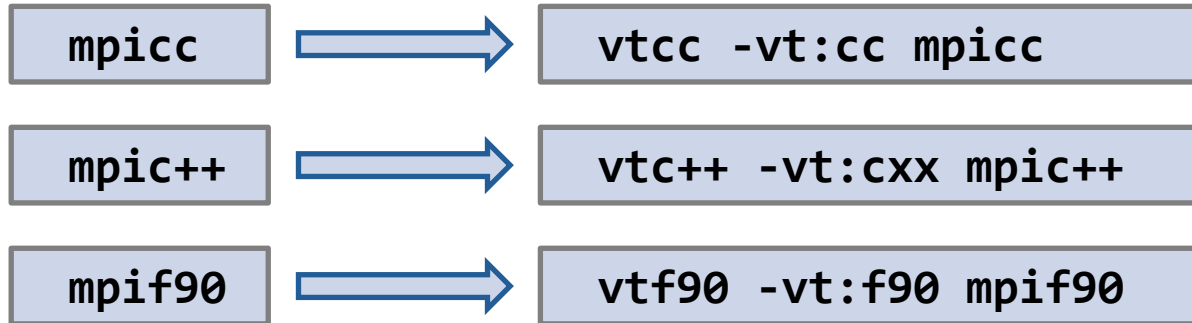
→ reads OTF files produced by VampirTrace

→ commercial

■ On RWTH's cluster modules are located in the UNITE category.

■ Code first has to be instrumented using VampirTrace:

→ Recompile with instrumentation



→ When run, the instrumented binary produces trace files in OTF format.

■ Instrumentation type selected via `-vt:inst <type>`

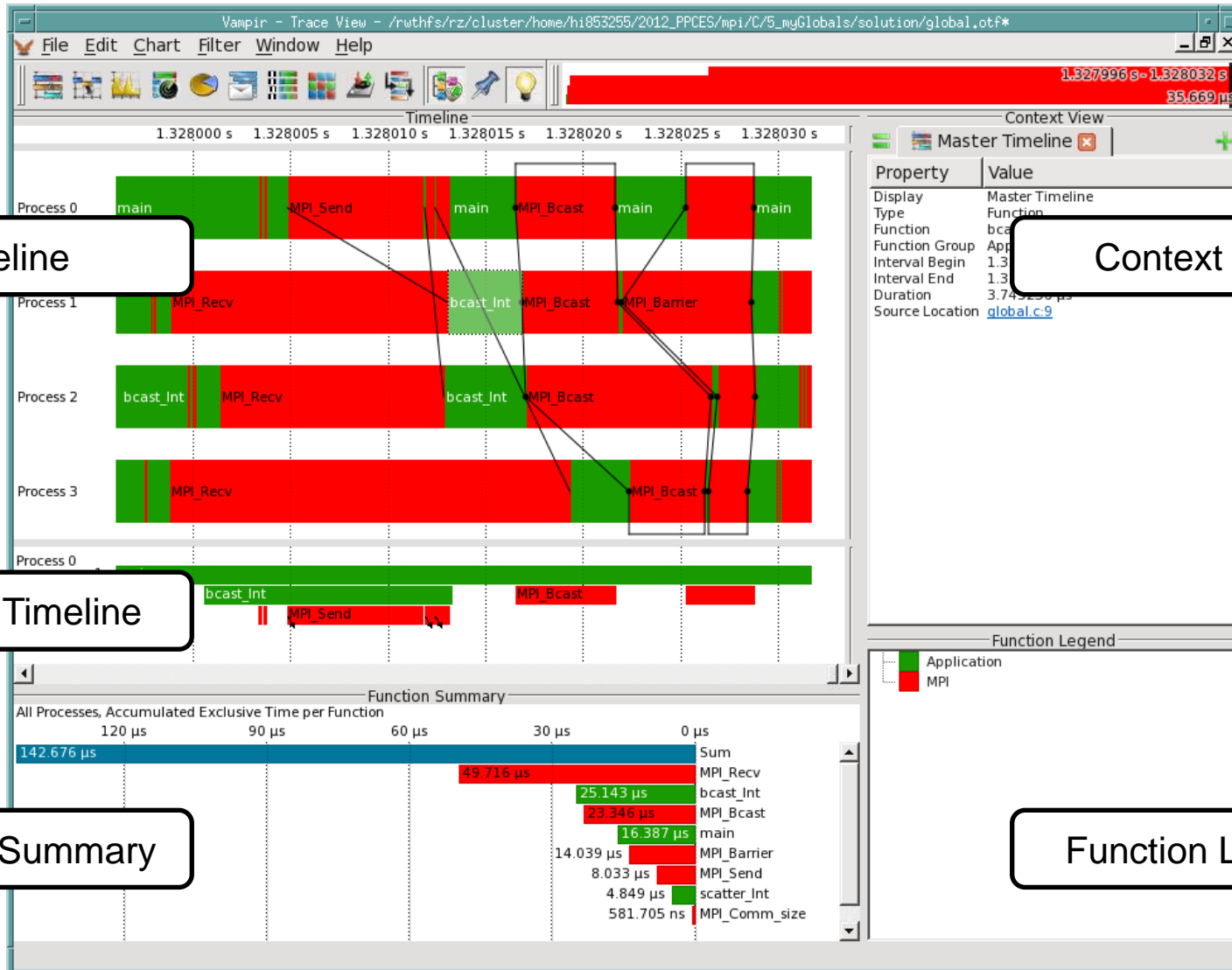
- `compinst` compiler assisted instrumentation (default)
all function calls traced; very detailed; huge trace files
- `manual` manual tracing using VampirTrace API
traces only MPI events and user-specified events;
significantly reduced trace file size

■ VampirTrace is controlled by many environment variables

- `VT_BUFFER_SIZE` internal trace buffer size; flushed to the disk when full (default: 32M)
- `VT_FILE_PREFIX` OTF file prefix (default: executable name)
- `VT_MAX_FLUSHES` number of trace buffer flushes before tracing is disabled (0 – no limit)
- `VT_SYNC_FLUSH` synchronised buffer flushes (default: no)

■ Things to be aware of:

- By default buffers are flushed asynchronously and it takes time
- Significant skew in program's performance profile possible
- No trace written after abnormal program termination



Timeline

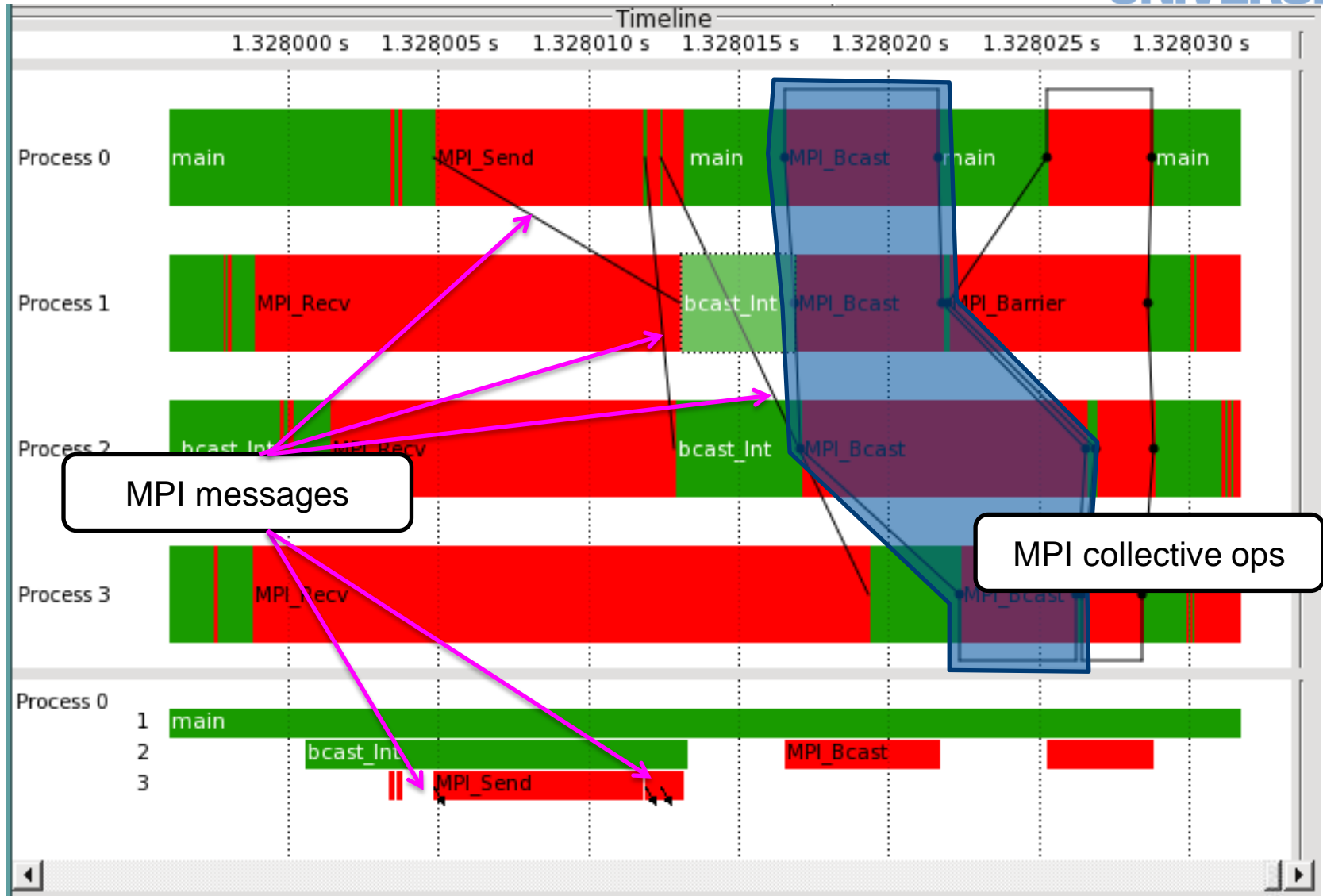
Context View

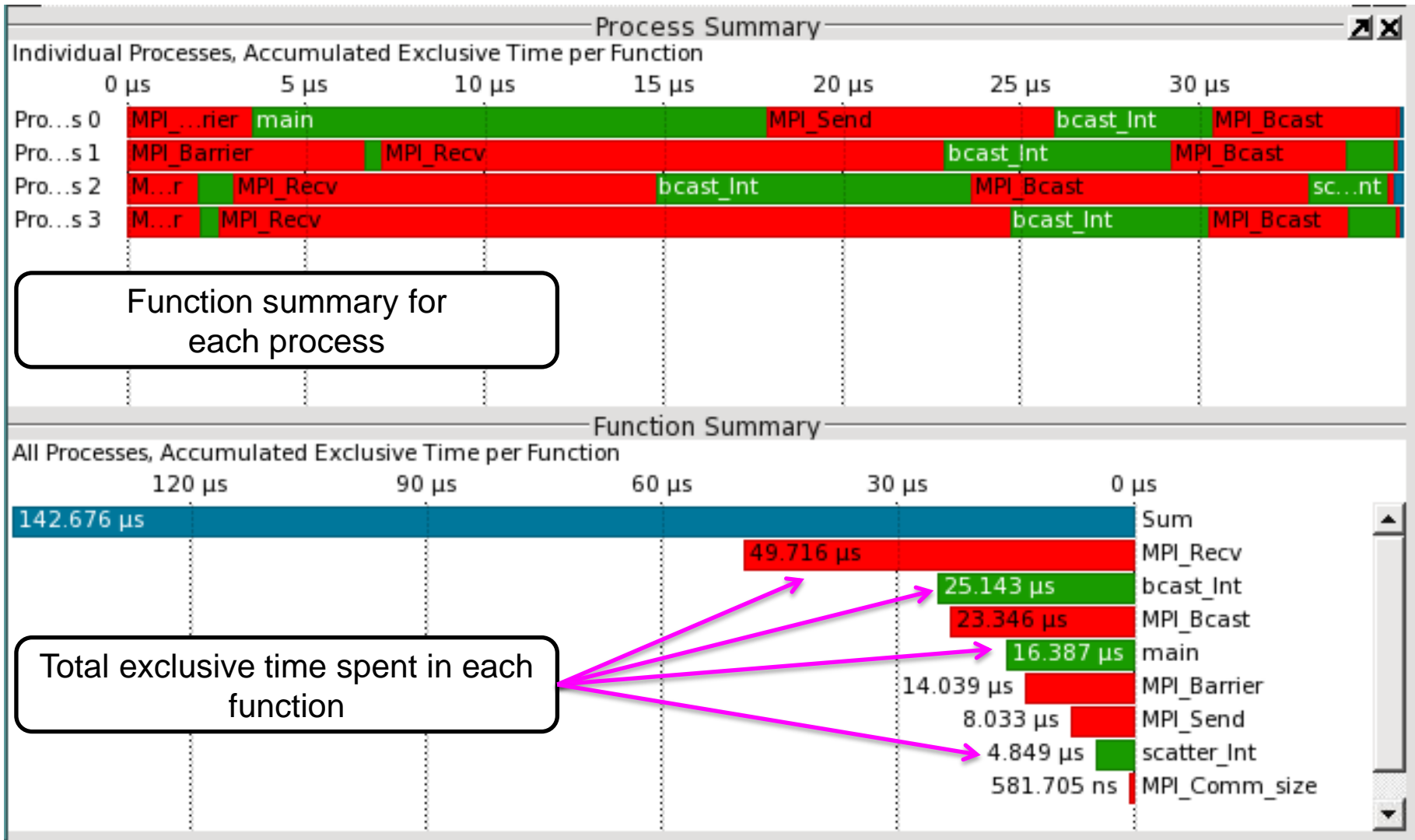
Process Timeline

Function Summary

Function Legend

Vampir: Timeline and Process Timeline

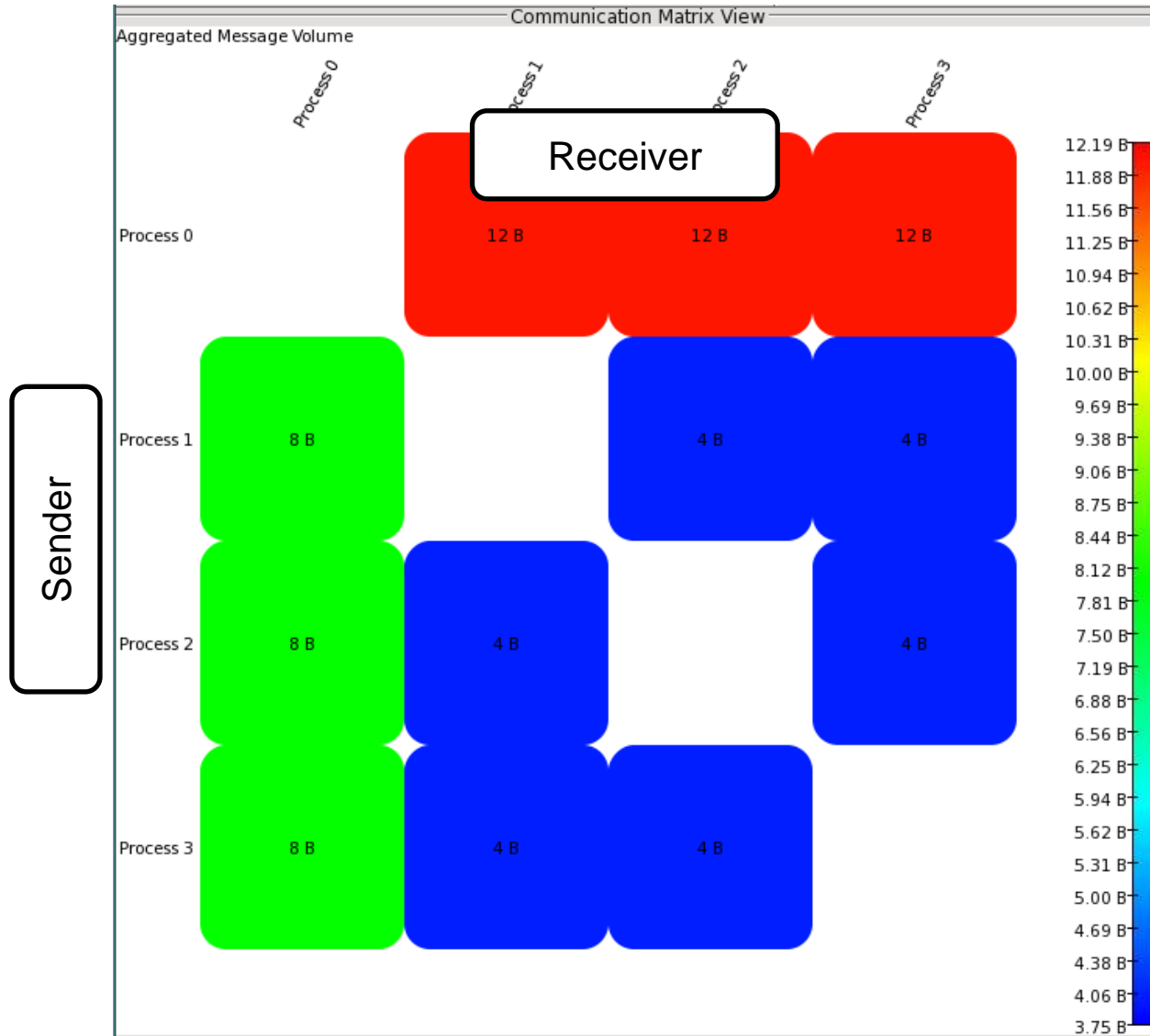




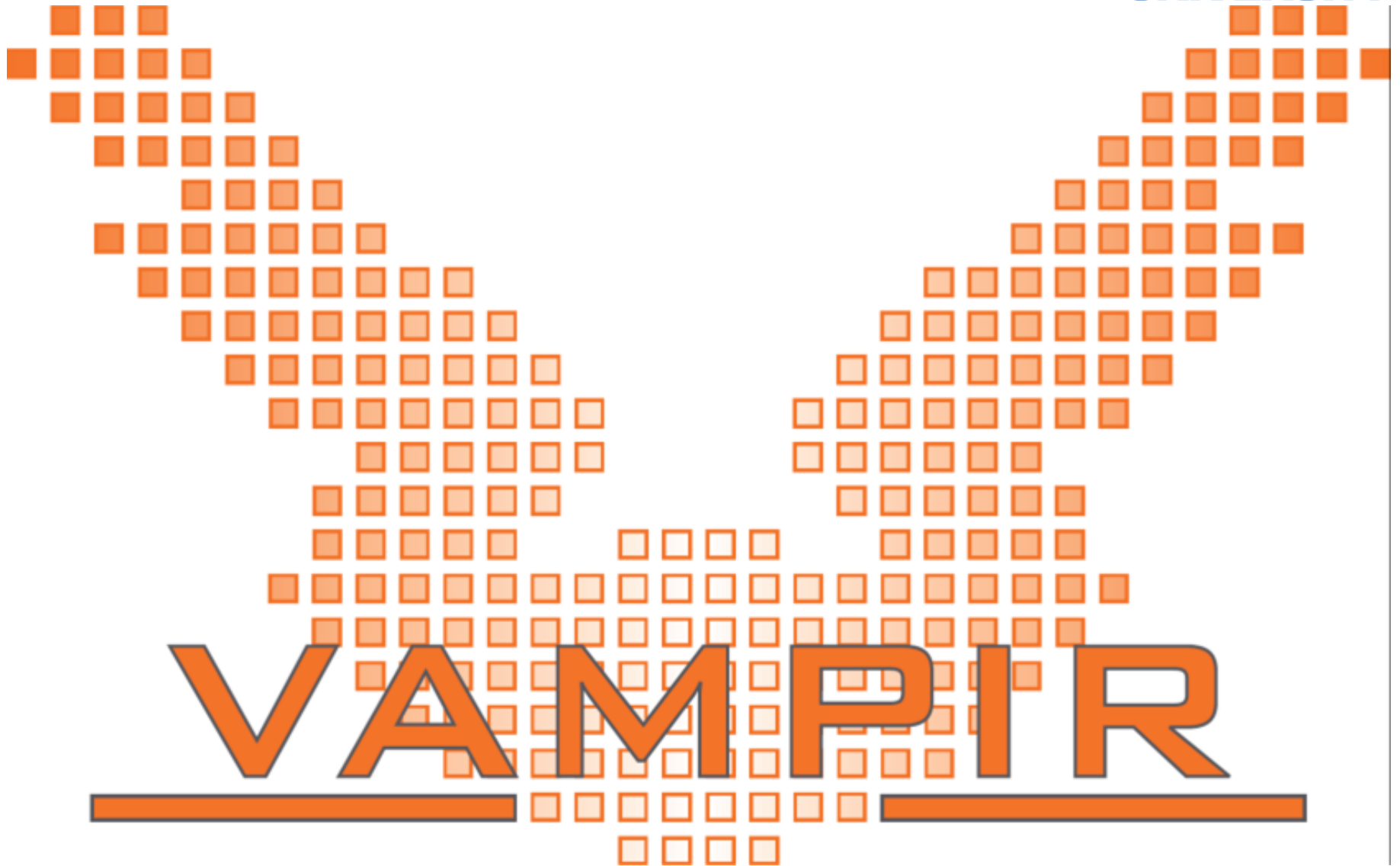
Function summary for each process

Total exclusive time spent in each function

Vampir: Communication Matrix View



Vampir: Live Demonstration



Thank you for your attention!