Message Passing with MPI

Christian Iwainsky

HiPerCH

Agenda

Recap

MPI – Part 1

- Concepts
- Point-to-Point
- Basic Datatypes

MPI – Part 2

The Daily Parallel Life

Recap:
SPMD, MPSD, SMP ... what?

TECHNISCHE UNIVERSITÄT DARMSTADT

- **Multiple Programs Single Data**

  e.g. Stream Processing or Pipes on the Shell

- **Single Program Multiple Data (SPMD)**

  One Program (binary) working on Multiple (parts) of Data

  MPI-programs are a typical form of SPMD

  Typically an identification mechanism is used to control the program flow

```
if (myIdentification == special)
{
        do s.th. different
}
else
{
        ...
}
```

Recap:
The Process-Wall

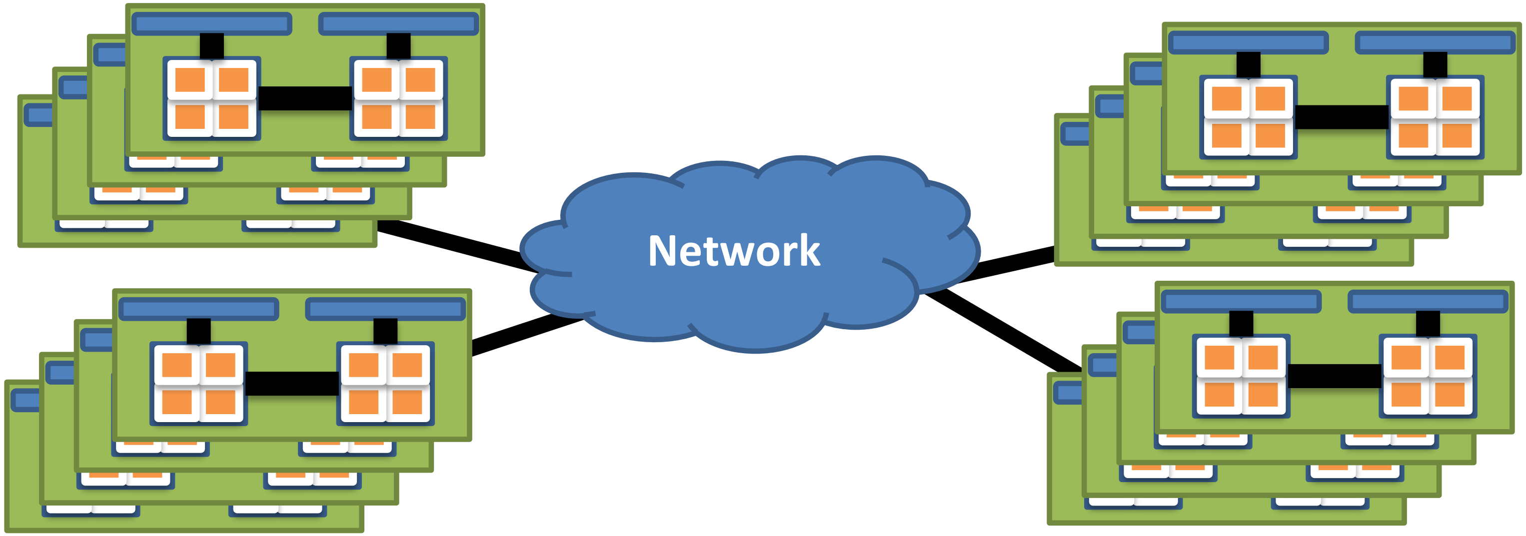
- **Def.: A process is an instance of a computer program**
  - Executable code: e.g. assembly code
  - Memory: Heap, Stack, Process-state (CPU-registers, etc.)
  - One or more threads of execution
  - Operating-system context

- **Important:**
  - Isolation: Once process can not modify any other process without interaction of the operation system
    - No direct data exchange
    - No direct synchronization

TECHNISCHE UNIVERSITÄT DARMSTADT

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



Recap: Interprocess communication

Interaction with other processes

- Shared Memory
  - Restriction: Same machine / motherboard
- File system
  - Slow, shared file-system required
- Sockets/networking and named pipes
  - Coordination/ With whom to communicate
- Other
  - Special libraries
  - **MPI**

Recap:
Example sockets and network

- **Many different pieces of information necessary**
  - How many communication partners
  - Where and how can those partners be reached
    - **Worst case: Full Qualifying Name and Port**:

      e.g. linuxnc001.rz.rwth-aachen.de:21587
  - How to coordinate
    - Does the user have to start each instance of his parallel program

Requirements

- **Identification of participating processes**
  - Who is also working on this problem?
- **Method to exchange data**
  - Whom to send data?
  - What kind of data?
  - How much data?
  - Has the data arrived?
- **Method to synchronize**
  - Are we at the same point in the program?
- **Method to start a set of processes**
  - How do we start processes and get them working together?

➡ **MPI**

Agenda

TECHNISCHE UNIVERSITÄT DARMSTADT

Motivation

MPI – Part 1

- Concepts
- Point-to-Point
- Basic Datatypes

MPI – Part 2

The Daily Parallel Life

- **MPI = Message Passing Interface**
  - a) **de-facto standard API for message passing**
  - b) **Library implementing functionality of the MPI standard**

- **MPI is used to describe (communication) interaction for applications with distributed memory**

MPI Basics

- Startup, Initialisation, Finalization, Shutdown
- MPI_Send, MPI_Recv
- Message-Envelope

Point-to-point Communication

- Basic MPI-Datatypes
- MPI_SendRecv
- MPI_Isend,MPI_Recv,MPI_Wait, MPI_WaitAll, MPI_Query
- MPI_Bsend, MPI_Brecv
- MPI_Ssend,MPI_Srecv
- Common mistakes

TECHNISCHE UNIVERSITÄT DARMSTADT

Startup, Initialization, Finalization, Shutdown for C/C++

C/C++

```
① #include "mpi.h"
② int main(int argc, char ** argv)
  {
  ... some code ...
③ MPI_Init(&argc,&argv);

④ ... computation  & communication ...

⑤ MPI_Finalize();
⑥ ... wrapup ...
  return 0;
  }
```

1. Inclusion of the MPI-header
2. Non coordinated running code: serial
   - NO MPI-calls allowed, few exceptions
   - All program instances run the same code
3. Initialization of the MPI Environment
   Implicit Synchronization
4. User code
   Typically computation and communication
5. Terminate MPI environment
   Internal buffers are flushed
6. Non coordinated running code: serial
   - NO CALLS to MPI functions allowed afterwards

Startup, Initialization, Finalization, Shutdown for Fortran 90

Fortran

```
(1)
(2) program example
    use mpi
     ... some code ...
(3) call MPI_INIT(ierr)

(4) ... computation  & communication ...

(5) call MPI_FINALIZE(ierr)
(6) ... wrapup ...
    end
```

1. Inclusion of the MPI-header
2. Non coordinated running code: serial
   - **NO MPI-calls allowed, few exceptions**
   - **All program instances run the same code**
3. Initialization of the MPI Environment
   Implicit Synchronization
4. User code
   Typically computation and communication
5. Terminate MPI environment
   Internal buffers are flushed
6. Non coordinated running code: serial
   - **NO CALLS to MPI functions allowed afterwards**

MPI – Part 1: General structure

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

Fortran

```
program example
include 'mpif.h'
 ... some code ...
call MPI_INIT(ierr)
... other code ...
call MPI_COMM_SIZE(MPI_COMM_WORLD,      (1)
          numberOfProcs,ierr)
call MPI_COMM_RANK(MPI_COMM_WORLD,      (2)
          myRank,ierr)
... computation  & communication ...
call MPI_FINALIZE(ierr)
... wrapup ...
end
```

(1) Obtain the number of participating processes of this MPI program instance

e.g. if there are 2 processes working **numberOfProcs** would contain 2 after the call

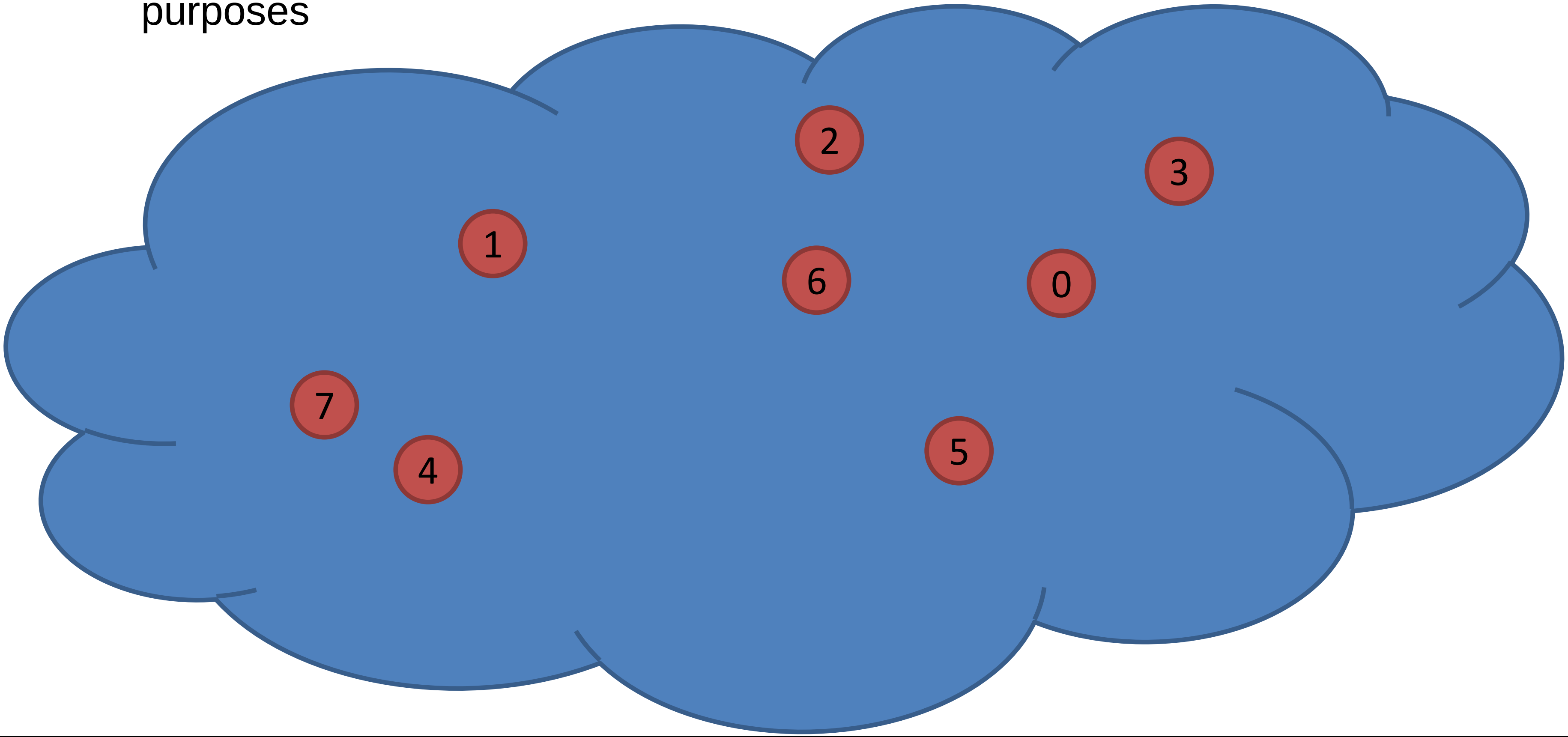
(2) Obtain the number of participating processes of this MPI program instance

Note: MPI numbers its processes starting from "0"

E.g. if there are 2 processes working **myRank** would either be "0" for the first and "1" for the second process after the call

MPI – Part 1: Ranks

- **Instances of an MPI-program are initially indistinguishable**
- **Only difference obtainable through MPI-calls, typically ranks**
  - getpid() should **not** be used as means of differentiation for communication purposes



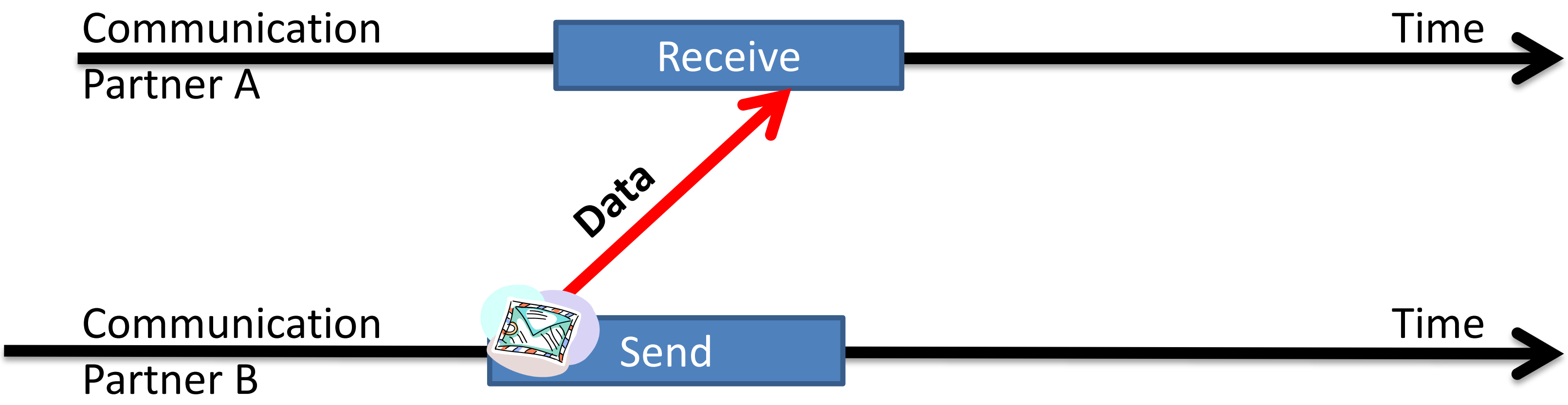
Checkup on the requirements

TECHNISCHE UNIVERSITÄT DARMSTADT

- **Identification of participating processes**
  - Who is also working on this problem?
- **Method to exchange data**
  - Whom to send data?
  - What kind of data?
  - How much data?
  - Has the data arrived?
- **Method to synchronize**
  - Are we at the same point in the program?
- **Method to start a set of processes**
  - How do we start processes and get them working together?

TECHNISCHE UNIVERSITÄT DARMSTADT

- **Recall: Goal to enable communication of parallel processes with isolated address spaces**



- **Required:**
  Send
  Receive
  Identification of receiver and sender
  Specification of what needs to be send/received

TECHNISCHE UNIVERSITÄT DARMSTADT

MPI_Send:

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

- data: memory location containing data
- count: number of elements of type "type" to send
- type: MPI type of each element
- dest: destination rank for the data
- tag: identification of message
- comm: communicator

```
MPI_SEND(DATA, COUNT, TYPE,DEST,TAG,COMM, IERROR)
```
Fortran

MPI – Part 1: Receiving

- MPI_Recv:

C/C++

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

- data: memory location intended for data
- count: number of elements of type “type” to send
- type: MPI type of each element
- source: rank to receive data from
  Special rank: MPI_ANY_SOURCE
- tag: identification of message to receive
  Special tag: MPI_ANY_TAG
- comm: communicator
- status: status of the return
  Special status: MPI_STATUS_IGNORE

Fortran

```
MPI_RECV(DATA, COUNT, TYPE,DEST,TAG,COMM, STATUS,IERROR)
```

MPI – Part 1: MPI-Return Value

- (Most) C/C++ - MPI calls have an return value
  e.g.: int MPI_Init(…)
- (Most) Fortran - MPI calls have and appended variable for the error-code
  e.g.: MPI_INIT(ierror)

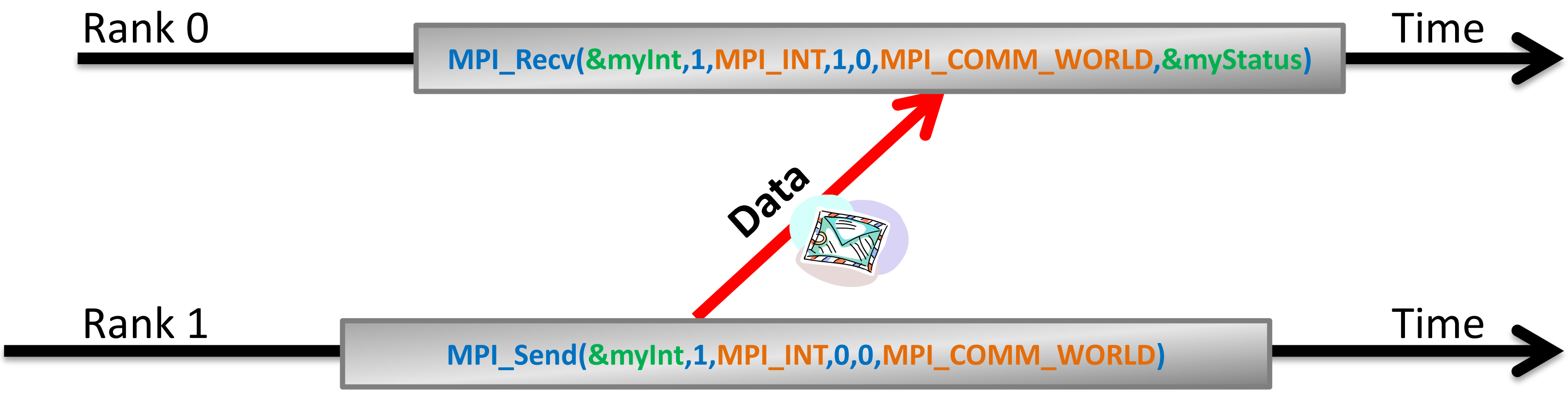
- This return value indicates the success
  C/C++: MPI_SUCCESS == MPI_Init(..)
  Fortran: call MPI_INIT(ierror)
  ierror .eq. MPI_SUCCES
  or failure otherwise.

- Node: returned error-codes are MPI-Implementation specific

For this presentation no more Fortran calls

TECHNISCHE UNIVERSITÄT DARMSTADT

Using MPI-Functions:



- These two transfer the content of the (integer) variable myInt from **rank 1** to **rank 0**

Checkup on the requirements

- **Identification of participating processes**
  - ✓ Who is also working on this problem?
- **Method to exchange data**
  - ✓ Whom to send data?
  - ✓ What kind of data?
  - ✓ How much data?
  - ✗ Has the data arrived?
- **Method to synchronize**
  - Are we at the same point in the program?
- **Method to start a set of processes**
  - How do we start processes and get them working together?

MPI – Part 1: Simple Message Passing Example

C/C++

```
#include "mpi.h"
int main(int argc, char ** argv)
{
int numberOfProcs,myRank,data;
MPI_Init(&argc,&argv);                          ①
MPI_Comm_size(MPI_COMM_WORLD,                   ②
        &numberOfProcs);
MPI_Comm_rank(MPI_COMM_WORLD,
        &myRank);
if (myRank==0)                                  ③
  MPI_Recv(&data,1,MPI_INT,1,0,                 ④
        MPI_COMM_WORLD,&status);
else if (myRank==1)
  MPI_Send(&data,1,MPI_INT,0,0,
        MPI_COMM_WORLD);
MPI_Finalize();                                 ⑤
return 0;
}
```

① Initialization of MPI library

② Get identification of processes

③ Do s.th. different for each/a process

④ Communicate

⑤ Stop the MPI library

MPI – Part 1: Compilation

TECHNISCHE UNIVERSITÄT DARMSTADT

- MPI is a typical library
- For convenience vendors provide specialized compiler wrappers
  (simply replace regular compiler calls)

icc	mpiicc
icpc	mpiicpc
ifort	mpif90

```
mpiexec –n <maxprocs> … <program> <arg1> <arg2> <arg3> …
```

- A call to mpiexec starts the maxprocs instances of the program with arguments arg1, arg2, … and provides the MPI library with enough information to establish its network connections

- The MPI-standard specifies, but does not require this program

Checkup on the requirements

- **Identification of participating processes**
  - ✓ Who is also working on this problem?
- **Method to exchange data**
  - ✓ Whom to send data?
  - ✓ What kind of data?
  - ✓ How much data?
  - ✗ Has the data arrived?
- **Method to synchronize**
  - Are we at the same point in the program?
- **Method to start a set of processes**
  - ✓ How do we start processes and get them working together?

TECHNISCHE UNIVERSITÄT DARMSTADT

- Reception of an MPI Message is only controlled by the Message Envelope
- Recap: 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

- Recap: MPI_Recv

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

C/C++

MPI – Part 1: Message Envelope and Matching

TECHNISCHE UNIVERSITÄT DARMSTADT

- **Caveat: Reception of an MPI Message is also dependent on the data**
- **Recap:**

```
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 data types to match**
  - NOT ENFORCED BY THE IMPLEMENTATIONS

- **For a receive to complete also enough data has to have arrived**

Rank 0:
```
MPI_Send(myVar,1,MPI_INT,1,0,MPI_COMM_WOLRD)
… some code …
MPI_Send(myVar,1,MPI_INT,1,0,MPI_COMM_WOLRD)
```

Rank 1:
```
MPI_Recv(myVar,2,MPI_INT,0,0,MPI_COMM_WOLRD,stat)
… some code …
```

Incomplete

Message Reception

- **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: INTEGER, DIMENSION(MPI_STATUS_SIZE) :: status

- status(MPI_SOURCE) message source rank
- status(MPI_TAG) message tag
- status(MPI_ERROR) receive status code

MPI provides many predefined data types

- Fortran:

MPI datatype	Fortran datatype
MPI_INTEGER	INTEGER
MPI_REAL	REAL
MPI_DOUBLE_PRECISION	DOUBLE PRECISION
MPI_COMPLEX	COMPLEX
MPI_LOGICAL	LOGICAL
MPI_CHARACTER	CHARACTER(1)
MPI_BYTE	n.a.
…	…

8 binary digits

- C/C++
- Userdefined datatypes

MPI – Part 1:
MPI-Data-Types

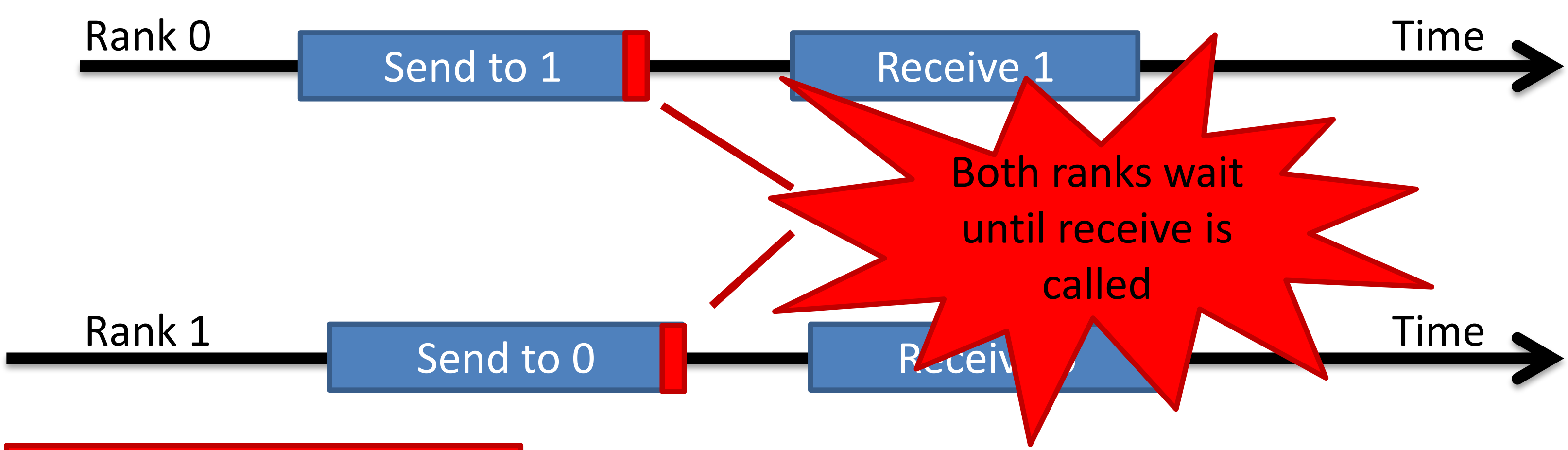
MPI provides many predefined data types

- Fortran
- C/C++:

MPI datatype	Fortran datatype
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	n.a.

- Userdefined datatypes

- **All MPI function calls are blocking**
  - Except when explicitly specified differently
  - Function call only returns once its operation has completed
  - NOTE: MPI_Send and MPI_Recv are blocking
- **Example:**

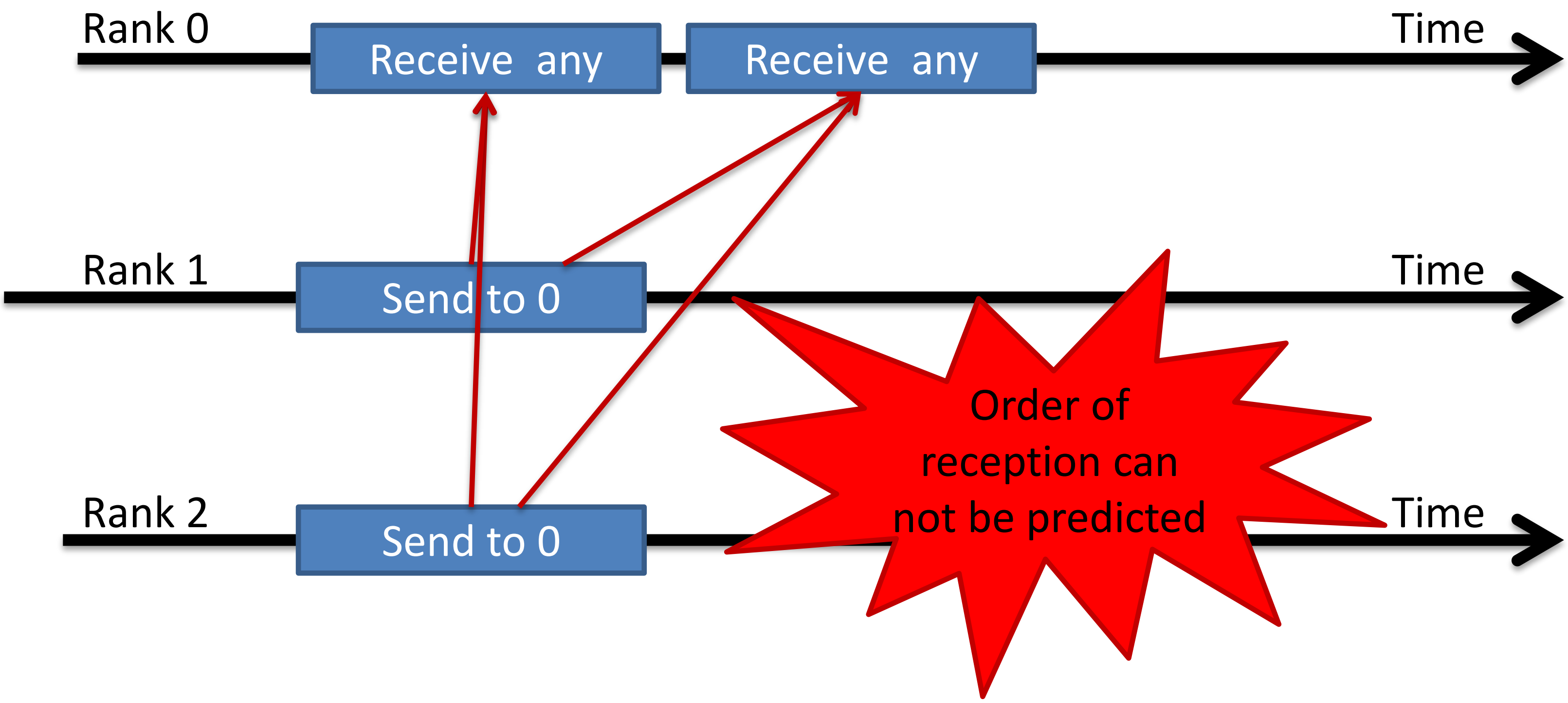


Called: DEADLOCK

TECHNISCHE UNIVERSITÄT DARMSTADT

- **Example:**

In which order are the messages arriving?



Called: RACE CONDITION

MPI – Part 1: MPI_Sendrecv (1/2)

TECHNISCHE UNIVERSITÄT DARMSTADT

```
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++

- **Arguments:**

	Send	Receive
Data	sendData	recvData
Count	sendCount	recvCount
Type	sendType	recvType
Destination	Dest	- n.a. -
Source	- n.a. -	Src
Tag	sendTag	recvTag

- comm: common communicator
- status: status of the recv-part

TECHNISCHE UNIVERSITÄT DARMSTADT

Note:

- MPI_Sendrecv sends only one message and receives one message
- The send and the receive – data may not overlap

  use MPI_Sendrecv_replace instead:

C/C++

```
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 send data, then receive (same amount of) data to the same memory location

- **The MPI_Status struct contains information about the received message:**

  “source” of the message:

  MPI_SOURCE field of the struct

  “tag” of the message:

  MPI_TAG field of the struct

  „error code“ of the receive operation:

  MPI_ERROR field of the struct

  “size” of the message:

  Next slide!

MPI – Part 1: Gathering information


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

- Query if a message from source with tag is available
- Message is not received
- Information about the matched message is stored in the status field
- E.g.: Is there any message?

```
MPI_Probe(MPI_ANY_SOURCE,MPI_ANY_TAG,
        MPI_COMM_WORLD, &status)
```
C/C++

- Note: MPI_Probe is blocking!

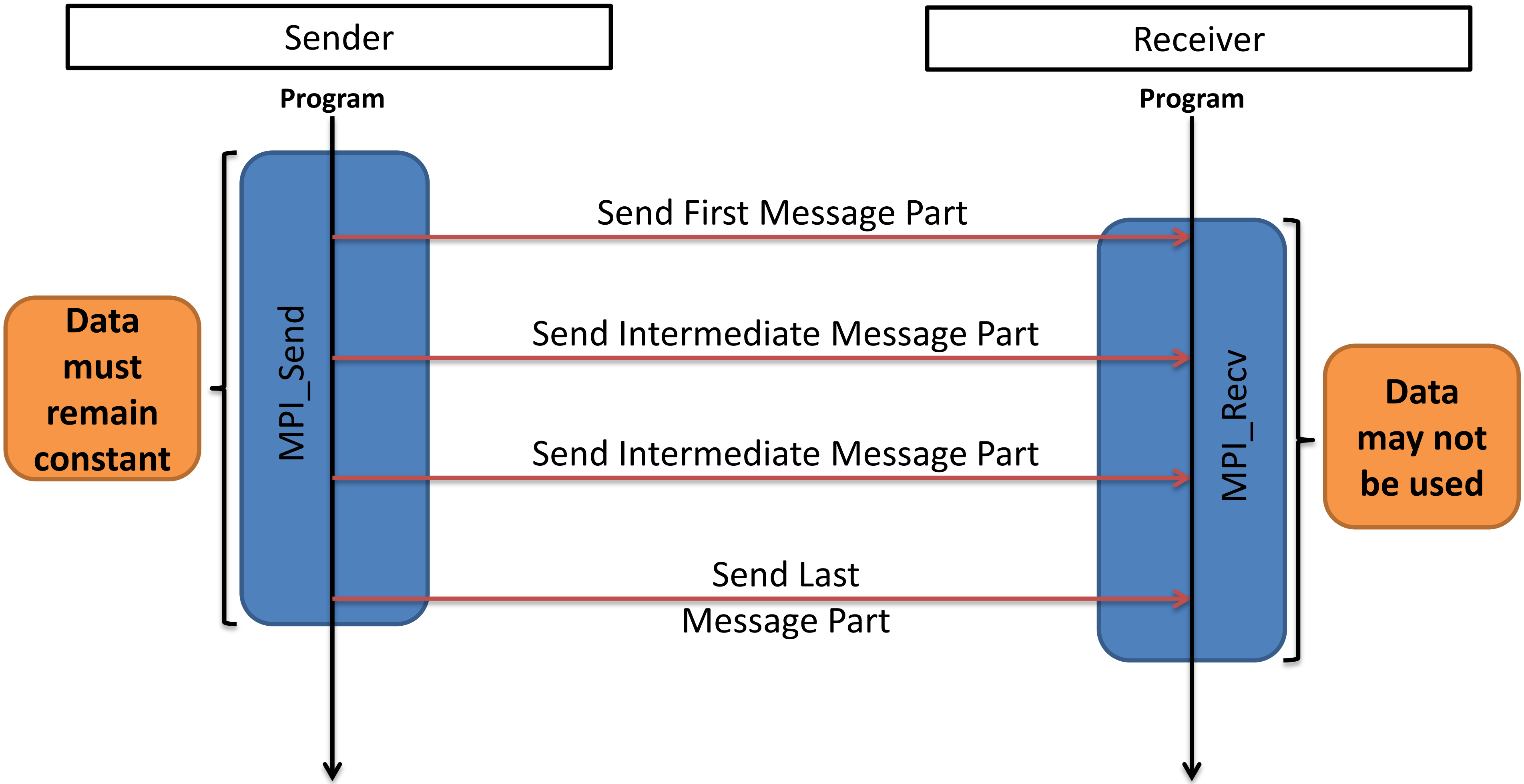
```
MPI_Get_count(MPI_Status * status, MPI_Datatype datatype, int * count)
```
C/C++

- Compute how many elements of datatype could be formed from the (unreceived) message referenced by status

MPI – Part 1: Non-Blocking communication (1 / 6)

TECHNISCHE UNIVERSITÄT DARMSTADT

- **(Blocking) MPI_Send / MPI_Recv:**



- MPI_Isend / MPI_Irecv: start an nonblocking asynchronous communication

```
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: Handle to reference the started communication

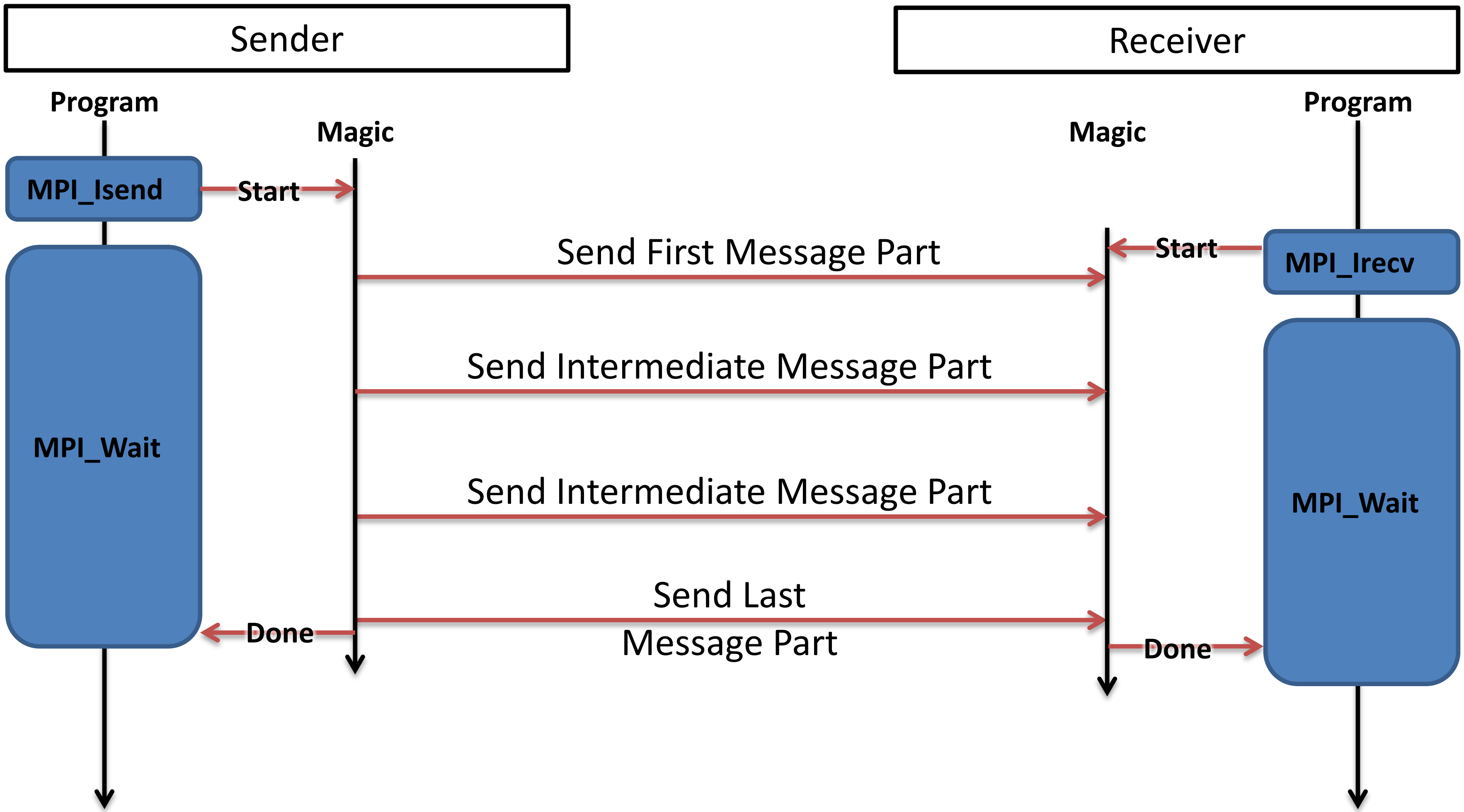
- MPI_Wait: wait for the referenced communication operation to finish

```
MPI_Wait (MPI_Request * request, MPI_Status * status)
```
C/C++

- request: Handle to a previously started communication
- status: filled by MPI_Wait with the status of the completed communication

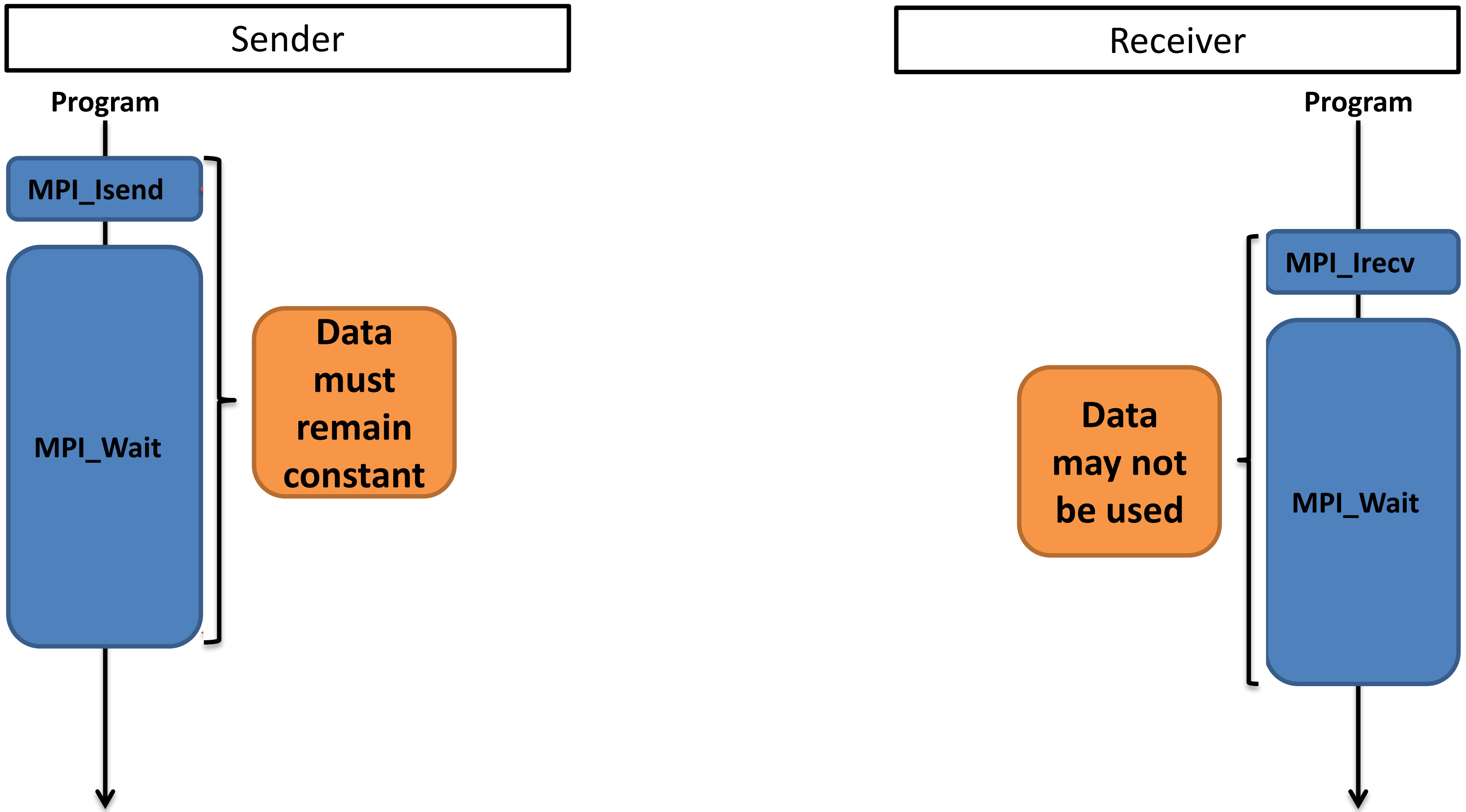
TECHNISCHE UNIVERSITÄT DARMSTADT

MPI_Isend / MPI_Irecv/MPI_Wait:



TECHNISCHE UNIVERSITÄT DARMSTADT

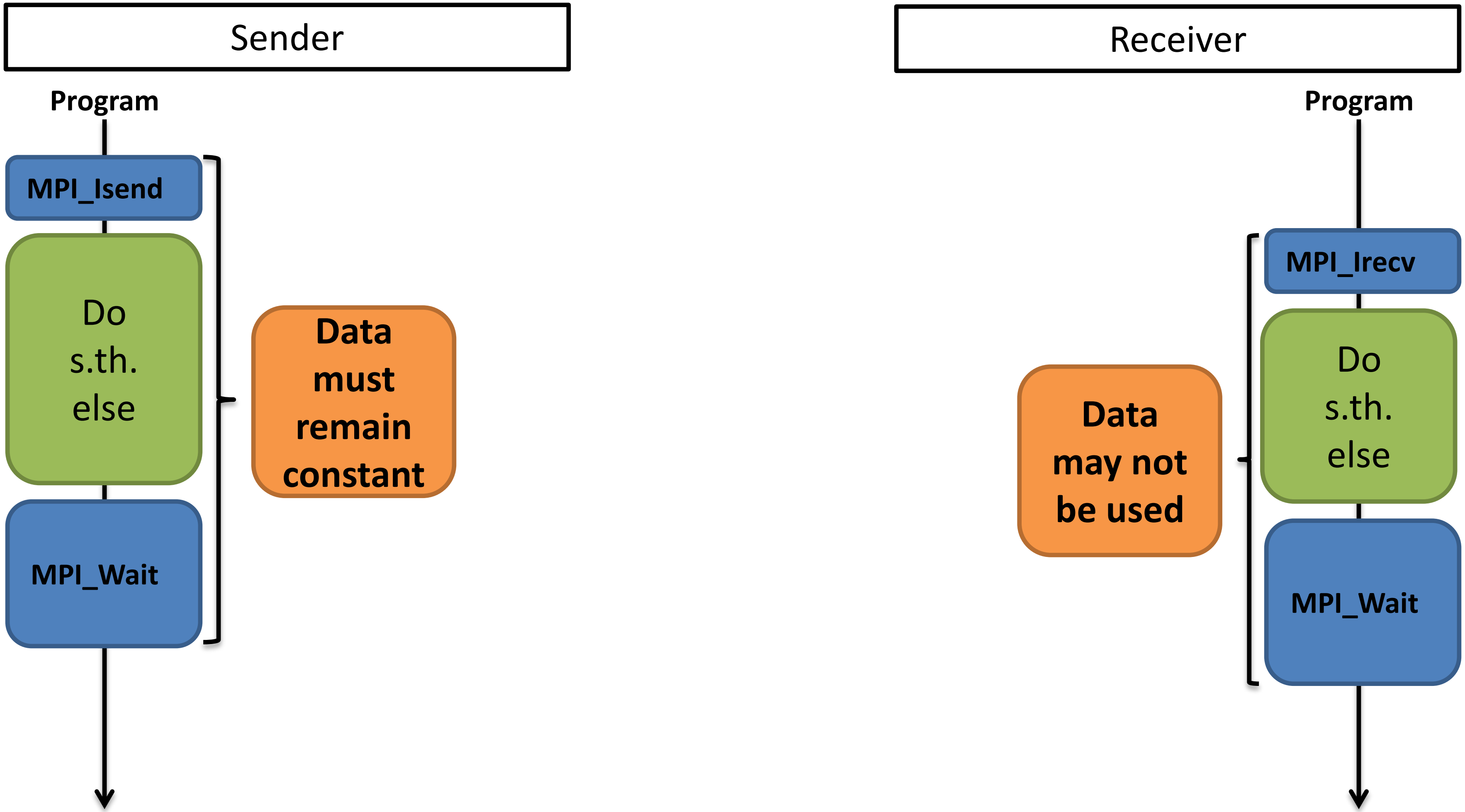
MPI_Isend / MPI_Irecv/MPI_Wait:



MPI – Part 1: Non-Blocking communication (4 / 6)

TECHNISCHE UNIVERSITÄT DARMSTADT

MPI_Isend / MPI_Irecv/MPI_Wait:



MPI – Part 1: Non-Blocking communication (5 / 6)

- Testing and waiting
- MPI_TEST: test if given request has finished without blocking

```
MPI_Test(MPI_Request * reqest, int  * flag, MPI_Status * status)
```
C/C++

- flag: is set to true is request has finished otherwise not
- status: contains status of operation if test was successfull
- Note: the request is considered completed and set to MPI_REQUEST_NULL

- MPI_REQUEST_GET_STATUS: similar to MPI_TEST, but does not complete the request

```
MPI_Request_get_status(MPI_Request * reqest, int  * flag,
          MPI_Status * status)
```
C/C++

MPI – Part 1:
Non-Blocking communication (6 / 6)

Other test and wait functions

- MPI_Request_free
  Free a request handle, communication will complete

- MPI_Wait_any
  Wait for multiple requests, block until one completes
- MPI_Test_any
  Test for multiple requests without blocking, only one request is free

- MPI_Wait_all
  Wait for all requests

- MPI_Wait_some
  Wait for one or more requests, all completed requests are freed
- MPI_Test_some
  Test for one or more request, all completed requests are freed

MPI – Part 1: Communication modes

- **MPI supports 4 send modes:**
  - Standard
  - Buffered
  - Synchronous
  - Ready

- **Note: Only 1 receive mode**

MPI – Part 1: Send modes (1/2)

TECHNISCHE UNIVERSITÄT DARMSTADT

- **Normal Blocking Send Mode:**

  The calls blocks until the message-data and envelope have been copied to internal buffers or been used to send a message

- **Buffered Send Mode:**

  Letter Prefix: B

  The call blocks until the envelope and all of the message-data has been copied to an MPI-internal buffer.

  Transmission occurs at a potential later point

- **Synchronous Send Mode:**

  Letter Prefix: S

  The call blocks until a matching receive has been called and the envelope and data has been put aside or transmitted

- **Ready Send Mode:**

  Letter Prefix: R

  May only be called if the matching receive has been posted.

  Behavior as normal send

MPI – Part 1:
Send modes (2/2)

- **Function names:**
  - MPI_Bsend
  - MPI_Ibsend
  - MPI_Ssend
  - MPI_Issend
  - MPI-Rsend
  - MPI-Irsend

- **Specify buffer for buffered send**
  - MPI_Buffer_attach
  - MPI_Buffer_detach

MPI – Part 1:
Utility functions

- **MPI_Abort: Try to abort all collaborating MPI-processes**

```
MPI_Abort(MPI_Comm comm, int errorcode)
```
C/C++

- Attempt to abort the MPI program returning errorcode if possible

MPI_Wtime:

```
double MPI_Wtime()
```
C/C++

- MPI_Wtime returns the (fraction-) number of seconds since some arbitrary point of time

- **MPI_Get_processor_name: Get a name for the current node hosting the MPI-process**

```
MPI_Get_processor_name(char * data,int * maxLen)
```
C/C++

- data: data contains a string identifying the executing host
- maxLen: variable initialized with the maximum string length

MPI – Part 1: Utility functions

- **Further utility functions:**

  MPI_Wtick

  MPI_Inizialized

  MPI_Finalized

MPI – Part 1: MPI History

- **Version 1.0 (1994): Fortran77 and C supported**
- **Version 1.1 (1995): Minor corrections and clarifications**
- **Version 1.2 (1997): Further corrections and clarifications**
- **Version 2.0 (1997): Major enhancements:**
  - One-sided communications
  - Parallel IO (p.d. add-on: ROMIO)
  - Dynamic process generation
  - Fortran 90 and C++ support
  - Thread Safety
  - Language interoperability
- **Version 2.1 (2008): Merging of MPI-1 and MPI-2**
- **Version 2.2 (2009): Minor correction and clarifications**

- **Version 3.0 (in the works):**
  - Non blocking collectives
  - Scalability issues

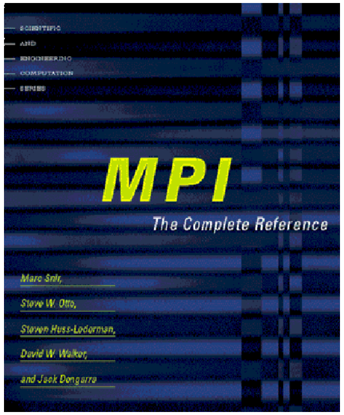
TECHNISCHE UNIVERSITÄT DARMSTADT

- **MPI. The Complete Reference Vol. 1: The MPI core. Second edition**

  by Marc Snir, Steve Otto, Steven Huss-Lederman,

  David Walker, Jack Dongarra.

  The MIT Press; 2 edition; 1998



- **MPI: The Complete Reference. Vol 2: The MPI-2 extensions**

  by William Gropp, Steven Huss-Lederman,

  Andrew Lumsdain, Ewing Lusk, Bill Nitzberg,

  William Saphir, Marc Snir

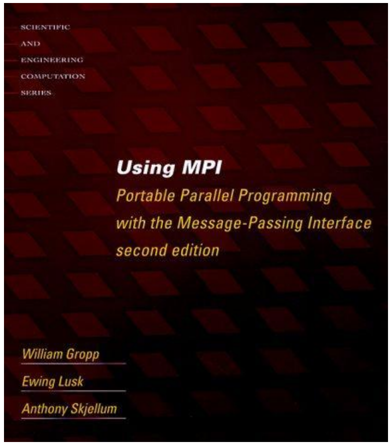
  The MIT Press; 2nd edition; 1998

MPI – Part 1: Literature (2/2)

- **Using MPI**

  by William Gropp, Ewing Lusk, Anthony Skjellum

  The MIT Press, Cambridge, London 1999



- **Using MPI-2**

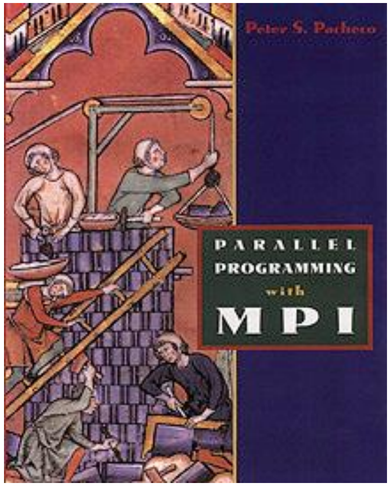
  William Gropp, Ewing Lusk, Rajeev Thakur

  The MIT Press, Cambridge, London 2000

- **Parallel Programming With MPI**

  by Peter Pacheco

  440 Seiten - Morgan Kaufmann Publishers



MPI – Part 1: More MPI Information & Documentation

- **The MPI Forum**
  - http://www.mpi-forum.org/
- **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 web page with further links**
  - http://www.rz.rwth-aachen.de/mpi/
- **Manual pages**
  - man MPI
  - man mprun
  - man MPI_Xxxx (for all MPI calls)

UP NEXT:
MPI PART 2