

PARALLEL PROGRAMMING WITH MPI

Caspar van Leewen
Carlos Teijeiro Barjas

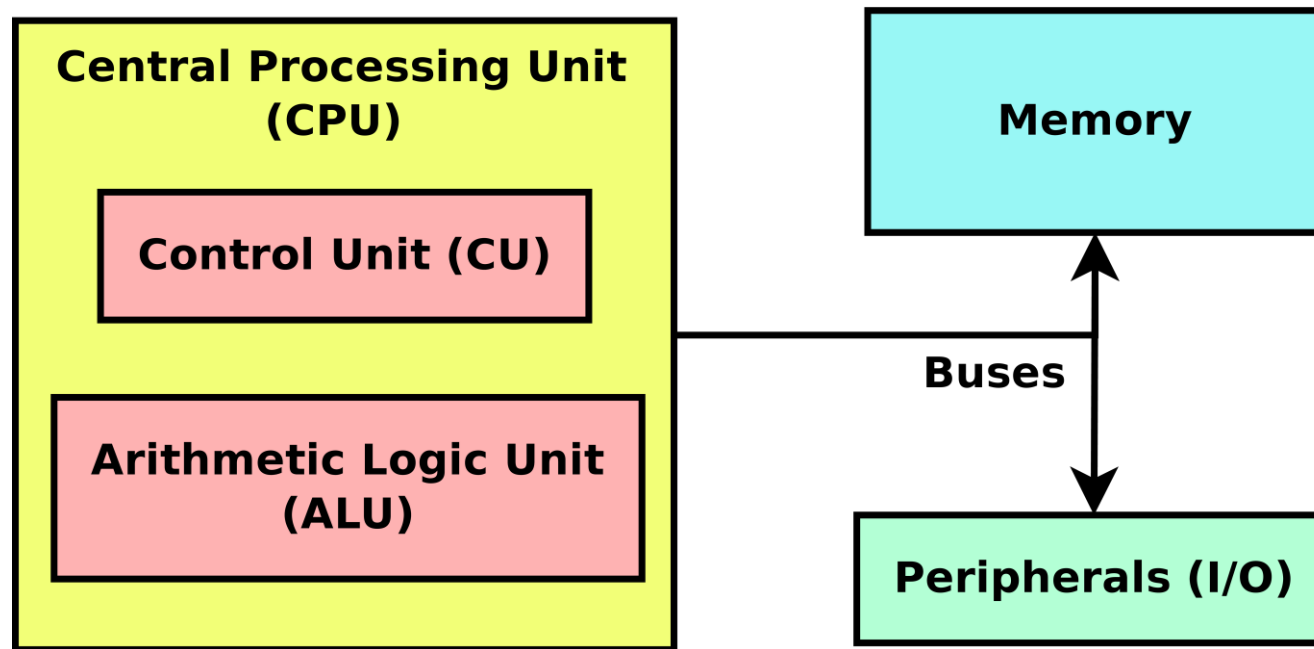
Goals

- The goal is to learn:
 - how an MPI program runs on a distributed memory system
 - how to write MPI programs
 - how to use basic point-to-point and collective MPI communication
 - how topologies can be defined
 - some useful examples for further work with MPI

Content

- MPI and distributed memory parallelism
- Structure of the library functions
- Different hands-on exercises

Recap: a computer is...



Recap: a CPU is...

Example of the MIPS architecture: control unit lines in orange, data lines in black

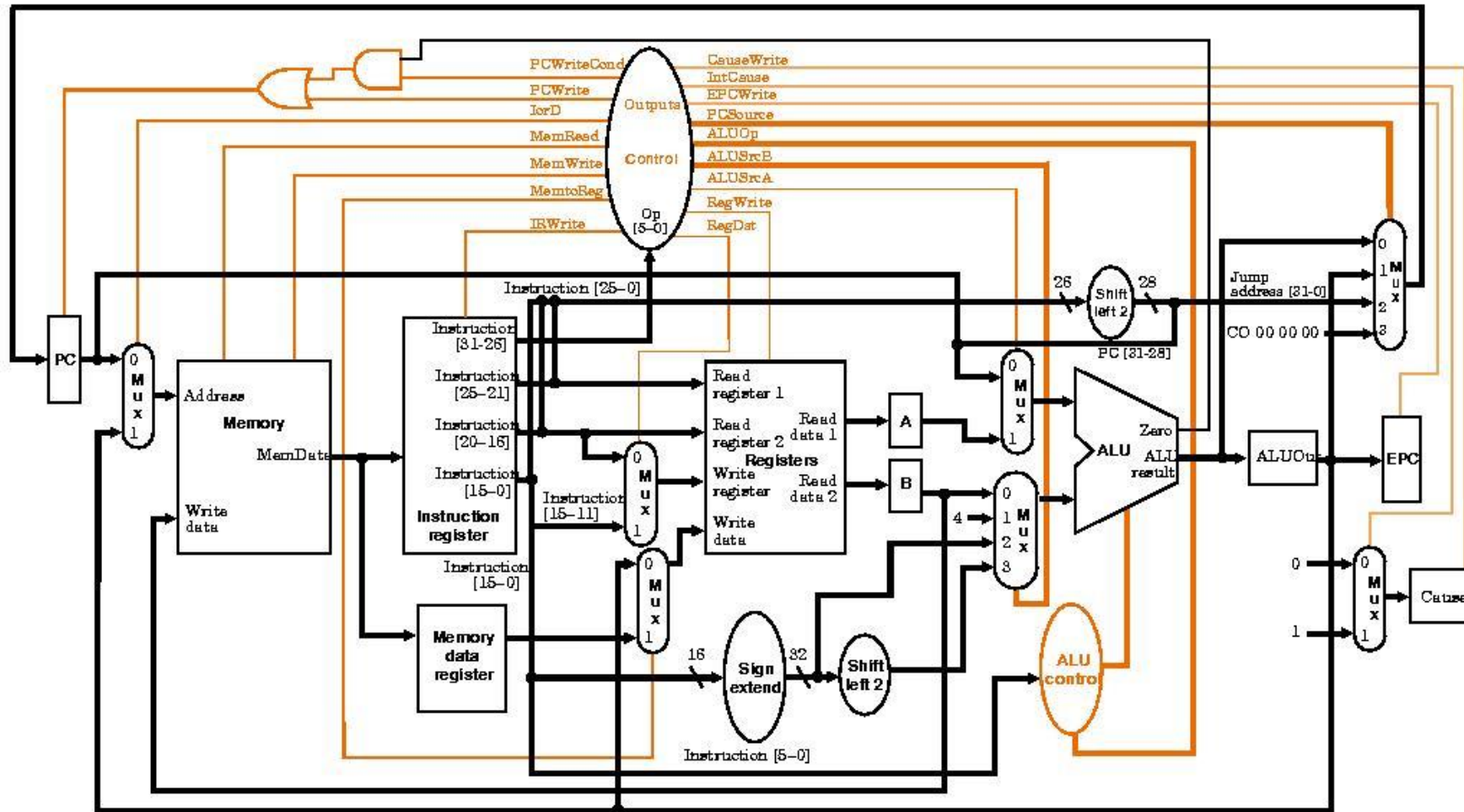
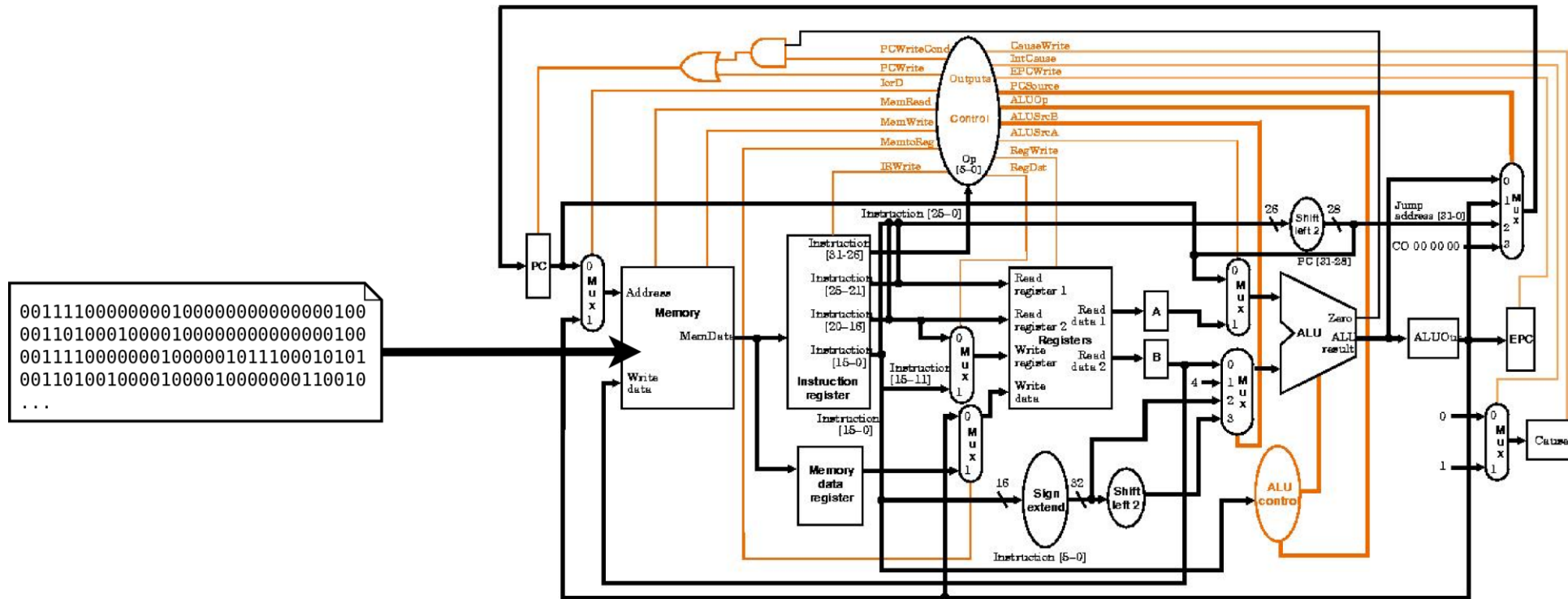
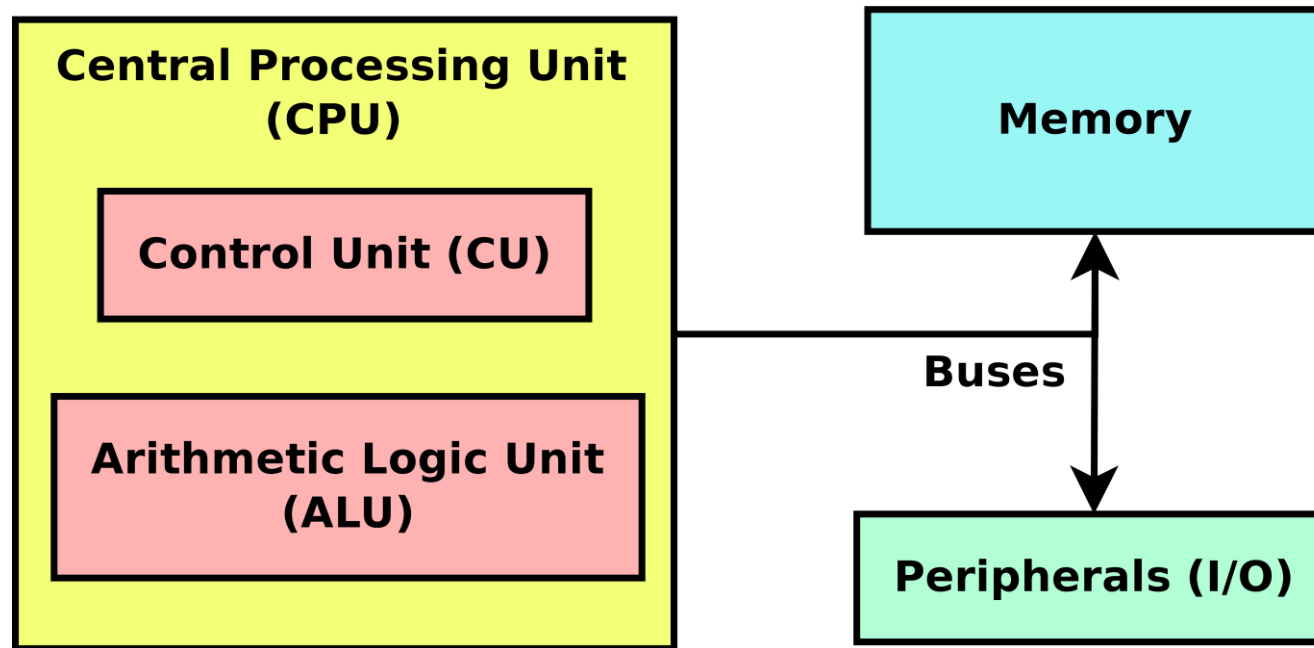


Image source: <http://people.cs.pitt.edu/~don/coe1502/current/Unit4a/Unit4a.html> (last visited: 2018)

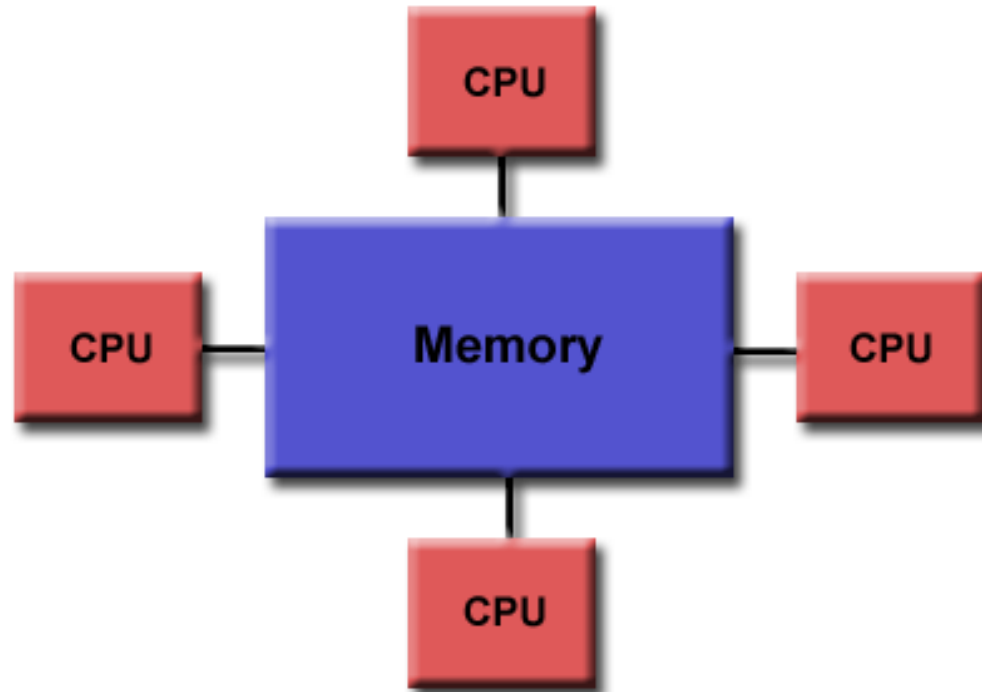
Recap: a program becomes a process...



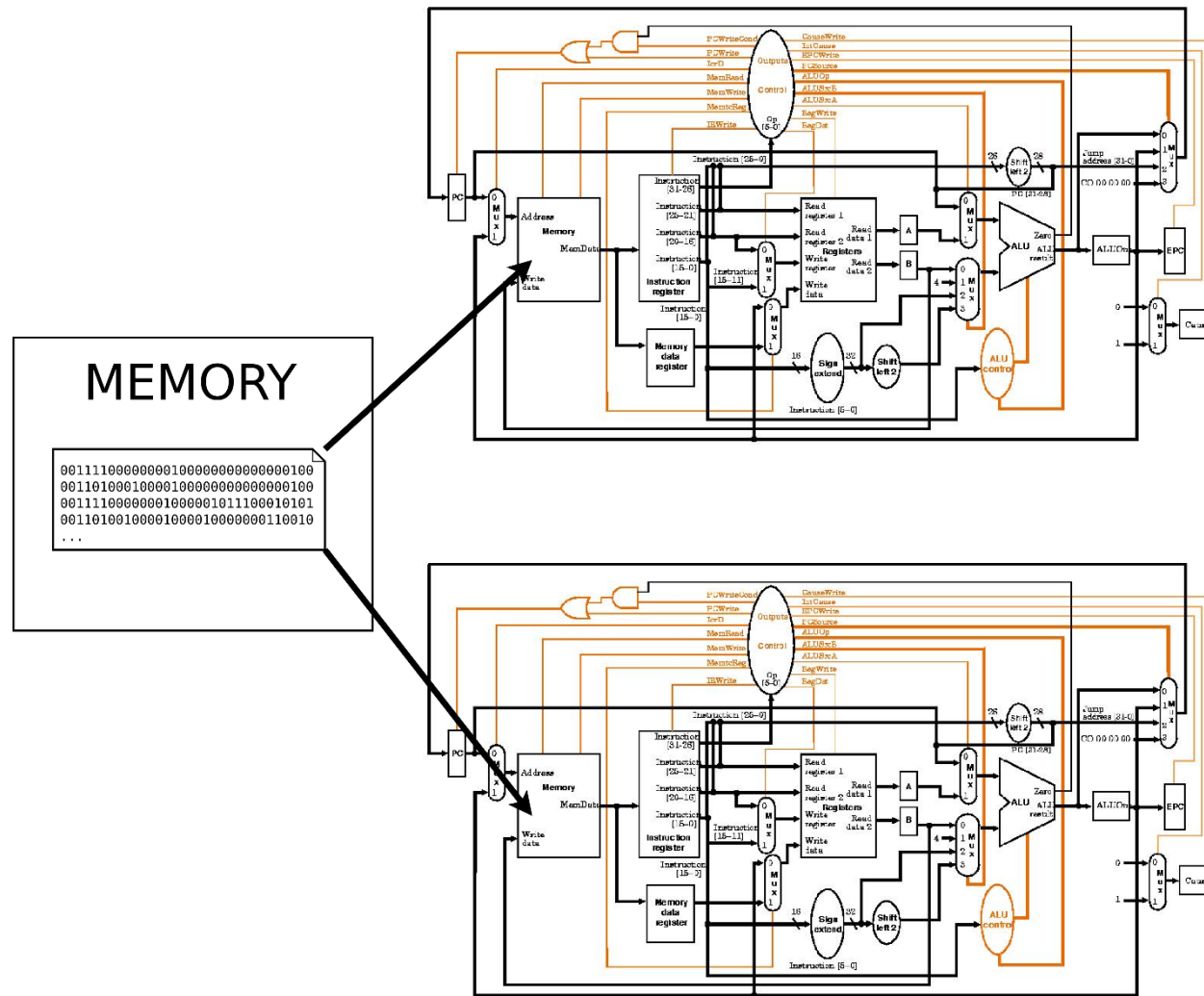
Recap: different types of computer architectures



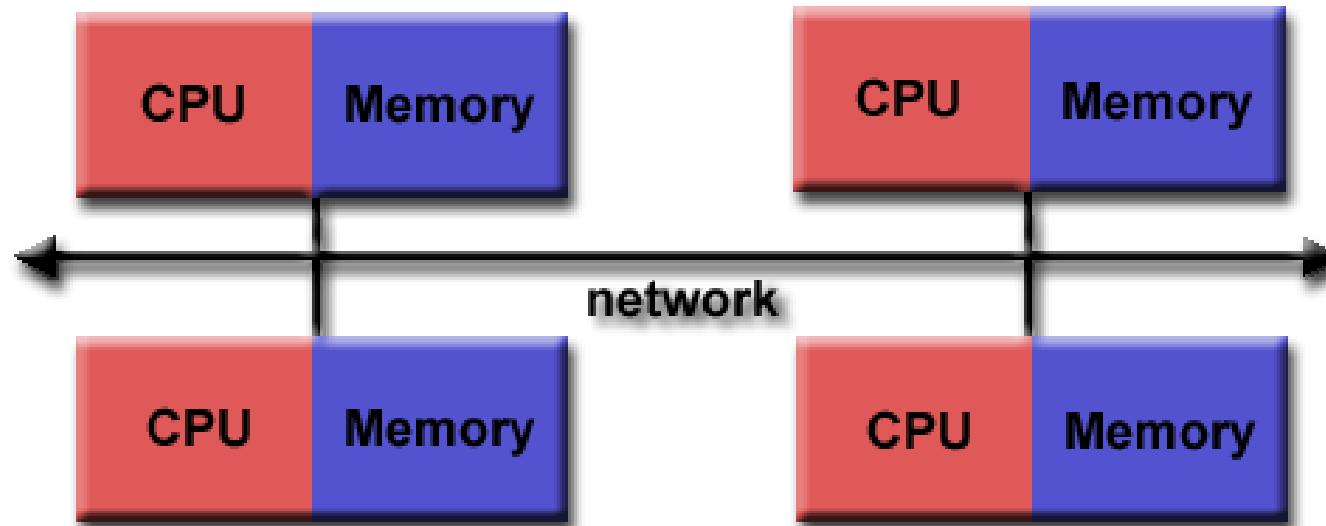
Recap: different types of computer architectures



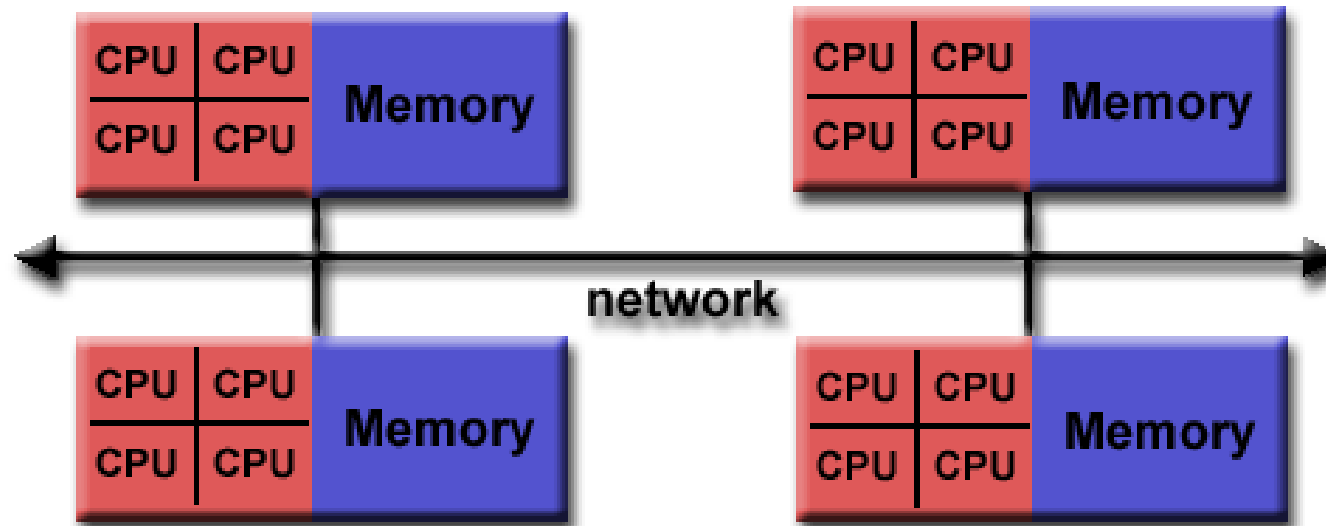
Recap: different types of computer architectures



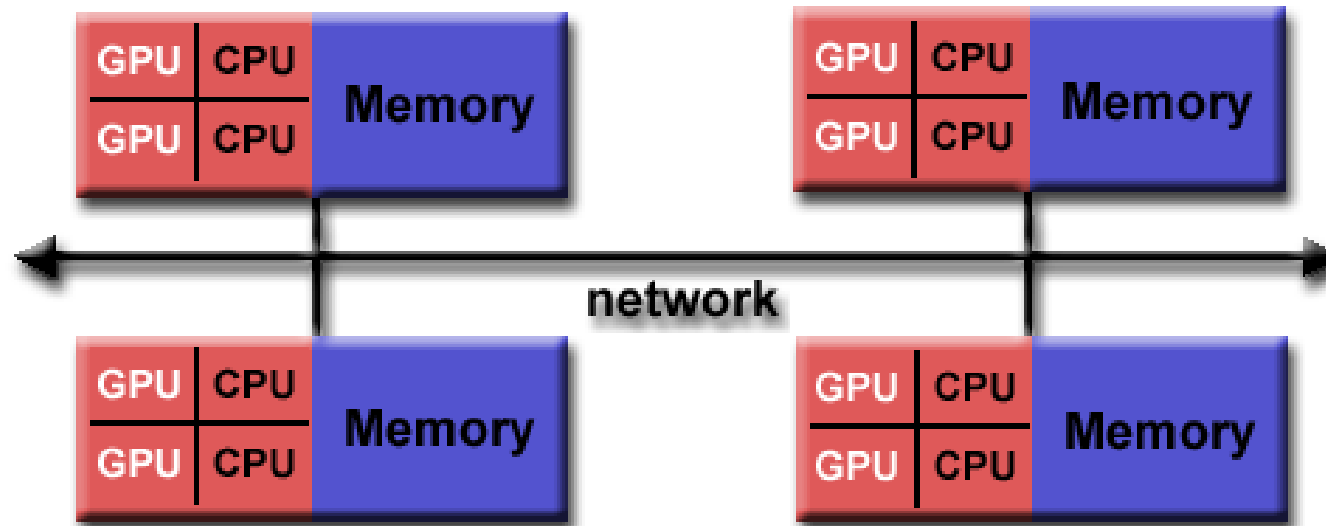
Different types of computer architectures



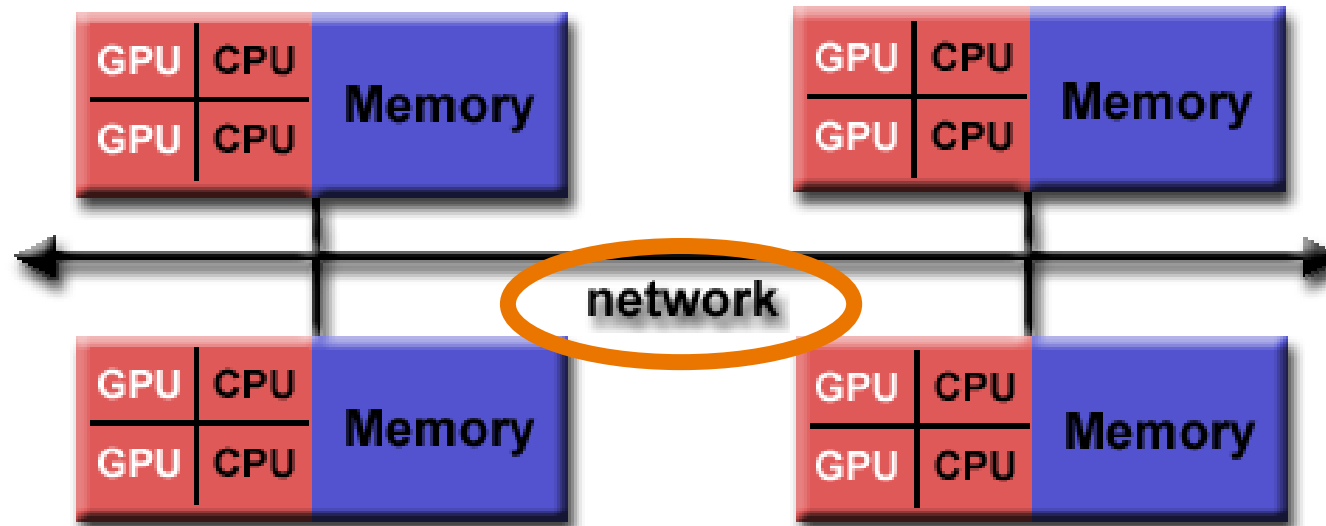
Different types of computer architectures



Different types of computer architectures

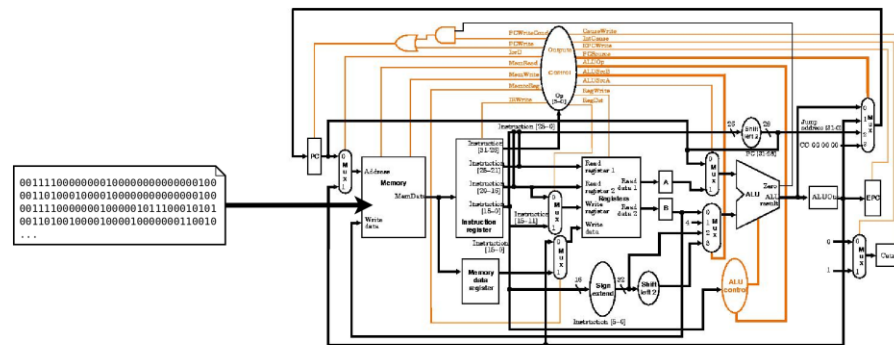
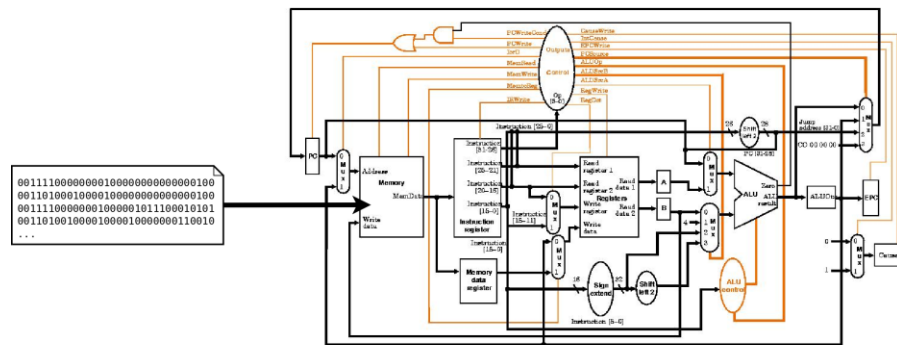
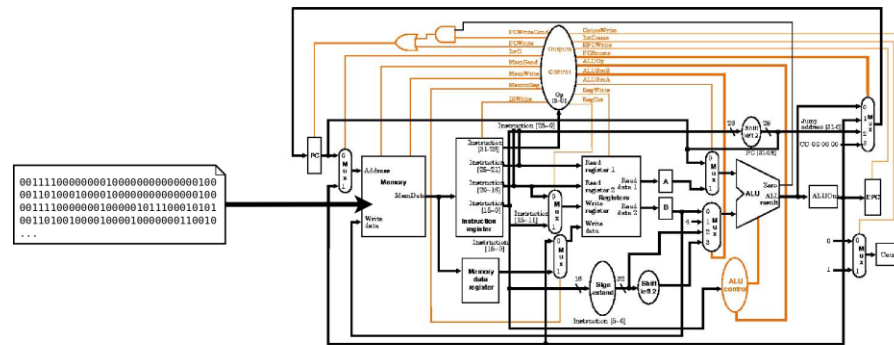
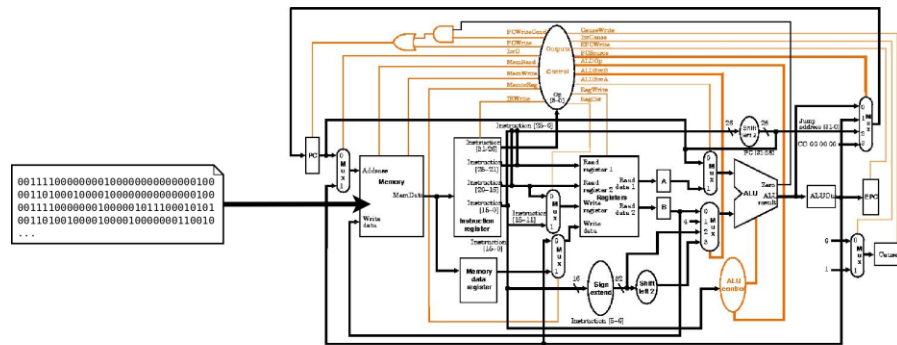


Different types of computer architectures



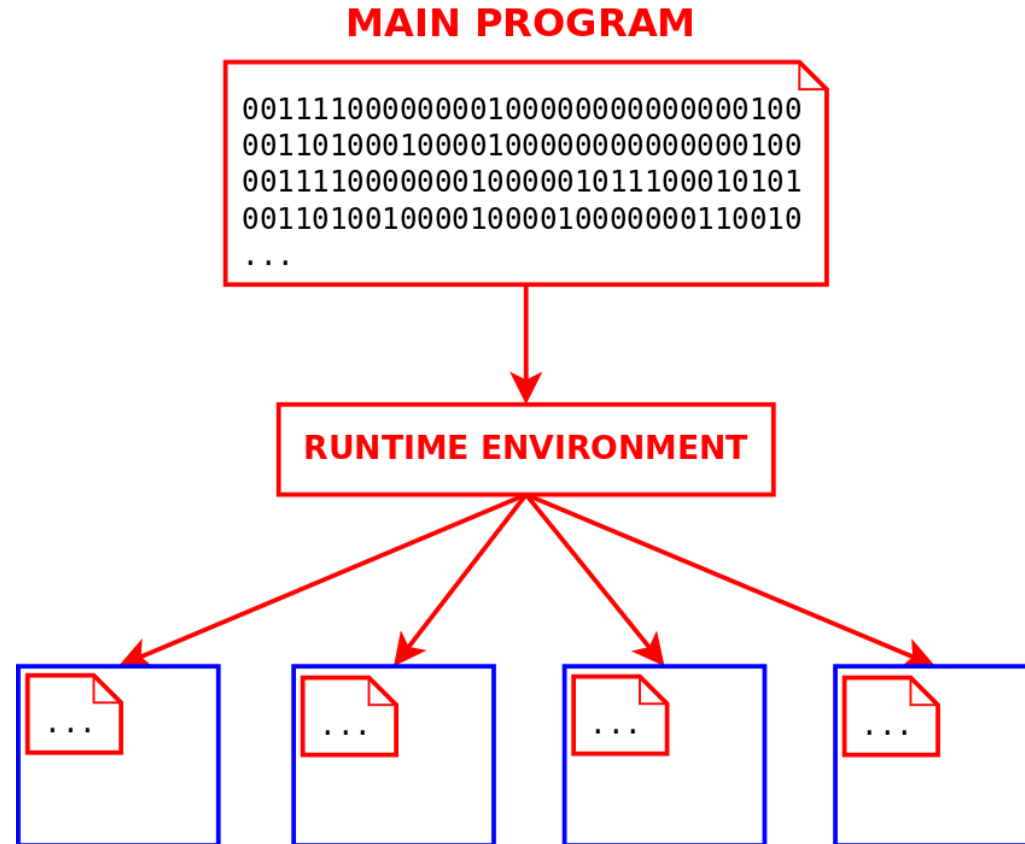
Different types of computer architectures

- Basic execution model: SPMD (Single Program, Multiple Data)
- Even MPMD (Multiple Program, Multiple Data) could be possible...



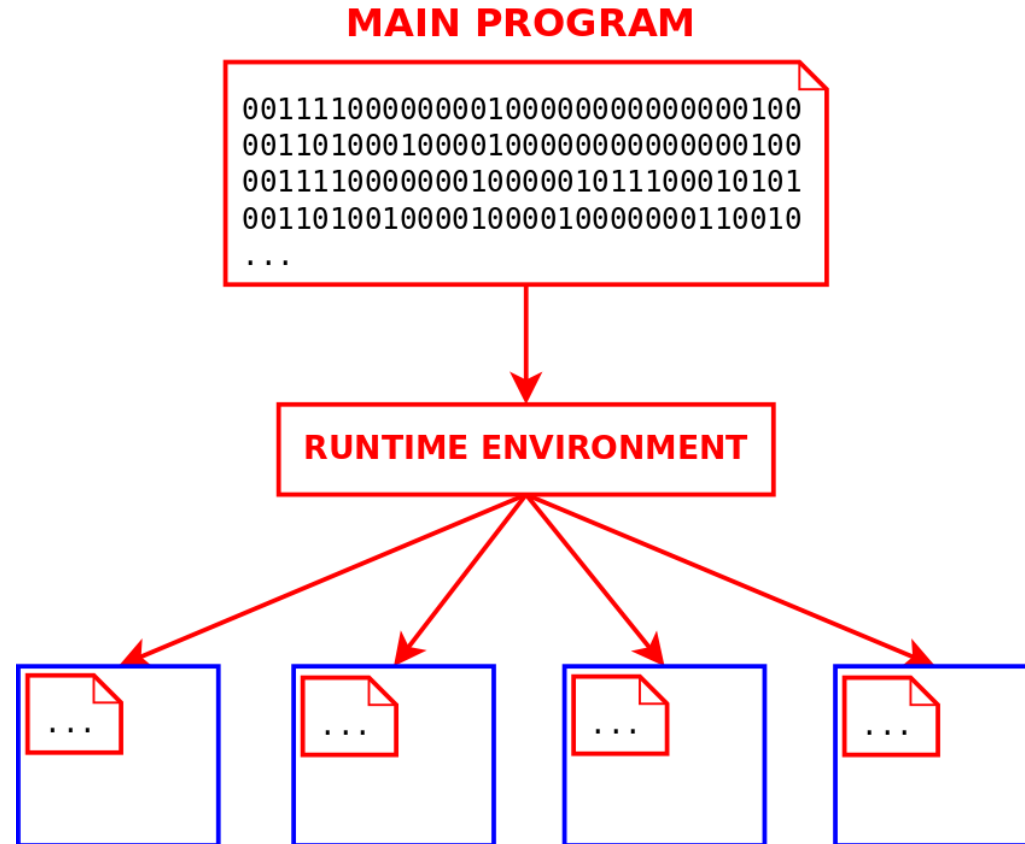
Many processes in different computers: MPI

- **MPI: Message Passing Interface**
- Different processes are created on the one or many machines (nodes)
- Each process executes the same program and has a private copy of all variables
- Inter-process communication by sending/receiving messages and synchronizations
- Bindings for several languages, like C/C++, Fortran, Java or Python



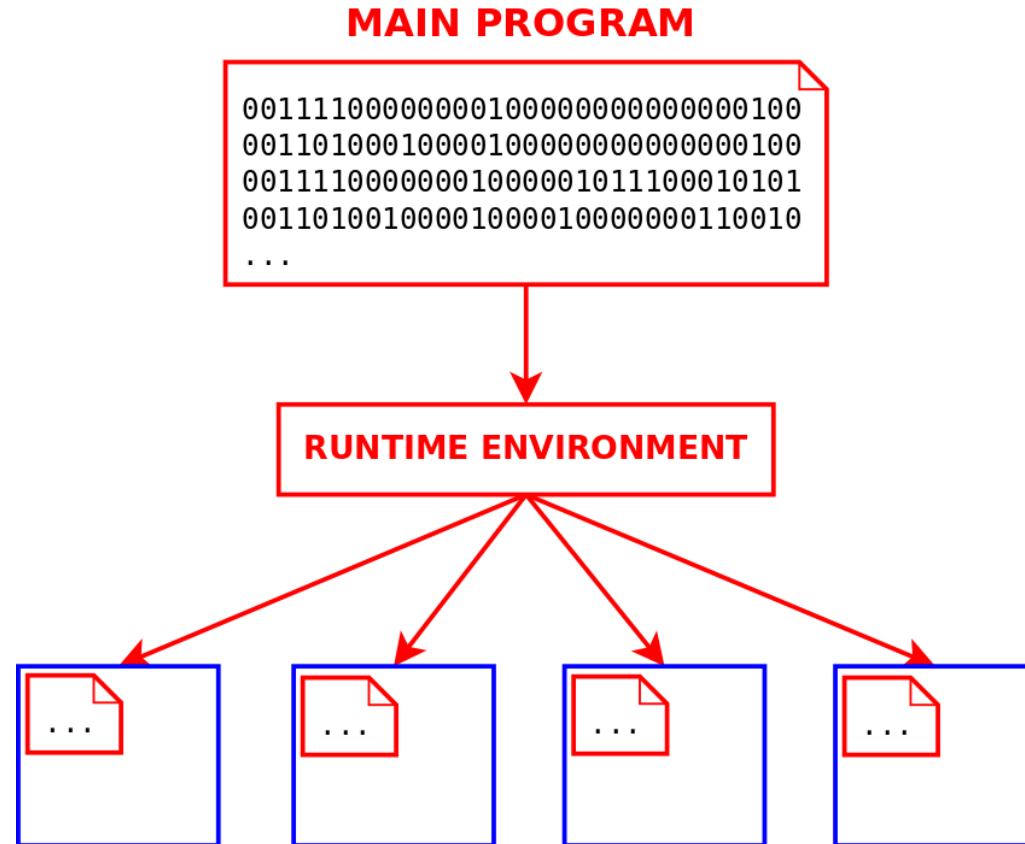
Many processes in different computers: MPI

- The interface defines the protocol and the semantic specifications of its functional parts
- Different implementations of the interface can exist
 - IntelMPI
 - OpenMPI
 - MPICH / MVAPICH
- De-facto standard for parallel programming on distributed HPC architectures



Many processes in different computers: MPI

- In general, MPI requires specific compilation (e.g. mpicc) and specific runtime environment (e.g. mpirun)
- The MPI compiler loads the necessary libraries from the selected MPI implementation
- The MPI runtime is the process that manages communications
 - Internally implemented with daemon processes



Example of multi-process execution with MPI

```
top - 11:05:19 up 41 days, 15:15, 1 user, load average: 0,52, 0,26, 0,16
Tasks: 473 total, 5 running, 468 sleeping, 0 stopped, 0 zombie
%Cpu(s): 0,0 us, 0,1 sy, 16,7 ni, 83,2 id, 0,0 wa, 0,0 hi, 0,0 si, 0,0 st
KiB Mem : 49284800 total, 35059112 free, 784368 used, 13441320 buff/cache
KiB Swap: 25165820 total, 25136612 free, 29208 used. 46491000 avail Mem
```

| PID | USER | PR | NI | VIRT | RES | SHR | S | %CPU | %MEM | TIME+ | COMMAND |
|-------|---------|----|----|--------|------|------|---|-------|------|----------|--------------|
| 28875 | casparl | 39 | 19 | 419216 | 6880 | 4240 | R | 100,0 | 0,0 | 0:04.27 | my_mpi_prog |
| 28876 | casparl | 39 | 19 | 419220 | 6876 | 4236 | R | 100,0 | 0,0 | 0:04.27 | my_mpi_prog |
| 28877 | casparl | 39 | 19 | 419220 | 6868 | 4240 | R | 100,0 | 0,0 | 0:04.26 | my_mpi_prog |
| 28878 | casparl | 39 | 19 | 419216 | 6876 | 4236 | R | 100,0 | 0,0 | 0:04.26 | my_mpi_prog |
| 3356 | root | 20 | 0 | 15944 | 1820 | 956 | S | 1,7 | 0,0 | 53:35.03 | pim |
| 477 | root | 20 | 0 | 0 | 0 | 0 | S | .3 | 0,0 | 59:26.22 | xfsaild/sda3 |

4 processes, each using a
single CPU core

Example of multi-process execution with MPI

- Note the difference:
 - OpenMP: 1 process, 4 threads per process

| PID | USER | PR | NI | VIRT | RES | SHR | S | %CPU | %MEM | TIME+ | COMMAND |
|-------|---------|----|----|--------|------|------|---|-------|------|----------|---------------|
| 31648 | casparl | 39 | 19 | 228816 | 1656 | 1180 | R | 400,0 | 0,0 | 0:22.37 | my_omp_prog |
| 84 | root | rt | 0 | 0 | 0 | 0 | S | 0,3 | 0,0 | 0:08.41 | migration/15 |
| 477 | root | 20 | 0 | 0 | 0 | 0 | S | 0,3 | 0,0 | 59:26.57 | xfssaild/sda3 |

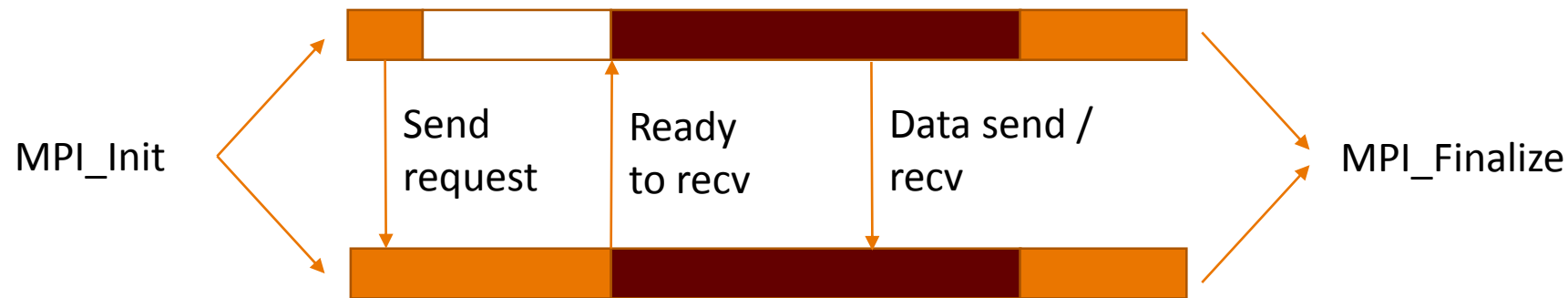
- MPI: 4 processes, 1 thread per process

| PID | USER | PR | NI | VIRT | RES | SHR | S | %CPU | %MEM | TIME+ | COMMAND |
|-------|---------|----|----|--------|------|------|---|-------|------|----------|---------------|
| 28875 | casparl | 39 | 19 | 419216 | 6880 | 4240 | R | 100,0 | 0,0 | 0:04.27 | my_mpi_prog |
| 28876 | casparl | 39 | 19 | 419220 | 6876 | 4236 | R | 100,0 | 0,0 | 0:04.27 | my_mpi_prog |
| 28877 | casparl | 39 | 19 | 419220 | 6868 | 4240 | R | 100,0 | 0,0 | 0:04.26 | my_mpi_prog |
| 28878 | casparl | 39 | 19 | 419216 | 6876 | 4236 | R | 100,0 | 0,0 | 0:04.26 | my_mpi_prog |
| 3356 | root | 20 | 0 | 15944 | 1820 | 956 | S | 1,7 | 0,0 | 53:35.03 | pim |
| 477 | root | 20 | 0 | 0 | 0 | 0 | S | 0,3 | 0,0 | 59:26.22 | xfssaild/sda3 |

- ... and you may even combine MPI + OpenMP...

Many processes in different computers: MPI

- The use of MPI requires some programming efforts to manage all processes coordinately
- In the beginning the MPI environment is initialized, and also finalized at the end of the parallel code
- The basic element is the communicator, which defines
 - communication context
 - set of processes that can communicate between each other inside that context



Many processes in different computers: MPI

- The use of MPI requires some programming efforts to manage all processes coordinately
- In the beginning the MPI environment is initialized, and also finalized at the end of the parallel code
- The basic element is the communicator, which defines
 - communication context
 - set of processes that can communicate between each other inside that context

| | MPI_COMM_WORLD | My_communicator |
|-----------|----------------|-----------------|
| Process 0 | Rank 0 | |
| Process 1 | Rank 1 | Rank 0 |
| Process 2 | Rank 2 | Rank 1 |

General references for MPI

- Specification documents in the web page of the [MPI Forum](#)
- Current specification document: [version 3.1](#)
- Revisions and discussions for the elaboration of the MPI standard 4.0 are ongoing

Basic MPI structure and predefined handles

- MPI functions are used inside the code and require including a file or module
 - C/C++: `#include <mpi.h>`
 - Fortran: `use mpi` (with Fortran 2008, definitely do “`use mpi_f08`”)
- Essential structure for MPI routines
 - C/C++: `error = MPI_Routine(arg0, arg1, ...);`
(the error output may be ignored, but it is very useful for debugging)
 - Fortran: `call MPI_ROUTINE(arg1, arg2, ..., ierror)`
(the error output is always the last argument: VERY IMPORTANT!!!)
- Global communicator: predefined handle
 - C/C++ AND Fortran: `MPI_COMM_WORLD` (type “`MPI_Comm`” or “`integer`”)

Basic MPI structure and predefined handles

- Every MPI code begins with an initialization of the environment
 - C/C++: `int MPI_Init(int *argc, char ***argv)`
typical call: `MPI_Init(&argc, &argv);`
argc and *argv* are taken from the main program, NULL is possible too
 - Fortran: `subroutine MPI_INIT(ierr)` `integer :: ierr`
typical call: `call MPI_Init(ierr)`
- Every MPI code ends with a finalize statement that stops the environment
 - C/C++: `int MPI_Finalize()`
 - Fortran: `subroutine MPI_FINALIZE(ierr)` `integer :: ierr`

Basic MPI structure and predefined handles

- MPI has different handles that identify specific objects
- Most common handle: predefined constants
 - Defined in `mpi.h` (C/C++) or in the `mpi/mpi_f08` modules (Fortran)
 - E.g. global communicator `MPI_COMM_WORLD`
- Other handles: type definitions for MPI specific variables or error return codes
 - E.g. communicator types
 - C/C++: `MPI_Comm my_communicator`
 - Fortran: `integer :: my_communicator`
- with `mpi_f08`: `TYPE(MPI_Comm)`

Basic MPI structure and predefined handles

- Every MPI process is identified with a number, called “rank”

- C/C++: `int MPI_Comm_rank(MPI_Comm comm, int *rank)`



MPI Communicator



Output argument with size value

- Fortran: `subroutine MPI_COMM_RANK(comm, rank, ierr)`



“use mpi”: `integer :: comm, rank, ierr`

“use mpi_f08”: `TYPE(MPI_Comm) :: comm`

`integer :: rank; integer, optional :: ierr`



ierr argument is optional with the module mpi_f08.

Try to use this module when possible!

ierr argument...
NEVER FORGET
when using the
mpi module!!!

Basic MPI structure and predefined handles

- The number of processes in a communicator can be obtained with another routine

- C/C++: `int MPI_Comm_size(MPI_Comm comm, int *size)`


 MPI Communicator  Output argument with size value


- Fortran: `subroutine MPI_COMM_SIZE(comm, size, ierr)`

“use mpi”: `integer :: comm, rank, ierr`

“use mpi_f08”: `TYPE(MPI_Comm) :: comm`

`integer :: rank; integer, optional :: ierr`

 ierr argument is optional with the module mpi_f08.
Try to use this module when possible!

 ierr argument...
NEVER FORGET
when using the
mpi module!!!

Ready for a first hands-on exercise?

- Go to the GitHub web page of the course

<https://github.com/sara-nl/PRACE-MPI-OpenMP>

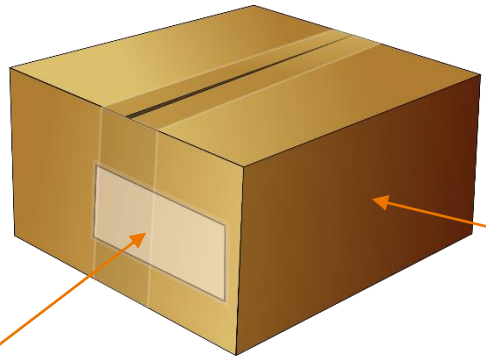
- Try to write your first MPI code!
 - C/C++: `mpi_hello_world.c`
 - Fortran: `mpi_hello_world.f`

Some ideas about MPI execution model

- The execution environment (e.g. mpirun, srun...) creates the requested processes
- After executing MPI_Init, the processes are located in the same communication environment: MPI_COMM_WORLD
- Before MPI_Init and after MPI_Finalize the processes still exist, but there is no possible interaction or coordination between them
- MPI follows the SPMD (Single Program, Multiple Data) execution model
 - All processes execute exactly the same program
 - The programmer has the responsibility of making processes cooperate
- General way of cooperation between processes
 - Manual distribution of data and/or tasks
 - Message exchanges (send / receive)

MPI point-to-point communication: messages

- MPI_Send() and MPI_Recv()
 - In order to send and receive it is necessary to create a “parcel”
 - Need for different information: mostly contents and “address label”



1. Source/Destination address
2. Tag
3. Communicator (MPI_COMM_WORLD)

1. Variable(s) to send/receive
2. How many variables to send
3. The size of one variable

MPI point-to-point communication: messages

- A message is any type of data that is communicated from one MPI process to another (or many) process(es)
- MPI datatypes can be *basic* or *derived*
 - Basic types are integer, float, double precision, etc.
 - Derived types are usually sets of basic datatypes and/or other derived datatypes
- The naming of datatypes varies with the programming language

MPI point-to-point communication: messages

- Routine to send a message

- C/C++: `int MPI_Send(void *buffer, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm);`

- Fortran: `MPI_SEND(buffer, count, datatype, dest, tag, comm, ierr)`
`<datatype> buffer(*)`
`integer :: count, datatype, dest, tag, comm, ierr`

MPI point-to-point communication: messages

- Routine to receive a message

- C/C++:

```
int MPI_Recv(void *buffer, int count, MPI_Datatype datatype,  
            int dest, int tag, MPI_Comm comm, MPI_Status status);
```

- Fortran:

```
MPI_RECV(buffer, count, datatype, src, tag, comm, status, ierr)  
          <datatype> buffer(*)  
          integer :: count, datatype, src, tag, comm  
          integer :: status(MPI_STATUS_SIZE), ierr
```

MPI point-to-point communication: messages

- Send / receive: C example

```
Rank 0    int send_val = 10;  
          MPI_Send(&send_val, 1, MPI_INT, 1, 10, MPI_COMM_WORLD)
```



Variable to send

MPI point-to-point communication: messages

- Send / receive: C example

```
Rank 0    int send_val = 10;  
          MPI_Send(&send_val, 1, MPI_INT, 1, 10, MPI_COMM_WORLD)
```



#Variables to send

MPI point-to-point communication: messages

- Send / receive: C example

```
Rank 0    int send_val = 10;  
          MPI_Send(&send_val, 1, MPI_INT, 1, 10, MPI_COMM_WORLD)
```



Datatype of the variable to send

MPI point-to-point communication: messages

- Send / receive: C example

```
Rank 0      int send_val = 10;  
            MPI_Send(&send_val, 1, MPI_INT, 1, 10, MPI_COMM_WORLD)
```




Datatype of the variable to send

MPI point-to-point communication: messages

- Send / receive: C example

Rank 0 `int send_val = 10;`
`MPI_Send(&send_val, 1, MPI_INT, 1, 10, MPI_COMM_WORLD)`



Destination rank

MPI point-to-point communication: messages

- Send / receive: C example

```
Rank 0      int send_val = 10;  
            MPI_Send(&send_val, 1, MPI_INT, 1, 10, MPI_COMM_WORLD)
```



Tag of the package

MPI point-to-point communication: messages

- Send / receive: C example

```
Rank 0    int send_val = 10;  
          MPI_Send(&send_val, 1, MPI_INT, 1, 10, MPI_COMM_WORLD)
```



MPI Communicator

MPI point-to-point communication: messages

- Send / receive: C example

Rank 0 `int send_val = 10;`
 `MPI_Send(&send_val, 1, MPI_INT, 1, 10, MPI_COMM_WORLD)`

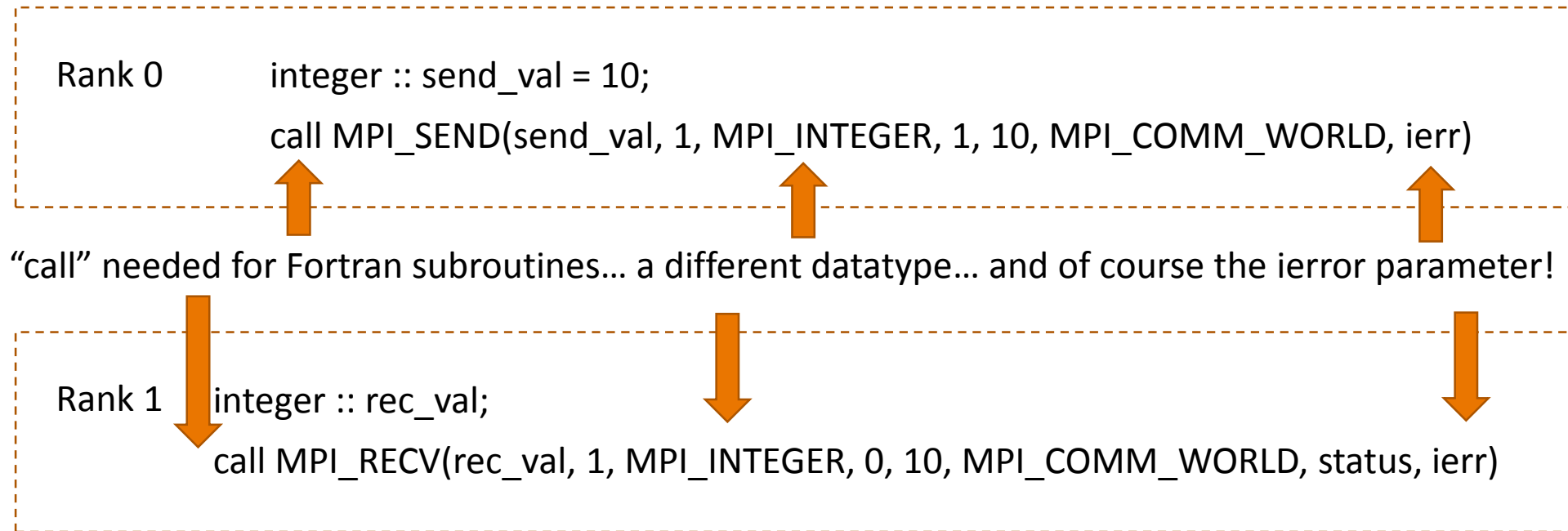
Contains info about message & sender

Rank 1 `int rec_val;`
 `MPI_Recv(&rec_val, 1, MPI_INT, 0, 10, MPI_COMM_WORLD, &status)`

Expects message of size 1 MPI_INT from rank 0 with tag 10

MPI point-to-point communication: messages

- Send / receive: Fortran example (just a couple differences)



MPI point-to-point communication: messages

| C Data Types | | Fortran Data Types | |
|--------------------------------------|--|---|--|
| MPI_CHAR | char | MPI_CHARACTER | character(1) |
| MPI_WCHAR | wchar_t - wide character | | |
| MPI_SHORT | signed short int | | |
| MPI_INT | signed int | MPI_INTEGER MPI_INTEGER1 MPI_INTEGER2 MPI_INTEGER4 | integer integer*1 integer*2 integer*4 |
| MPI_LONG | signed long int | | |
| MPI_UNSIGNED | unsigned int | | |
| MPI_FLOAT | float | MPI_REAL MPI_REAL2 MPI_REAL4 MPI_REAL8 | real real*2 real*4 real*8 |
| MPI_DOUBLE | double | MPI_DOUBLE_PRECISION | double precision |
| MPI_LONG_DOUBLE | long double | | |
| MPI_C_COMPLEX MPI_C_FLOAT_COMPLEX | float _Complex | MPI_COMPLEX | complex |
| MPI_C_DOUBLE_COMPLEX | double _Complex | MPI_DOUBLE_COMPLEX | double complex |
| MPI_C_BOOL | _Bool | MPI_LOGICAL | logical |
| MPI_BYTE | 8 binary digits | MPI_BYTE | 8 binary digits |
| MPI_PACKED | data packed or unpacked with MPI_Pack()/ MPI_Unpack | MPI_PACKED | data packed or unpacked with MPI_Pack()/ MPI_Unpack |

MPI point-to-point communication: messages

- Matching communication
 - Correct source and destination ranks inside the same communicator
 - Matching tags
 - Matching datatypes
 - ... and a sufficiently large buffer on the receiver's side
- If there are many matching communications, the order is always preserved in time
 - First come, first served
- Some status information is stored in the receiving side
 - It may be ignored by using the constant `MPI_STATUS_IGNORE` as an argument
 - ... but there is some valuable information inside

MPI point-to-point communication: messages

- Contents of status

- C/C++: `MPI_Status status`

- `status.MPI_SOURCE`

- `status.MPI_TAG`

- Fortran: `integer status(MPI_STATUS_SIZE)`

- `status(MPI_SOURCE)`

- `status(MPI_TAG)`

MPI point-to-point communication: messages

- Contents of status: getting the message count
 - C/C++: `int MPI_Get_count(MPI_Status *status,
MPI_Datatype datatype, int *count)`
 - Fortran: `MPI_GET_COUNT(status, datatype, count, ierror)`
`integer :: status(MPI_STATUS_SIZE)`
`integer :: datatype, count, ierror`
- The “get_count” functions will be very useful in some cases, in order to allocate the receive buffer in advance with enough data for a given message

Hands-on (very quick)

- Try the code on point-to-point communication
 - C/C++: `mpi_pnt2pnt.c`
 - Fortran: `mpi_pnt2pnt.f`
- Put the correct source and destination, and the necessary tags

Different modes for sending data

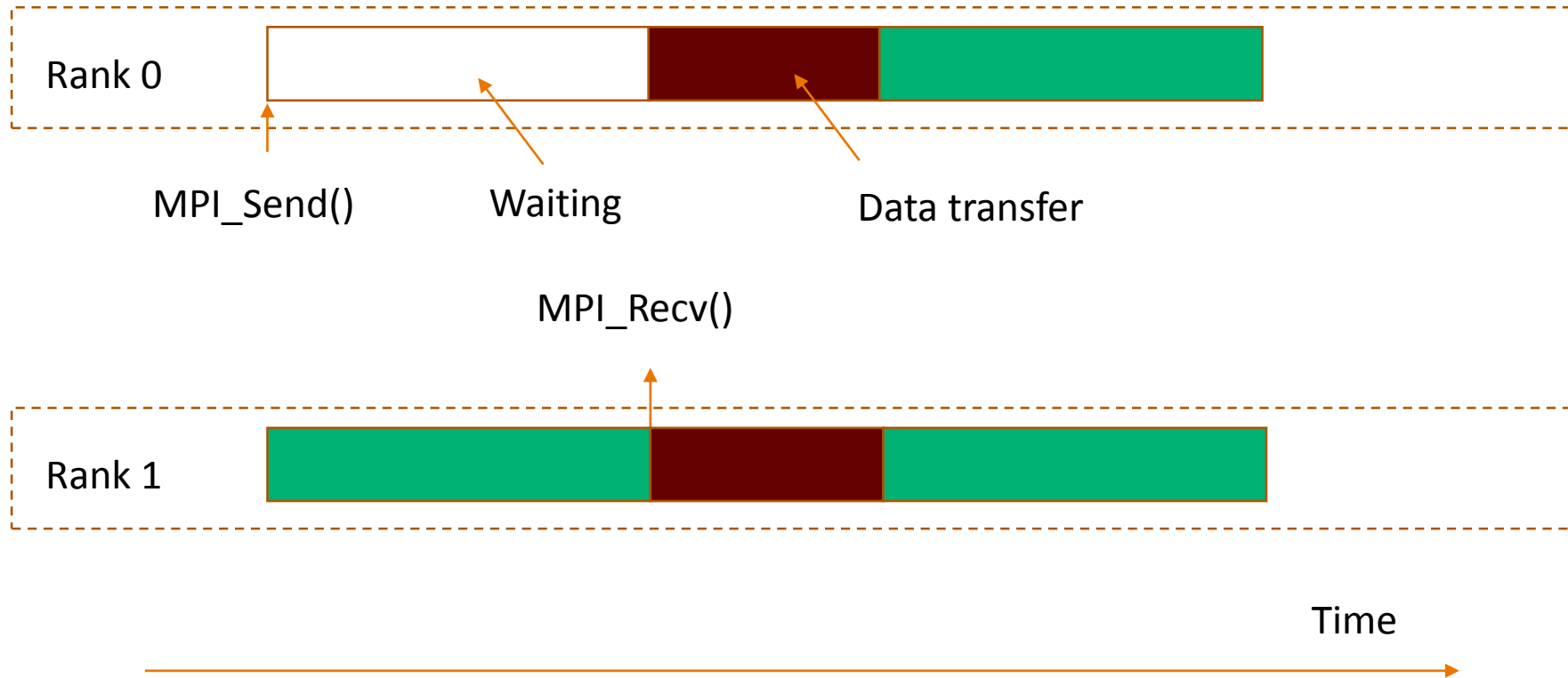
- Synchronous send (MPI_SSEND)
 - Requires synchronization between sender and receiver processes
 - It succeeds when a matching receive call is started
- Buffered send (MPI_BSEND)
 - Asynchronous: the send is always completed, even without a matching receive
 - It requires a buffer declared with MPI_BUFFER_ATTACH
- Generic send (MPI_SEND)
 - It may be either synchronous or buffered send: decision left to the MPI library
- Ready send (MPI_RSEND)
 - Performs an immediate send without checking the receiver side (better avoid...)

Only one mode for receiving data

- Generic receive (MPI_RECV)
 - Matches all possible send modes
 - It waits until a matching send is posted

MPI point-to-point communication: messages

- MPI_Send (in general) and MPI_Recv are blocking functions

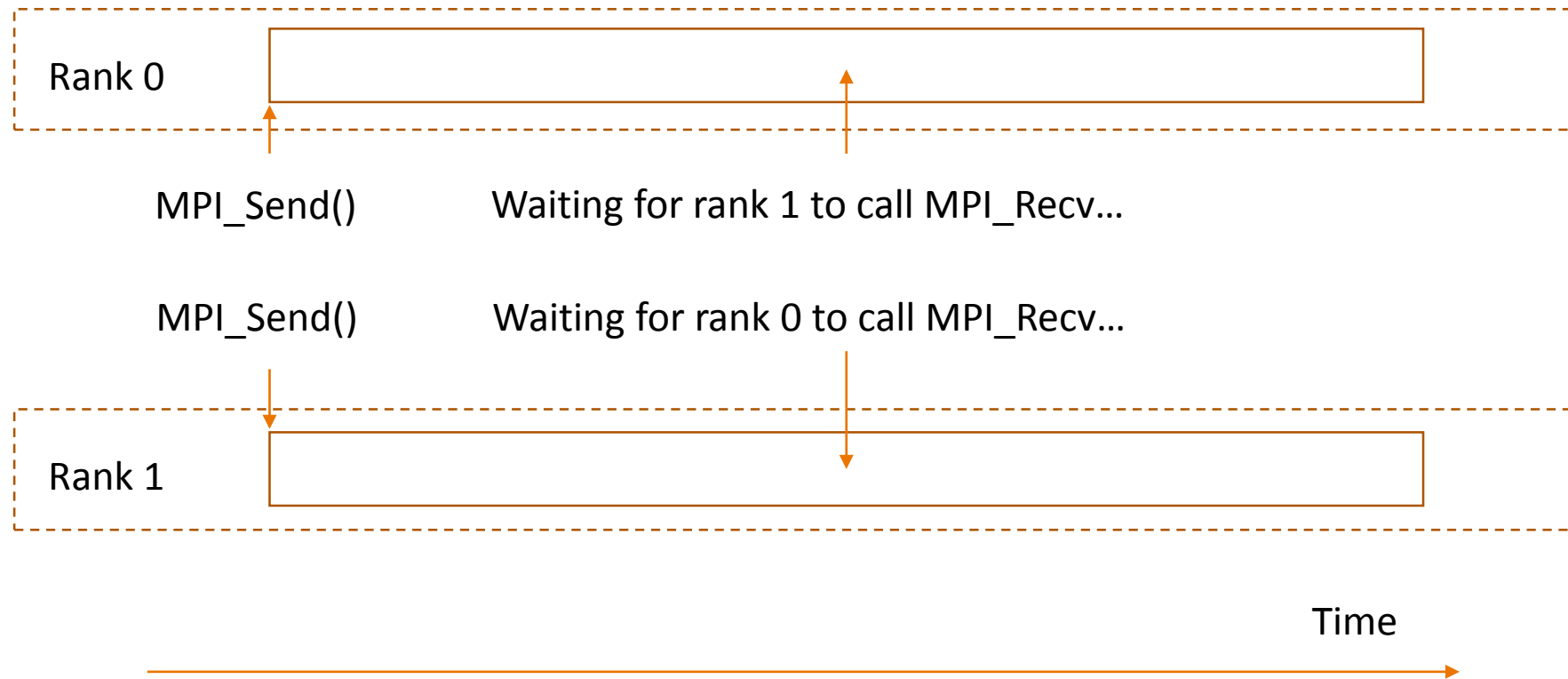


Hands-on

- Try the code of the ping-pong communication
 - C/C++: `mpi_pingpong.c`
 - Fortran: `mpi_pingpong.f`
- Have a look at the code and see how it is working
- What can be wrong?

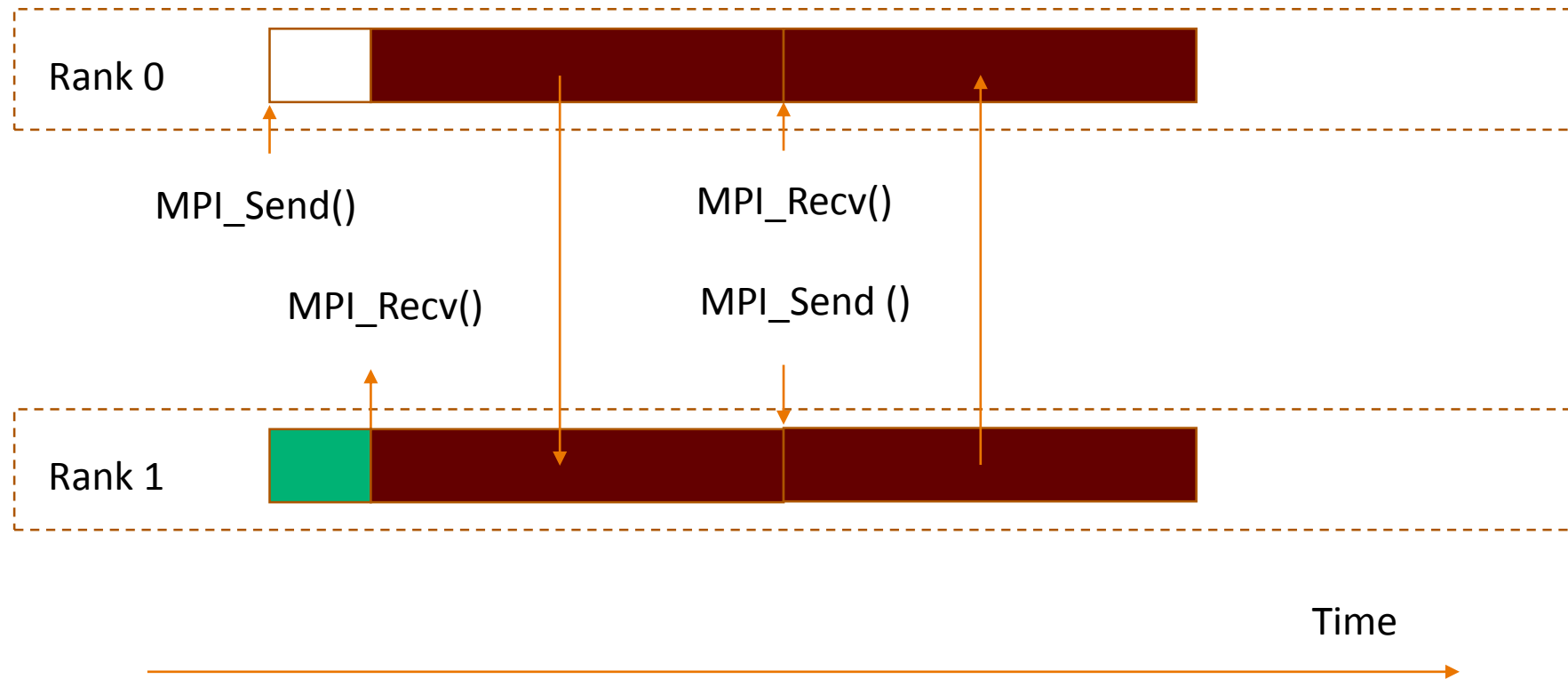
MPI point-to-point communication: messages

- MPI_Send and MPI_Recv are blocking functions: a deadlock may occur!!!



MPI point-to-point communication: messages

- Solution: reverse the order of MPI_Send and MPI_Recv for one of the communications



General problems with blocking communication

- If the MPI library implementation uses synchronous sends, a programmer may find
 - Deadlocks: complete block of the communication, in which two processes try to send at the same time
 - Serialization: inefficient communication because of a bad scheduling of communications

General problems with blocking communication

- If the MPI library implementation uses synchronous sends, a programmer may find
 - Deadlocks: complete block of the communication, in which two processes try to send at the same time
 - Serialization: inefficient communication because of a bad scheduling of communications
- Solution: non-blocking communications

Non-blocking communication

- It is possible to split the communication process (both send and recv) in two steps
- First step: post a communication request
 - Immediate return of the call and obtain an identifier for the MPI request
 - The effective communication will be done by the MPI library in the background
- Second step: wait for the communication to be completed
 - This will happen after the whole send buffer has been processed and the receive buffer contains all the necessary information

Non-blocking communication

- General routine to immediately send a message

- C/C++: `int MPI_Isend(void *buffer, int count, MPI_Datatype datatype,
int dest, int tag, MPI_Comm comm, MPI_Request request);`

- Fortran: `MPI_ISEND(buffer, count, datatype, dest, tag, comm, request, ierr)`
`<datatype> buffer(*)`
`integer :: count, datatype, dest, tag, comm, request, ierr`

Non-blocking communication

- General routine to immediately receive a message

- C/C++:

```
int MPI_Irecv(void *buffer, int count, MPI_Datatype datatype,  
             int src, int tag, MPI_Comm comm, MPI_Request request);
```

- Fortran:

```
MPI_IRecv(buffer, count, datatype, src, tag, comm, request, ierr)  
          <datatype> buffer(*)  
          integer :: count, datatype, src, tag, comm, request, ierr
```

Non-blocking communication

- General routine wait for a communication to complete
 - C/C++: `int MPI_Wait(MPI_Request request, MPI_Status status);`
 - Fortran: `MPI_WAIT(request, status, ierr)`
`integer :: request, status(MPI_STATUS_SIZE), ierr`

Non-blocking communication

- In a practical way, a call to the wait routine right after an immediate send or receive is analogous to a call to the blocking send or receive, respectively
- In any case, the request obtained from the isend or irecv should be matched in wait
- Between the two steps (immediate communication and wait) a process is allowed to perform additional operations that are unrelated to the communication
- This means the following:
 - The sending process is not allowed to access or reuse the sending buffer or any of the request information
 - The completeness of the operation is only ensured after calling the wait function
 - at least for C/C++

Non-blocking communication (Fortran only)

- There is a special issue with Fortran: it is an optimizing language, so a call to the wait routine may not imply that the send or recv buffers are ready for use again
- After MPI 3.0, an additional line of code is required In order to avoid any problem with buffer synchronization in Fortran
- Necessary steps:
 - Use the module `mpi_f08` and declare the `isend` or `irecv` buffer as asynchronous

 `<datatype>, asynchronous :: buffer`
 - Execute the following line of code always immediately after `MPI_WAIT`

```
IF (.NOT. MPI_ASYNC_PROTECTS_NONBLOCKING)  
&    CALL MPI_F_SYNC_REG( buf )
```

Reference: MPI: A Message-Passing Interface Standard Version 3.1, Chapter 17, page 644, section “Calling `MPI_F_SYNC_REG`”

Non-blocking communication

- It is also possible to perform a wait in a different way using the test routine
- The use of test requires a loop until the request is fulfilled
 - `flag == 1` in C or `flag == .TRUE.` in Fortran
 - Fortran: this is equivalent to wait, so the asynchronous buffer should also be protected using `MPI_F_SYNC_REG` as indicated
- Test function
 - C/C++: `int MPI_Test(MPI_Request request, int flag, MPI_Status status);`
 - Fortran: `MPI_TEST(request, flag, status, ierr)`
`integer :: request, status(MPI_STATUS_SIZE), ierr`
`logical :: flag`

Non-blocking communication

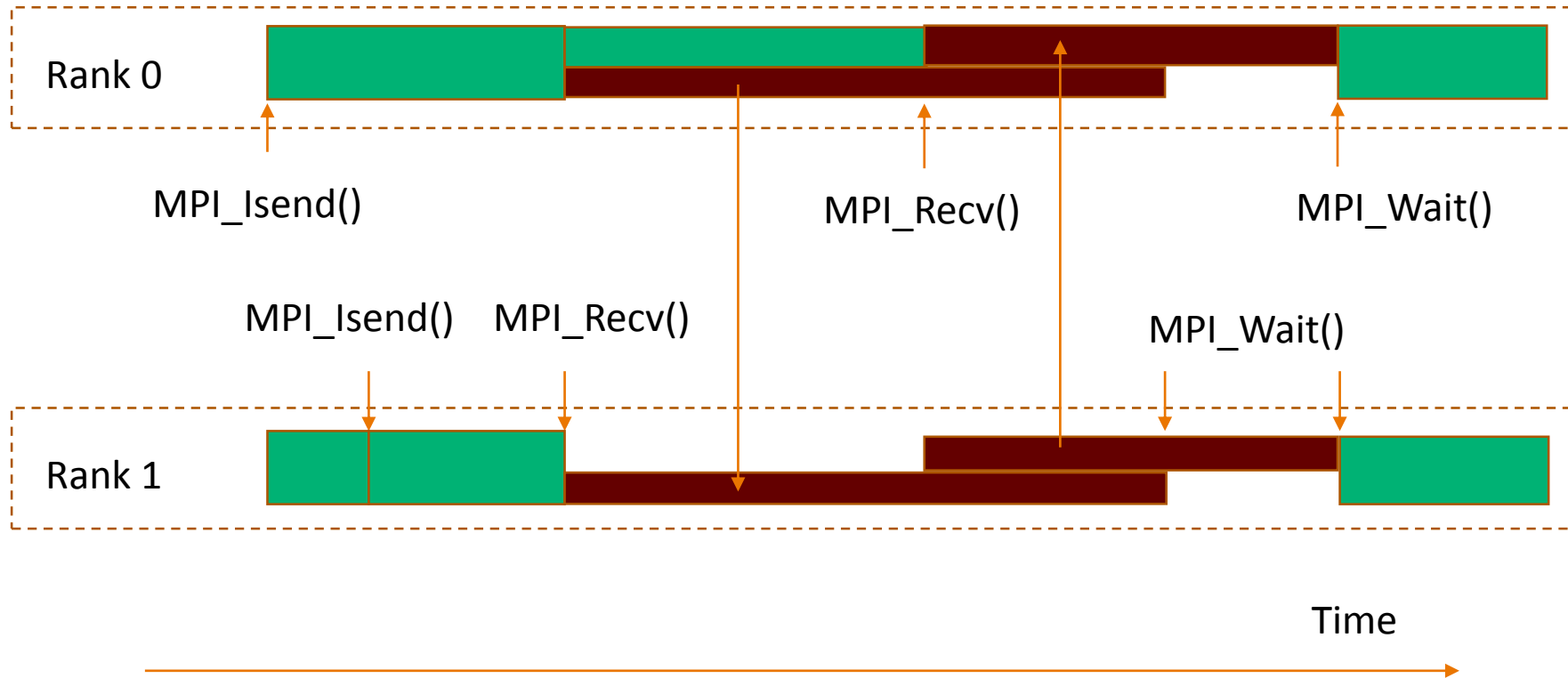
- Another features include It is also possible to perform a wait in a different way using the test routine
- The use of test requires a loop until the request is fulfilled
 - `flag == 1` in C or `flag == .TRUE.` in Fortran
 - Fortran: this is equivalent to wait, so the asynchronous buffer should also be protected using `MPI_F_SYNC_REG` as indicated
- Test function
 - C/C++: `int MPI_Test(MPI_Request request, int flag, MPI_Status status);`
 - Fortran: `MPI_TEST(request, flag, status, ierr)`

`integer :: request, status(MPI_STATUS_SIZE), ierr`

`logical :: flag`

Non-blocking communication example

- The processing of two MPI_Isend() and MPI_Recv() calls may be overlapped
- Deadlocks are gone, and it provides a way to avoid serialization!



Hands-on

- Work with the following code:
 - C/C++: `mpi_pingpong_nonblocking.c`
 - Fortran: `mpi_pingpong_nonblocking.f`
- Adapt the code to be really non-blocking
- Use `MPI_Isend` and/or `MPI_Irecv` conveniently
- Check the performance of the code compared to the blocking version
 - C/C++: `mpi_pingpong_blocking.c`
 - Fortran: `mpi_pingpong_blocking.f`
- Which is the fastest version, and how much faster is it?
- There is another function called `MPI_SENDRECV`... Can you check the standard specification and see if it would be possible to use it here?

Collective communication

- Sometimes it is necessary to use many point-to-point communications at a time, with specific coordination and synchronization between processes
- Collective functions provide optimized implementations for some common communication patterns
- Different types of collectives
 - Data movement
 - Computation
 - Synchronization

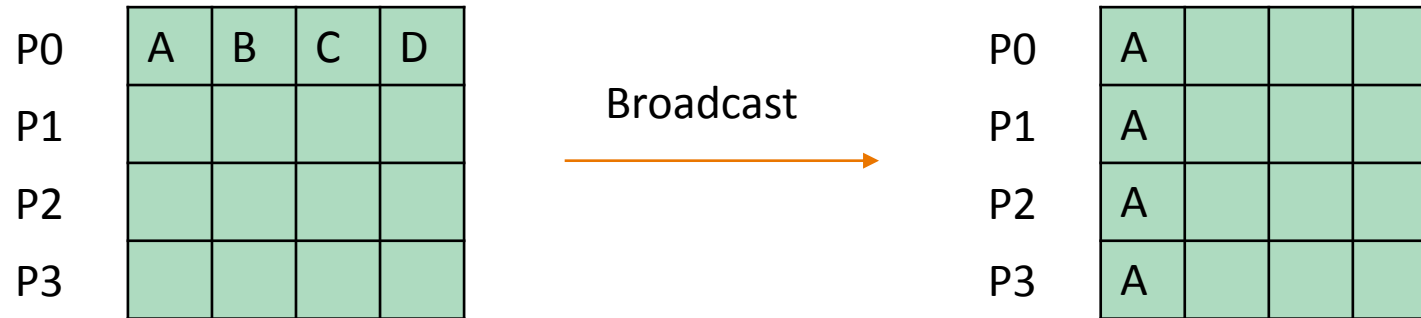
Collective communication

- One-to-all

| | | | | |
|----|---|---|---|---|
| P0 | A | B | C | D |
| P1 | | | | |
| P2 | | | | |
| P3 | | | | |

Collective communication

- One-to-all



Collective communication

- Broadcast

- C/C++: `int MPI_Bcast(void *buffer, int count, MPI_Datatype datatype, int root, MPI_Comm comm);`

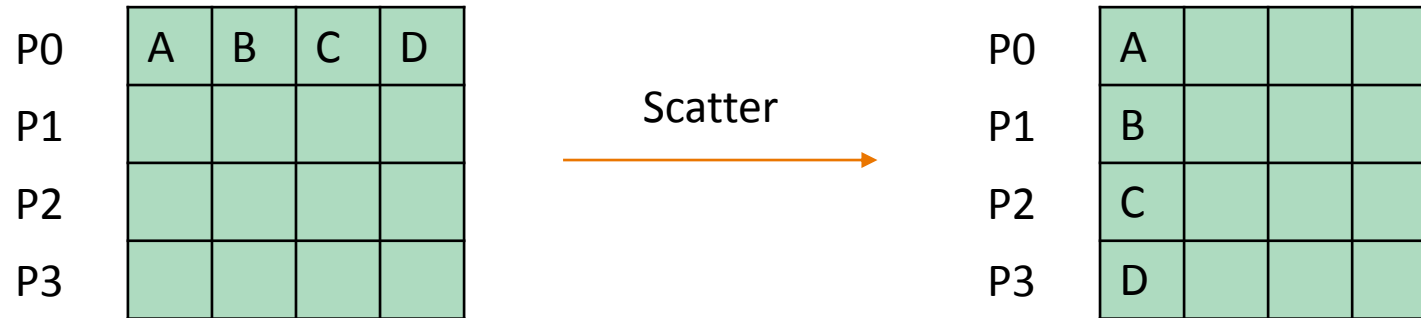
- Fortran: `MPI_BCAST(buffer, count, datatype, root, comm, ierr)`

`<datatype> buffer(*)`

`integer :: count, datatype, root, comm, ierr`

Collective communication

- One-to-all



Collective communication

- Scatter (with the option of defining MPI_IN_PLACE as sendbuffer)

- C/C++: `int MPI_Scatter(
 void *sendbuffer, int sendcount, MPI_Datatype senddatatype,
 void *recvbuffer, int recvcount, MPI_Datatype recvdatatype,
 int root, MPI_Comm comm);`

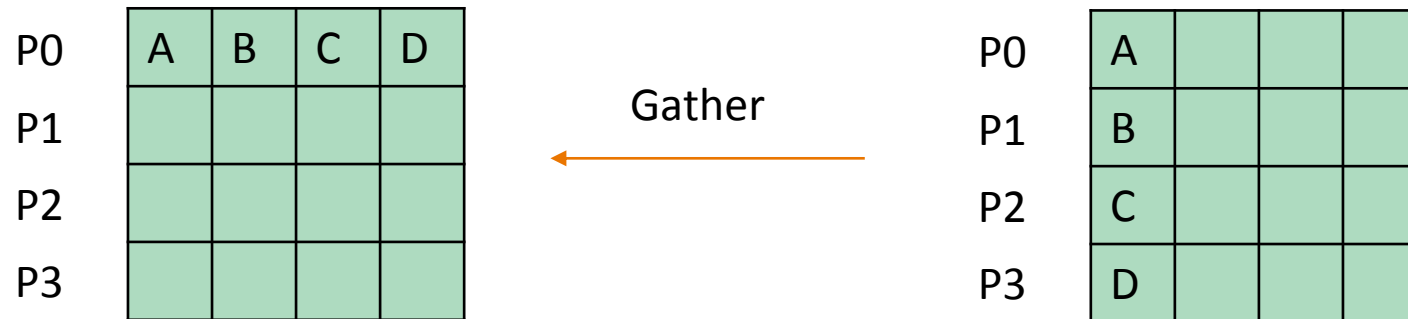
- Fortran: `MPI_SCATTER(sendbuffer, sendcount, senddatatype,
 recvbuffer, recvcount, recvdatatype,
 root, comm, ierr)`

`<datatype> sendbuffer(*), recvbuffer(*)`

`integer :: sendcount, senddatatype, recvcount, recvdatatype, root, comm, ierr`

Collective communication

- All-to-one



Collective communication

- Gather (with the option of defining MPI_IN_PLACE as sendbuffer)

- C/C++: `int MPI_Gather(
void *sendbuffer, int sendcount, MPI_Datatype senddatatype,
void *recvbuffer, int recvcount, MPI_Datatype recvdatatype,
int root, MPI_Comm comm);`

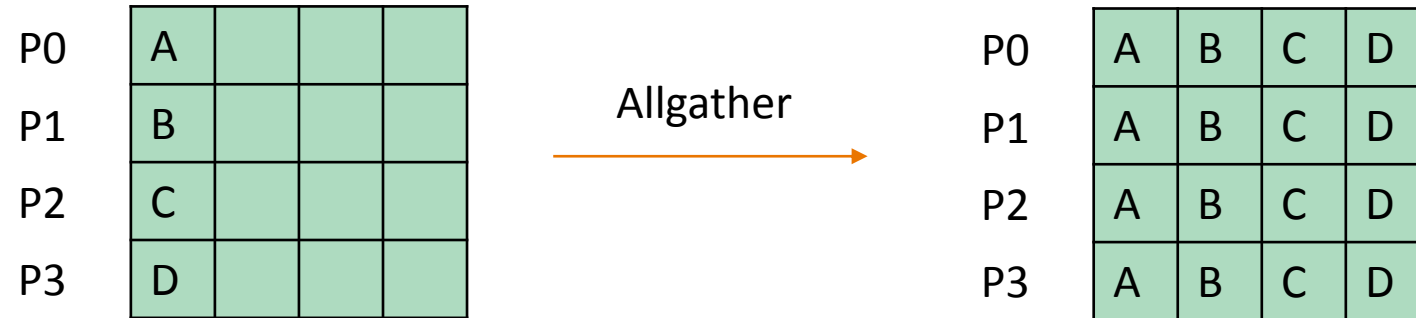
- Fortran: `MPI_GATHER(sendbuffer, sendcount, senddatatype,
recvbuffer, recvcount, recvdatatype,
root, comm, ierr)`

`<datatype> sendbuffer(*), recvbuffer(*)`

`integer :: sendcount, senddatatype, recvcount, recvdatatype, root, comm, ierr`

Collective communication

- All-to-all



Collective communication

- Allgather

- C/C++:

```
int MPI_Allgather(  
    void *sendbuffer, int sendcount, MPI_Datatype senddatatype,  
    void *recvbuffer, int recvcount, MPI_Datatype recvdatatype,  
    MPI_Comm comm);
```

- Fortran:

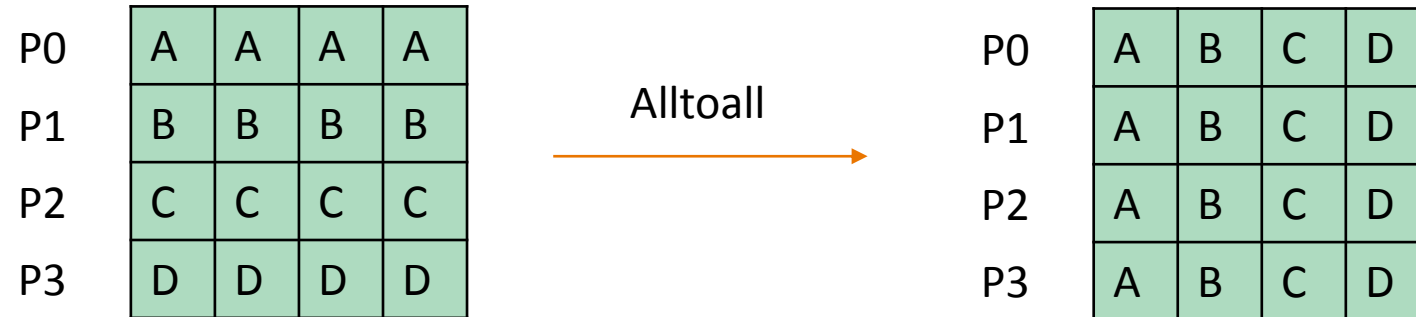
```
MPI_ALLGATHER(sendbuffer, sendcount, senddatatype,  
               recvbuffer, recvcount, recvdatatype,  
               comm, ierr)
```

`<datatype> sendbuffer(*), recvbuffer(*)`

`integer :: sendcount, senddatatype, recvcount, recvdatatype, comm, ierr`

Collective communication

- All-to-all



Collective communication

- Alltoall

- C/C++:

```
int MPI_Alltoall(  
    void *sendbuffer, int sendcount, MPI_Datatype senddatatype,  
    void *recvbuffer, int recvcount, MPI_Datatype recvdatatype,  
    MPI_Comm comm);
```

- Fortran:

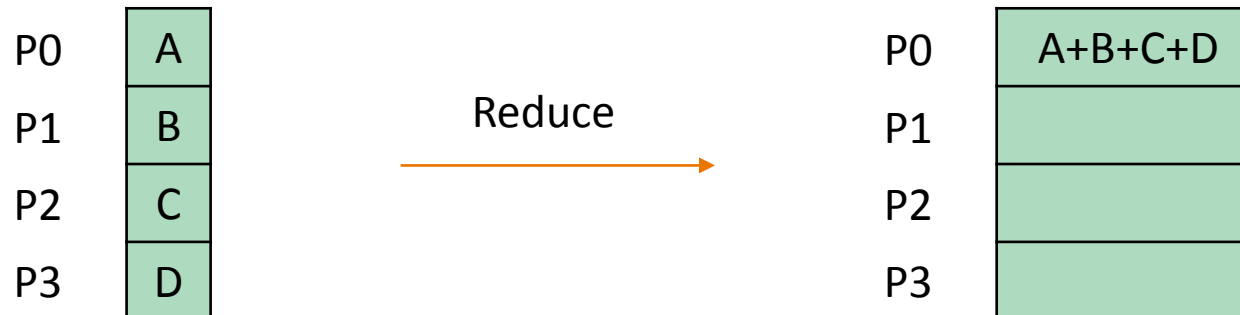
```
MPI_ALLTOALL(sendbuffer, sendcount, senddatatype,  
              recvbuffer, recvcount, recvdatatype,  
              comm, ierr)
```

<datatype> sendbuffer(*), recvbuffer(*)

integer :: sendcount, senddatatype, recvcount, recvdatatype, comm, ierr

Collective communication

- Computational operations are also possible
 - Most common one: reduce



Collective communication

- Reduce

- C/C++:

```
int MPI_Reduce(  
    void *sendbuffer, void *recvbuffer, int count,  
    MPI_Datatype datatype, MPI_Op op, int root,  
    MPI_Comm comm);
```
- Fortran:

```
MPI_ALLTOALL(sendbuffer, recvcount, count, datatype  
              op, root, comm, ierr)
```

<datatype> sendbuffer(*), recvbuffer(*)

integer :: count, datatype, op, comm, ierr

Collective communication: reduction operations

- There are different predefined operations for reduce
 - MPI_MIN, MPI_MAX, MPI_SUM, MPI_PROD
- Another option: custom definition of the operation
 - C/C++:

```
int MPI_Op_create(MPI_User_function *function,  
                  int commute, MPI_Op *op);
```
 - Fortran:

```
MPI_OP_CREATE(function, commute, op, ierr)
```

```
external :: function  
logical :: commute  
integer :: op, ierr
```


Collective communication

- Other collective operation: synchronization barrier
 - C/C++: `int MPI_Barrier(MPI_Comm comm);`
 - Fortran: `MPI_BARRIER(comm, ierr)`
`integer :: comm, ierr`

Hands-on (1)

- Try to solve the exercise on collective functions proposed in this directory

<https://github.com/sara-nl/PRACE-MPI-OpenMP/tree/master/MPI/collectives>

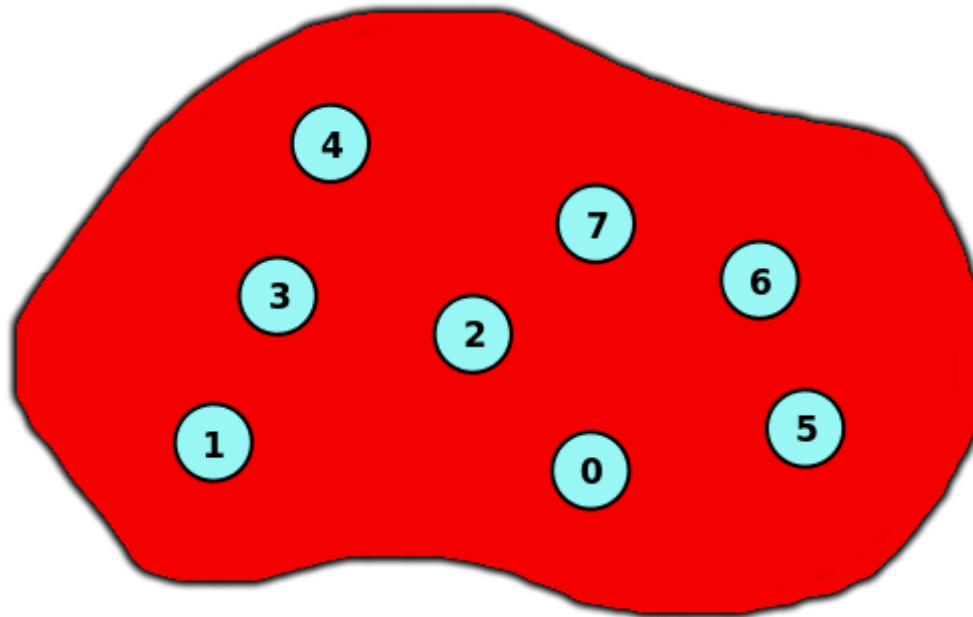
- The original codes are stored in the [PRACE CodeVault](#)

Hands-on (2)

- ... we've already come a long way here. Time for a parallelization from scratch!
- Try to parallelize the computation of π
 - C/C++: `mpi_pi.c`
 - Fortran: `mpi_pi.f`
- Three main tasks need to be performed
 - Distribution of the different iterations between processes
 - Getting the final complete result from all processes
 - Checking that there is scalability with a significant amount of threads

Communicators and virtual topologies

- Well known initial communicator: `MPI_COMM_WORLD`
- Main representation: pool of processes

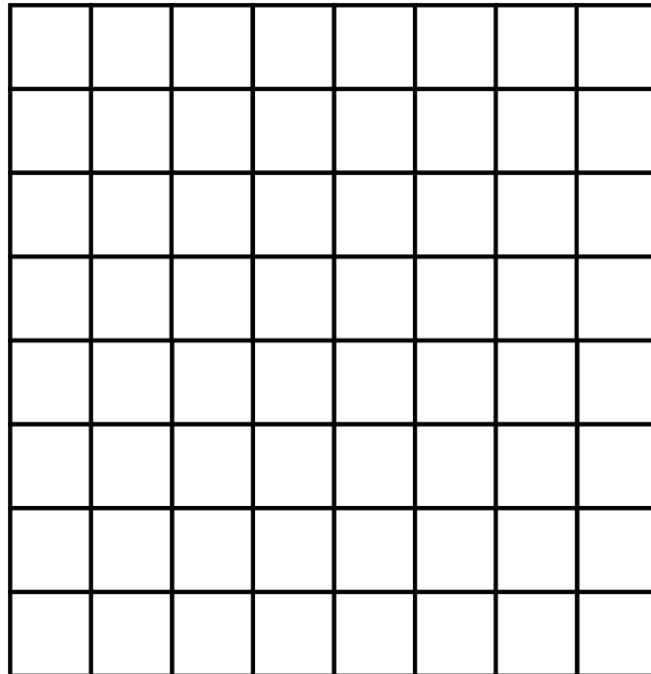


Communicators and virtual topologies

- Possible definition of specific connections between processes to obtain a more structured view of a communicator
- Reasons to obtain this?

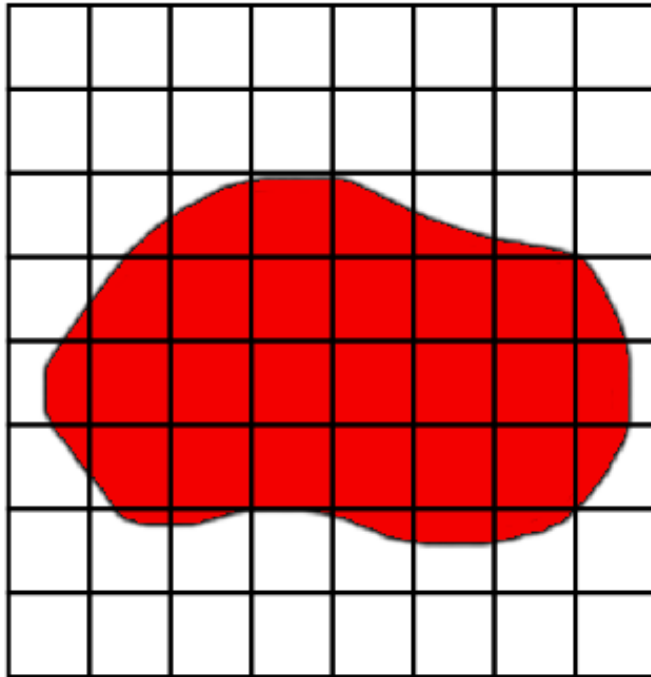
Communicators and virtual topologies

- Possible definition of specific connections between processes to obtain a more structured view of a communicator
- Reasons to obtain this?
- Mimic the structure of the data that we are working on!



Communicators and virtual topologies

- Possible definition of specific connections between processes to obtain a more structured view of a communicator
- Reasons to obtain this?
- Mimic the structure of the data that we are working on!



Communicators and virtual topologies

- MPI provides virtual topologies to organize processes
- Easy access to neighbor processes for communication
- Most common case: cartesian topology
 - C/C++:

```
int MPI_Cart_create(MPI_Comm comm_old, int ndims,  
                    int dims[], int periods[], int reorder,  
                    MPI_Comm *comm_cart);
```
 - Fortran:

```
MPI_CART_CREATE(comm_old, ndims, periods, reorder,  
                comm_cart, ierr)  
  
integer :: comm_old, ndims, dims(*), comm_cart, ierr  
logical :: periods(*), reorder
```


Communicators and virtual topologies

- MPI provides virtual topologies to organize processes
- Easy access to neighbor processes for communication
- Most common case: cartesian topology
 - C/C++:

```
int MPI_Cart_create(MPI_Comm comm_old, int ndims,  
                    int dims[], int periods[], int reorder,  
                    MPI_Comm *comm_cart);
```
 - Fortran:

```
MPI_CART_CREATE(comm_old, ndims, periods, reorder,  
                comm_cart, ierr)  
  
integer :: comm_old, ndims, dims(*), comm_cart, ierr  
logical :: periods(*), reorder
```

Communicators and virtual topologies

- Connections between processes in cartesian topologies are established by their given coordinates: easy to calculate and to convert to/from rank
 - C/C++: `int MPI_Cart_coords(MPI_Comm comm, int rank, int maxdims, int coords[]);`
 - Fortran: `MPI_CART_COORDS(comm, rank, maxdims, coords, ierr)`
`integer :: comm, rank, maxdims, coords, ierr`

Communicators and virtual topologies

- Connections between processes in cartesian topologies are established by their given coordinates: easy to calculate and to convert to/from rank
 - C/C++: `int MPI_Cart_rank(MPI_Comm comm, int coords[], int *rank);`
 - Fortran: `MPI_CART_RANK(comm, coords, rank, ierr)`
`integer :: comm, rank, maxdims, coords, ierr`

Communicators and virtual topologies

- Cartesian topologies may also be subdivided in their different dimensions

- C/C++:

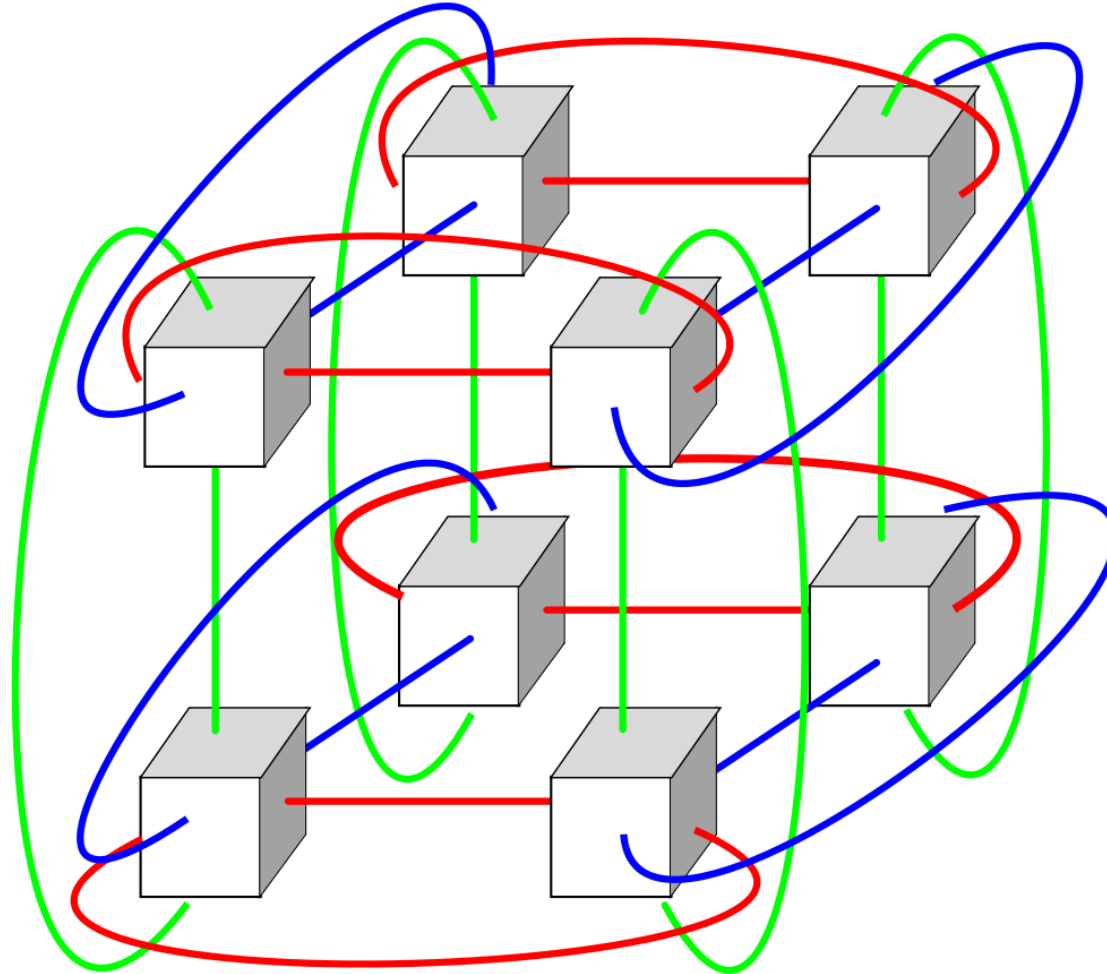
```
int MPI_Cart_sub(MPI_Comm comm, int remain_dims[],  
                MPI_Comm *comm_new);
```

- Fortran:

```
MPI_CART_SUB(comm, remain_dims, comm_new, ierr)  
integer :: comm, comm_new, ierr  
logical :: remain_dims(*)
```

Hands-on

- Just do this:
 - C/C++: `mpi_torus.c`
 - Fortran: `mpi_torus.f`



Communicators and virtual topologies

- Communicators also have maximum flexibility to be created with a custom amount of threads, and later reorganized
- It is necessary to operate at the level of groups of processes to define a completely custom communicator
 - C/C++: `int MPI_Comm_split(MPI_Comm comm, int color, int key, MPI_Comm *newcomm);`
 - Fortran: `MPI_COMM_SPLIT(comm, color, key, newcomm, ierr)`
`integer :: comm, color, key, newcomm, ierr`

Communicators and virtual topologies

- Communicators also have maximum flexibility to be created with a custom amount of threads, and later reorganized
- It is necessary to operate at the level of groups of processes to define a completely custom communicator
 - C/C++: `int MPI_Comm_group(MPI_Comm comm, MPI_Group *group);`
 - Fortran: `MPI_COMM_GROUP(comm, group, ierr)`
`integer :: comm, group, ierr`

Communicators and virtual topologies

- Communicators also have maximum flexibility to be created with a custom amount of threads, and later reorganized
- It is necessary to operate at the level of groups of processes to define a completely custom communicator
 - C/C++:

```
int MPI_Group_incl(MPI_Group group, int nranks, int ranks[],  
                  MPI_Group *newgroup);
```
 - Fortran:

```
MPI_GROUP_INCL(group,nranks,ranks,newgroup, ierr)  
integer :: group, nranks, ranks(*), newgroup, ierr
```


Communicators and virtual topologies

- Communicators also have maximum flexibility to be created with a custom amount of threads, and later reorganized
- It is necessary to operate at the level of groups of processes to define a completely custom communicator
 - C/C++: `int MPI_Comm_create(MPI_Comm comm, MPI_Group group, MPI_Comm *newcomm);`
 - Fortran: `MPI_COMM_CREATE(comm, group, newcomm, ierr)`
`integer :: comm, group, newcomm, ierr`

Hands-on

- Take the previous example of the 3D torus
- Create communicators for each dimension of it
- Check the values of the ranks for every communicator

General comments (1)

- MPI is based on processes, but also provides threading possibilities
 - `MPI_InitThread(int *argc, char ** argv[], int thr_requested, int thr_provided);`
 - Varying level of support by the different MPI implementations
- MPI provides a quite complete I/O library with plenty of functionality
 - Recommended: use of high-level libraries that use it with wrappers
 - E.g. PnetCDF, SIONlib
- Derived data types can be defined and may be useful, but relatively cumbersome
 - Try to check if it is possible to codify relatively complex communication using basic data types
 - Use the general type `MPI_PACKED` and different routines (`MPI_Pack` and `MPI_Unpack` to put different data types in the same stream

General comments (& 2)

- MPI also supports one-sided and shared memory communication
 - MPI_Get / MPI_Put / MPI_Win_create ...
 - Efficient way of performing data movements without explicit “handshake”
 - More info in a forthcoming PRACE MOOC on FutureLearn
- MPI has lots of further possibilities
 - Keep checking the [standard documentation](#)
 - Keep practising and be challenged!

PARALLEL PROGRAMMING WITH OPENMP

Caspar van Leewen
Carlos Teijeiro Barjas