Introduction to Message Passing Interfaces

1st Lesson

Dr. Matteo Barborini

HPC Platform – University of Luxembourg

Maison du Nombre
6, avenue de la Fonte L-4364 Esch-sur-Alzette Luxembourg

matteo.barborini@uni.lu

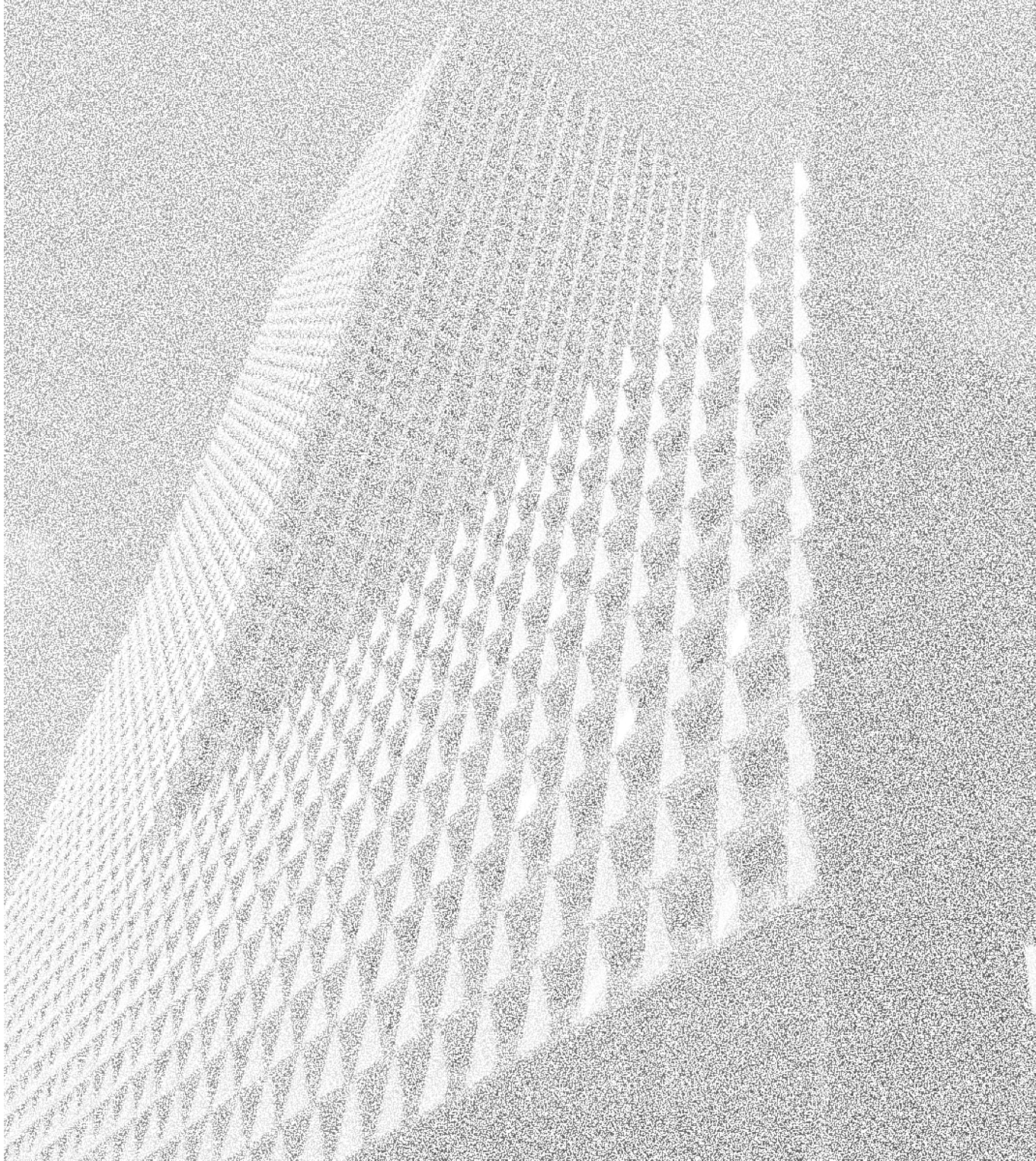


Table of Contents

1. Recalling basic hardware characteristics
2. Message Passing Interface
3. Brief comparision to OpenMP
4. Download, compilation and execution
5. First examples in C

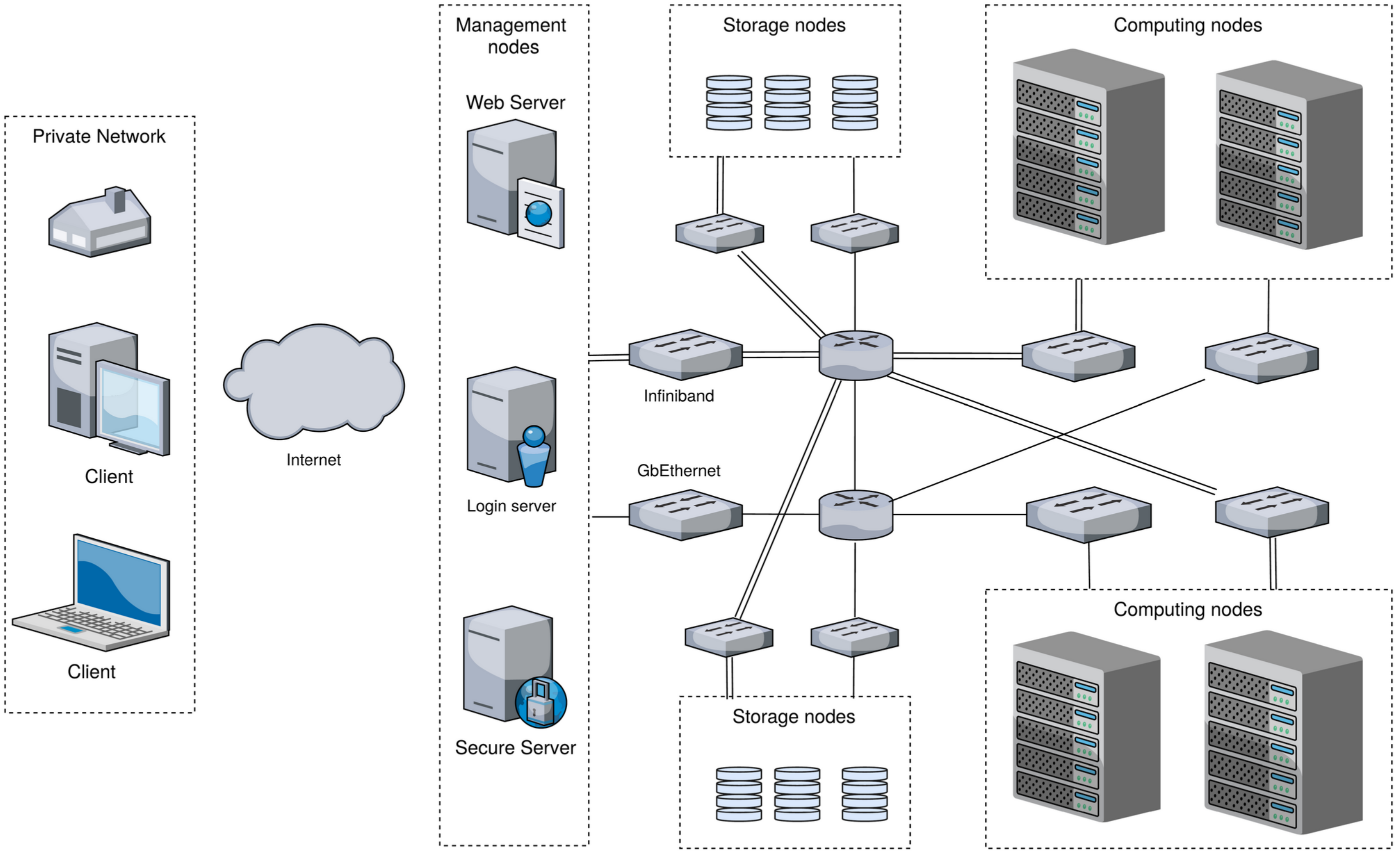
uni.lu

UNIVERSITY OF
LUXEMBOURG

HPC facility & Parallel computing

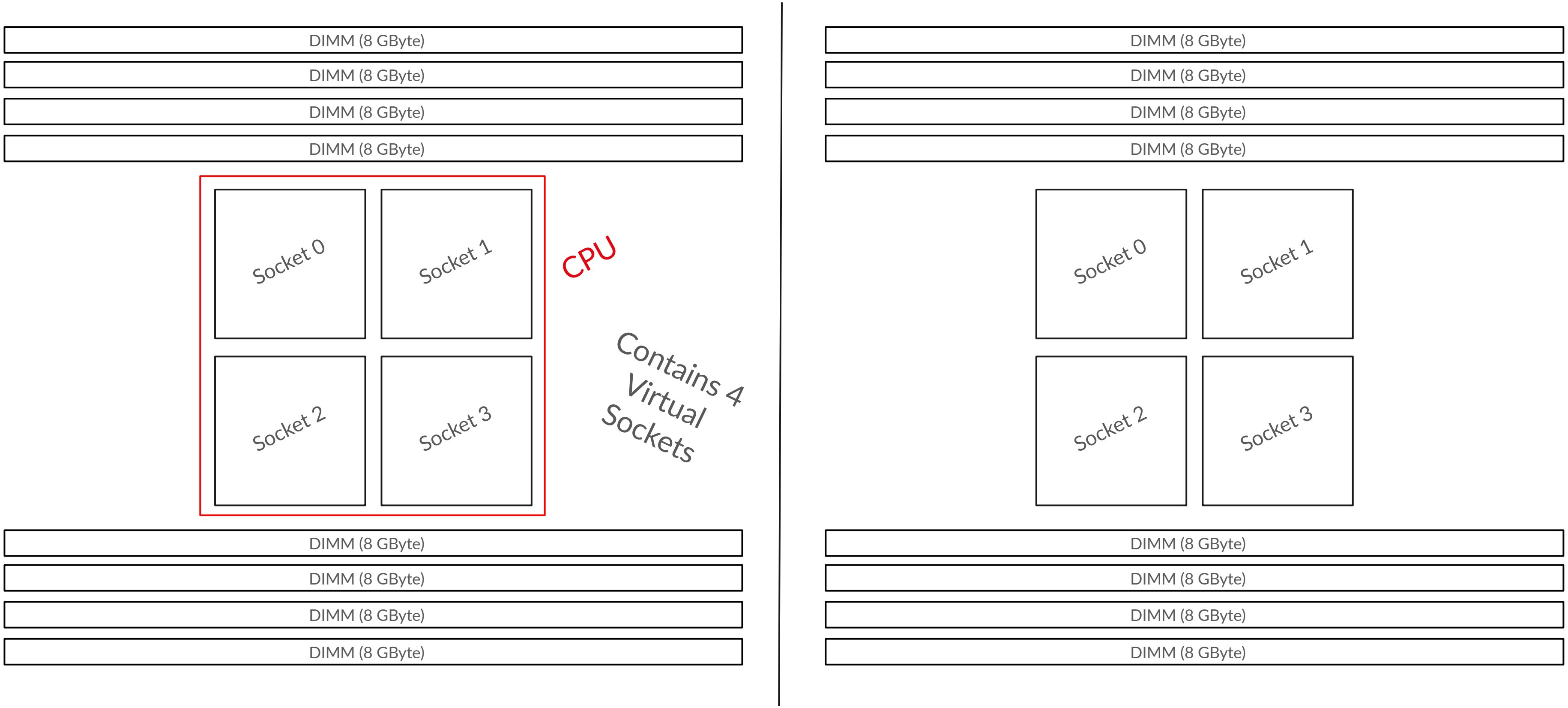
Introduction to MPI programming in C – 2025/2026
Lesson 1

High-Performance Computing facility



F. Reghenzani, G. Massari and W. Fornaciari, "Timing Predictability in High-Performance Computing With Probabilistic Real-Time," IEEE Access, 8, 208566-208582, 2020

Non-uniform memory access (NUMA) Nodes



DIMM (8 GByte)
DIMM (8 GByte)
DIMM (8 GByte)
DIMM (8 GByte)

Socket 0
Socket 1
Socket 2
Socket 3

CPU

Contains 4 Virtual Sockets

DIMM (8 GByte)
DIMM (8 GByte)
DIMM (8 GByte)
DIMM (8 GByte)

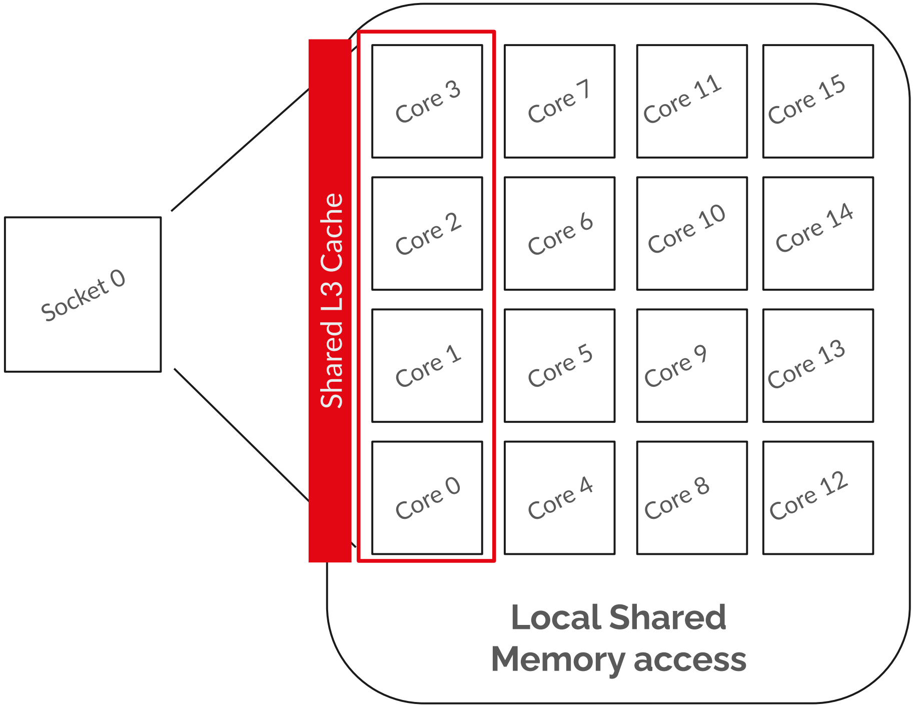
DIMM (8 GByte)
DIMM (8 GByte)
DIMM (8 GByte)
DIMM (8 GByte)

Socket 0
Socket 1
Socket 2
Socket 3

DIMM (8 GByte)
DIMM (8 GByte)
DIMM (8 GByte)
DIMM (8 GByte)

Z. Majo, T. R. Gross, "Memory system performance in a NUMA multicore multiprocessor" SYSTOR '11: Proceedings of the 4th Annual International Conference on Systems and Storage, 12, 1-10, 2011

Non-uniform memory access (NUMA) Nodes



Socket 0

Shared L3 Cache

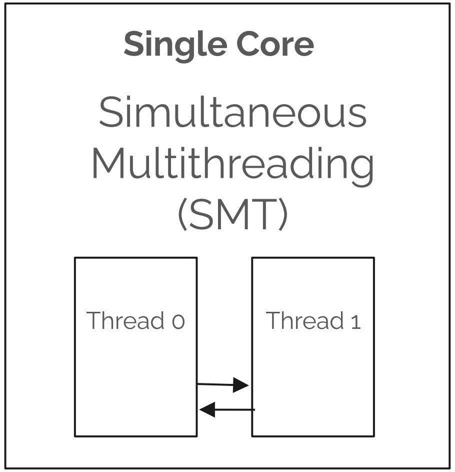
Core 3 Core 7 Core 11 Core 15

Core 2 Core 6 Core 10 Core 14

Core 1 Core 5 Core 9 Core 13

Core 0 Core 4 Core 8 Core 12

Local Shared Memory access



Single Core

Simultaneous Multithreading (SMT)

Thread 0

Thread 1

Z. Majo, T. R. Gross, "Memory system performance in a NUMA multicore multiprocessor" SYSTOR '11: Proceedings of the 4th Annual International Conference on Systems and Storage, 12, 1-10, 2011

What you have seen so far

Open Multi-Processing (OpenMP)

OpenMP (Open Multi-Processing) is an **API (Application Programming Interface)** that supports multi-platform shared-memory parallel programming in C, C++, and Fortran and also GPU programming (both Nvidia and AMD).

Programming Model: Shared memory

Execution: Threads run in parallel on shared memory space

Ease of Use: Easier to implement (uses compiler directives in C/C++/Fortran)

Communication: Implicit (threads share variables by default)

Scalability: Limited by number of cores on a single machine.

Best For: Multi-core shared-memory systems

www.openmp.org, T. Deakin and T. G. Mattson, "Programming Your GPU with OpenMP: Performance Portability for GPUs"

uni.lu
UNIVERSITY OF
LUXEMBOURG

Message Passing Interface

Introduction to MPI programming in C – 2025/2026
Lesson 1

Message Passing Interface (MPI)

MPI (Message Passing Interface) is a **standardized and portable** message-passing system designed to allow **processes to communicate with each other** in a parallel computing environment — especially across **distributed memory systems** like clusters or supercomputers.

Programming Model: Distributed memory

Execution: Independent processes with separate memory spaces

Ease of Use: More complex (requires explicit message passing)

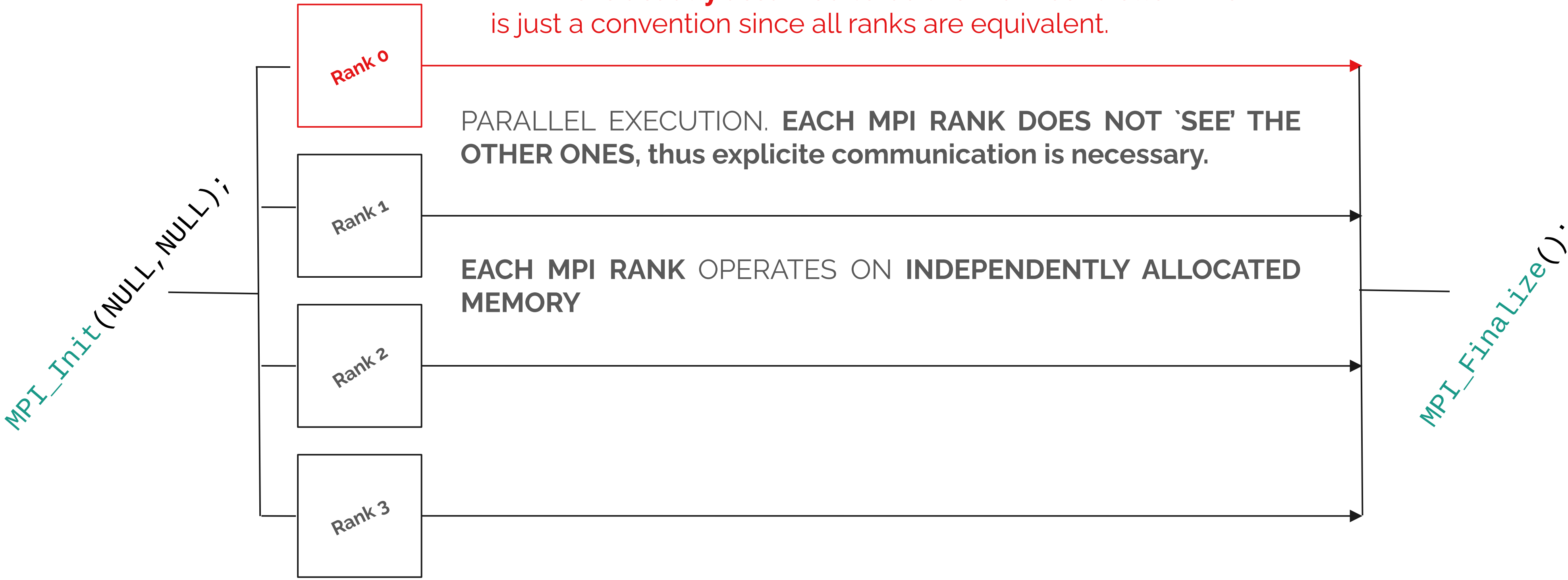
Communication: Explicit (send and receive messages between processes)

Scalability: Can scale very well across multiple nodes/machines if the computations can be uniformally distributed.

Best For: Large-scale distributed computing

www.mpi-forum.org/docs/, www.open-mpi.org

Message Passing Interface (MPI)



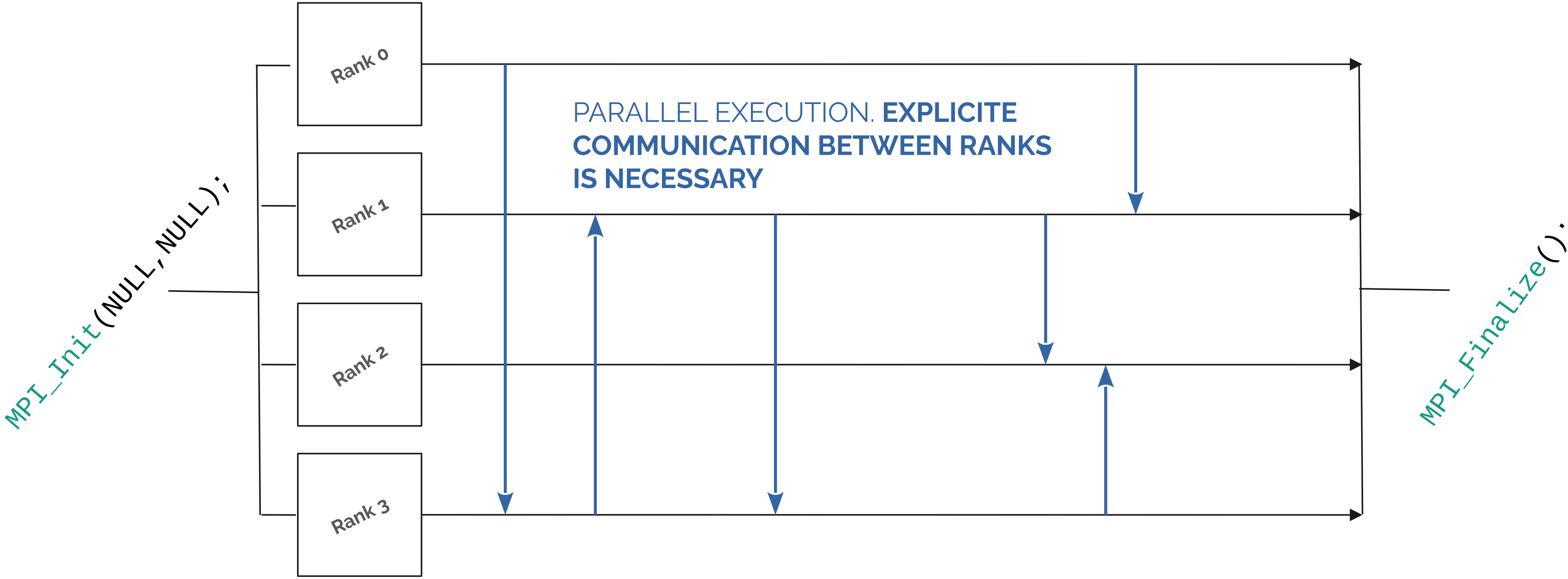
RANK 0 is usually assumed to be the main controller. This is just a convention since all ranks are equivalent.

PARALLEL EXECUTION. **EACH MPI RANK DOES NOT `SEE' THE OTHER ONES, thus explicite communication is necessary.**

EACH MPI RANK OPERATES ON **INDEPENDENTLY ALLOCATED MEMORY**

www.mpi-forum.org/docs/, www.open-mpi.org

Message Passing Interface (MPI)



PARALLEL EXECUTION. **EXPLICITE COMMUNICATION BETWEEN RANKS IS NECESSARY**

www.mpi-forum.org/docs/, www.open-mpi.org

UNIVERSITY OF
LUXEMBOURG

1.1.MPI_hello_world.c

Introduction to MPI programming in C – 2025/2026
Lesson 1

Example: 1.1.MPI_hello_world.c

To include the MPI library it is sufficient in both C and C++ to include the header of the library through the preprocessor command

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

The **MPI environment** is then **initialized** through the function:

```
int MPI_Init(int *argc, char ***argv)
```

that takes two sets of pointers as optional (unused) arguments (that can be both set to NULL)

```
MPI_Init(&argc, &argv);
```

or

```
MPI_Init(NULL, NULL);
```

The **MPI environment is** then **finalized** through the function:

```
int MPI_Finalize( void )
```

www.mpi-forum.org/docs/, www.open-mpi.org

Example: 1.1.MPI_hello_world.c

Within this example, after the initialization of the environment we find two additional commands, that are used to interrogate the environment about its properties:

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

This function returns the size of the communicator, that in the example is defined by the preconstructed MPI variable **MPI_COMM_WORLD.** Thus now ***size*** is the total number of MPI tasks.

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

This function returns the **number of the rank** associated to the communicator (preconstructed MPI variable **MPI_COMM_WORLD**). Thus now ***rank*** is the number of the particular MPI task that calls the function. Each rank will have a different number from 0 to max ranks specified when calling the MPI application with **`mpiexec`**.

www.mpi-forum.org/docs/, www.open-mpi.org

Example: 1.1.MPI_hello_world.c

A function that is usually, not used is

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

that returns the name of the processor as a string, and the length of the string.

In the example the name string variable's length is initialized through the variable

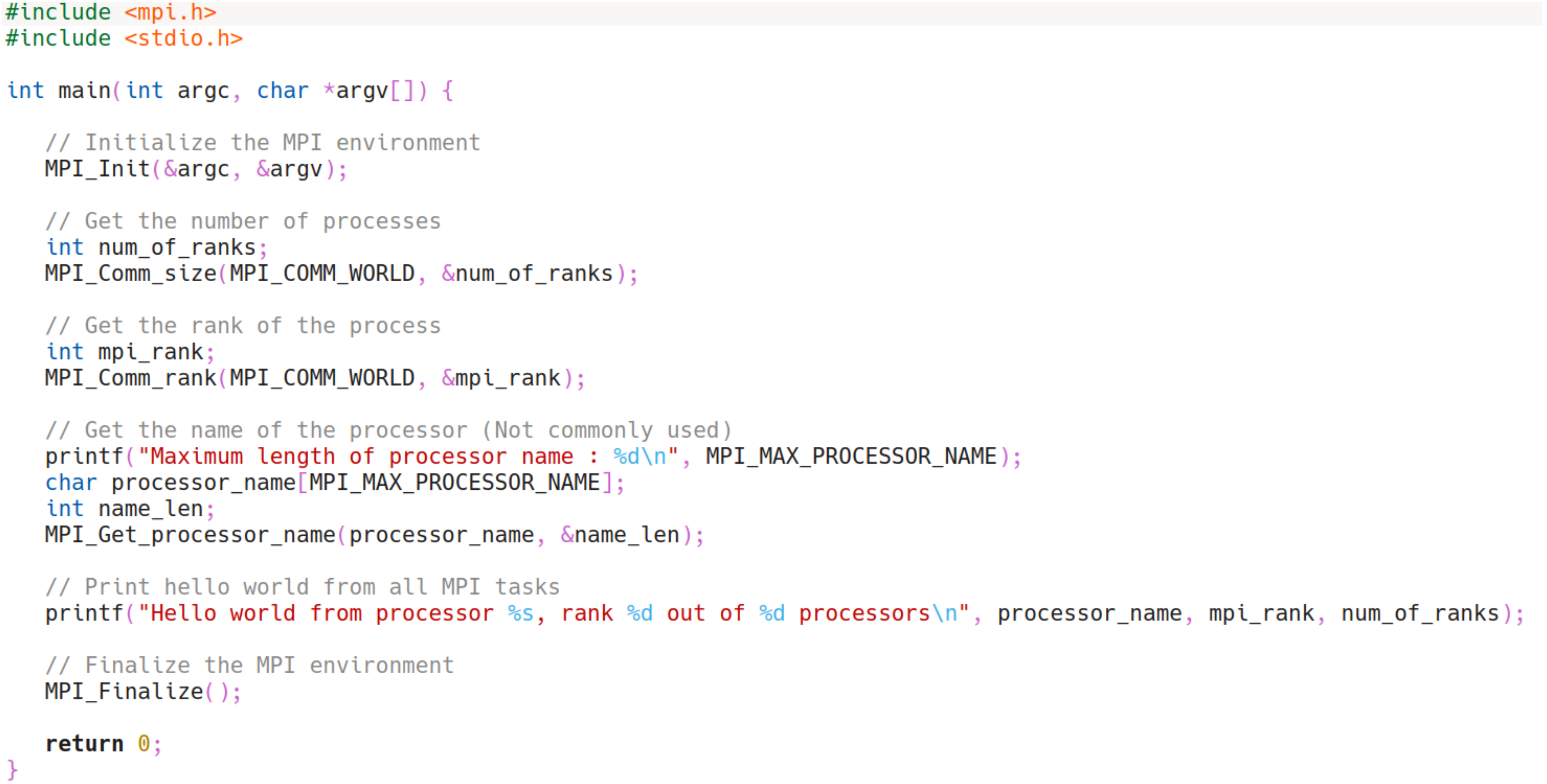
```
MPI_MAX_PROCESSOR_NAME
```

as

```
char processor_name[MPI_MAX_PROCESSOR_NAME];
```

www.mpi-forum.org/docs/, www.open-mpi.org

Example: 1.1.MPI_hello_world.c



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

int main(int argc, char *argv[]) {

    // Initialize the MPI environment
    MPI_Init(&argc, &argv);

    // Get the number of processes
    int num_of_ranks;
    MPI_Comm_size(MPI_COMM_WORLD, &num_of_ranks);

    // Get the rank of the process
    int mpi_rank;
    MPI_Comm_rank(MPI_COMM_WORLD, &mpi_rank);

    // Get the name of the processor (Not commonly used)
    printf("Maximum length of processor name : %d\n", MPI_MAX_PROCESSOR_NAME);
    char processor_name[MPI_MAX_PROCESSOR_NAME];
    int name_len;
    MPI_Get_processor_name(processor_name, &name_len);

    // Print hello world from all MPI tasks
    printf("Hello world from processor %s, rank %d out of %d processors\n", processor_name, mpi_rank, num_of_ranks);

    // Finalize the MPI environment
    MPI_Finalize();

    return 0;
}
```

www.mpi-forum.org/docs/, www.open-mpi.org

Compiling, linking and executing with MPI

Introduction to MPI programming in C – 2025/2026
Lesson 1

Compilation of MPI libraries

On a laptop machine in order to use MPI it is essential to install for example Intel MPI or OpenMPI libraries

Latest version of the **OpenMPI package** can be found in **www.open-mpi.org**

After the installation it is essential to export the bin and lib directories

```
export PATH=$HOME/Software/OpenMPI-5.0.6/bin:$PATH
export LD_LIBRARY_PATH=$HOME/Software/OpenMPI-5.0.6/lib:$LD_LIBRARY_PATH
```

On Meluxina or the ULHPC clusters the MPI libraries are usually loaded together with the toolchains

Meluxina:	`module load foss`	`module load intel`
ULHPC:	`module load toolchain/foss`	`module load toolchain/intel`

www.mpi-forum.org/docs/, www.open-mpi.org

Compilation of applications with MPI wrappers

By loading the MPI libraries, or exporting the system variables, different wrappers will be made available

```
mpicc, mpif90, mpic++, mpiifort, mpicxx …
```

To compile a simple C application such as the one defined in the previous slides we run:

```
mpicc -o myapplication.exe myapplication.c
```

or for Fortran 90 and higher:

```
mpif90 -o myapplication.exe myapplication.f90
```

To check the characteristics of each wrapper (compiler, options, ecc.) it is sufficient to run the command

```
mpicc --show, mpif90 --show, mpic++ --show, mpicxx --show
```

www.mpi-forum.org/docs/, www.open-mpi.org

Information about the MPI wrappers

The option `version` of the wrappers

```
mpicc --version
```

details the compiler associated to the wrapper

```
gcc (Ubuntu 15.2.0-4ubuntu4) 15.2.0
```

To check the details of the wrapper

```
mpicc ––show
```

That prints the full string of compiler and links to the MPI library

```
gcc -I/home/matteo/Software/OpenMPI-5.0.6/include
-L/home/matteo/Software/OpenMPI-5.0.6/lib -Wl,-rpath
-Wl,/home/matteo/Software/OpenMPI-5.0.6/lib -Wl,--enable-new-dtags -lmpi
```

www.mpi-forum.org/docs/, www.open-mpi.org

Executing the application on an HPC

To execute an application on an HPC, for example AION. A typical script can be written as

```
#!/bin/bash -l

#SBATCH -N 1
#SBATCH --ntasks-per-node 128 # MPI processes per node
#SBATCH --cpus-per-task 1     # Cpus available for each MPI process (OpenMP)
#SBATCH --time=0-01:00:00
#SBATCH -p batch

module load toolchain/foss

export OMP_NUM_THREADS=$SLURM_CPUS_PER_TASK

srun -n $SLURM_NTASKS myapplication.exe > output
```

Here we request **128 MPI tasks distributed in 1 AION node**, each task as available 1 core

www.mpi-forum.org/docs/, www.open-mpi.org, slurm.schedmd.com/sbatch, hpc-docs.uni.lu/slurm/launchers

Executing on laptop

First you export the directories of the MPI libraries

```
export PATH=$HOME/Software/OpenMPI-5.0.6/bin:$PATH
export LD_LIBRARY_PATH=$HOME/Software/OpenMPI-5.0.6/lib:$LD_LIBRARY_PATH
```

Assuming `${num_of_ranks} = 4` we have as are result of running the application:

```
mpiexec -n ${num_of_ranks} application.exe
```

The result of the execution gives:

```
Maximum length of processor name : 256
Maximum length of processor name : 256
Maximum length of processor name : 256
Maximum length of processor name : 256
Hello world from processor argo, rank 2 out of 4 processors
Hello world from processor argo, rank 3 out of 4 processors
Hello world from processor argo, rank 0 out of 4 processors
Hello world from processor argo, rank 1 out of 4 processors
```

(Asynchronous writing)

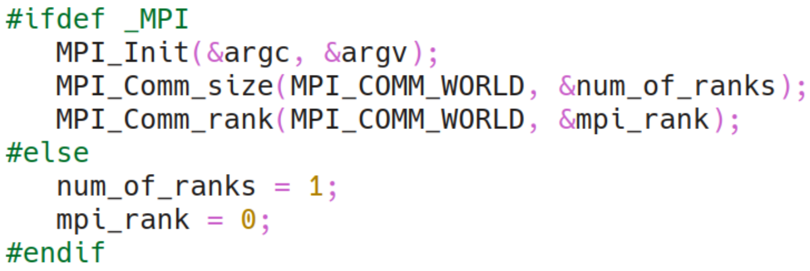
uni.lu
UNIVERSITY OF
LUXEMBOURG

Wrapping in preprocessors

Introduction to MPI programming in C – 2025/2026
Lesson 1

Example: 1.2.MPI_hello_world_PP.c

Through the use of the preprocessor in C we can introduce a variable that is used automatically to switch of the MPI calls, in order to make the serial version of a code compilable without MPI libraries (if not present).



```
#ifdef _MPI
   MPI_Init(&argc, &argv);
   MPI_Comm_size(MPI_COMM_WORLD, &num_of_ranks);
   MPI_Comm_rank(MPI_COMM_WORLD, &mpi_rank);
#else
   num_of_ranks = 1;
   mpi_rank = 0;
#endif
```

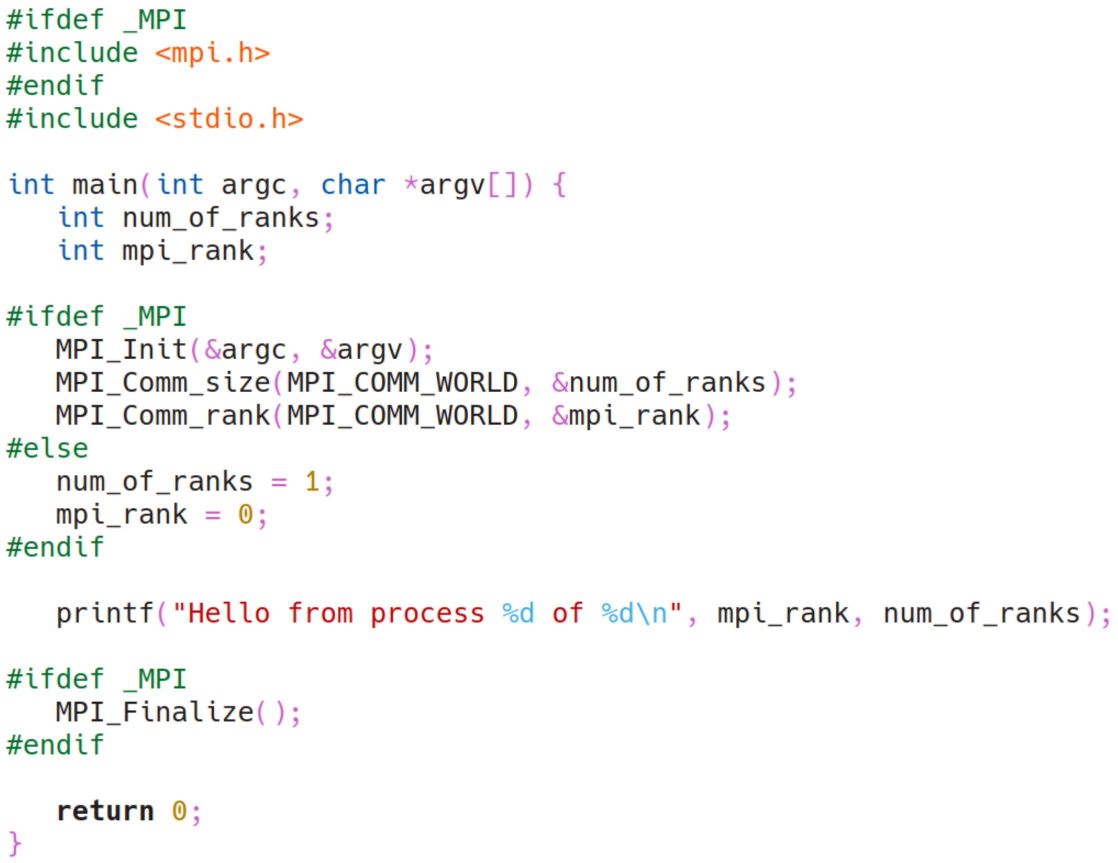
In this case _MPI is the variable that is checked. In order to compile the code with mpi we thus need to write:.

```
mpicc -o myapplication.exe myapplication.c -D_MPI
```

While to compile the serial version that does not need MPI we can simply do.

```
gcc -o myapplication.exe myapplication.c
```

Example: 1.2.MPI_hello_world_PP.c



```c
#ifdef _MPI
#include <mpi.h>
#endif
#include <stdio.h>

int main(int argc, char *argv[]) {
   int num_of_ranks;
   int mpi_rank;

#ifdef _MPI
   MPI_Init(&argc, &argv);
   MPI_Comm_size(MPI_COMM_WORLD, &num_of_ranks);
   MPI_Comm_rank(MPI_COMM_WORLD, &mpi_rank);
#else
   num_of_ranks = 1;
   mpi_rank = 0;
#endif

   printf("Hello from process %d of %d\n", mpi_rank, num_of_ranks);

#ifdef _MPI
   MPI_Finalize();
#endif

   return 0;
}
```

www.mpi-forum.org/docs/, www.open-mpi.org

MPI_Bcast

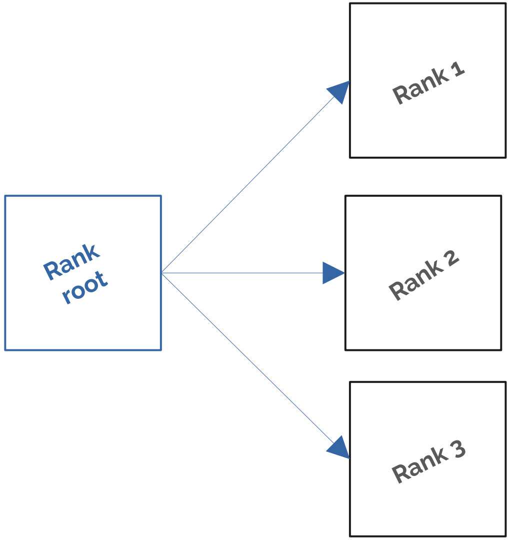
Introduction to MPI programming in C – 2025/2026
Lesson 1

Example: 2.1.MPI_bcast.c

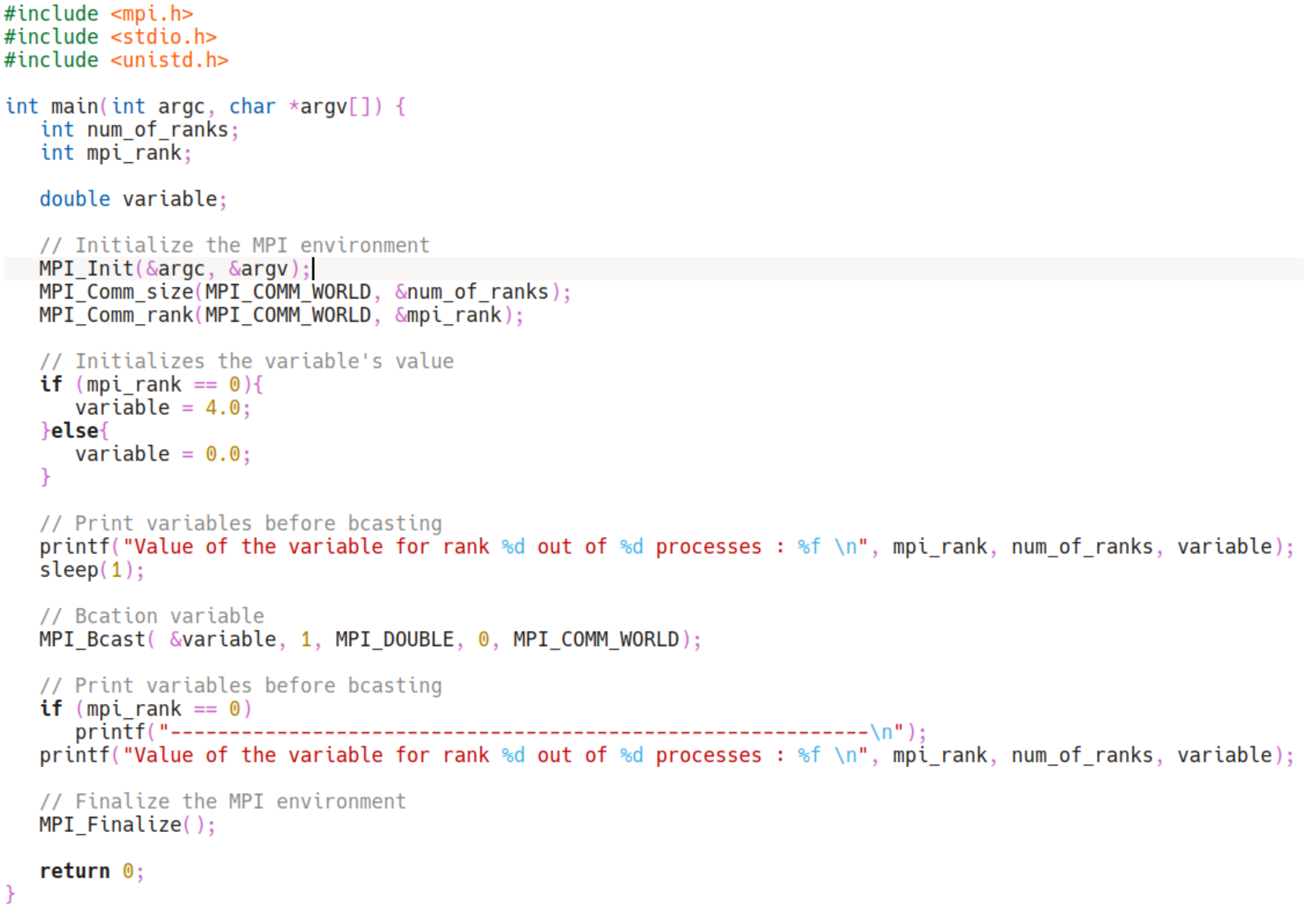
The command Bcast is used to broadcast a variable, vector or matrix, of type **MPI_datatype** to all participants of the **MPI_COMM_WORLD** environment

```
int MPI_Bcast( void *buffer,
                int count,
                MPI_Datatype datatype,
                int root,
                MPI_Comm comm)
```

Thus **root** is sending ***buffer** (you must specify the pointer to the element in the memory) of length **count** (corresponding to the number of elements).



Example: 2.1.MPI_bcast.c



```c
#include <mpi.h>
#include <stdio.h>
#include <unistd.h>

int main(int argc, char *argv[]) {
   int num_of_ranks;
   int mpi_rank;

   double variable;

   // Initialize the MPI environment
   MPI_Init(&argc, &argv);
   MPI_Comm_size(MPI_COMM_WORLD, &num_of_ranks);
   MPI_Comm_rank(MPI_COMM_WORLD, &mpi_rank);

   // Initializes the variable's value
   if (mpi_rank == 0){
      variable = 4.0;
   }else{
      variable = 0.0;
   }

   // Print variables before bcasting
   printf("Value of the variable for rank %d out of %d processes : %f \n", mpi_rank, num_of_ranks, variable);
   sleep(1);

   // Bcation variable
   MPI_Bcast( &variable, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);

   // Print variables before bcasting
   if (mpi_rank == 0)
      printf("-----------------------------------------------------------\n");
   printf("Value of the variable for rank %d out of %d processes : %f \n", mpi_rank, num_of_ranks, variable);

   // Finalize the MPI environment
   MPI_Finalize();

   return 0;
}
```

MPI Data_types

MPI datatype	C datatype
`MPI_CHAR`	`char (treated as printable character)`
`MPI_SHORT`	`signed short int`
`MPI_INT`	`signed int`
`MPI_LONG`	`signed long int`
`MPI_LONG_LONG_INT / MPI_LONG_LONG`	`signed long long int`
`MPI_SIGNED_CHAR`	`signed char (treated as integral value)`
`MPI_UNSIGNED_CHAR`	`unsigned char (treated as integral value)`
`MPI_UNSIGNED_SHORT`	`unsigned short int`
`MPI_UNSIGNED`	`unsigned int`
`MPI_UNSIGNED_LONG`	`unsigned long int`
`MPI_UNSIGNED_LONG_LONG`	`unsigned long long int`
`MPI_FLOAT`	`float`
`MPI_DOUBLE`	`double`
`MPI_LONG_DOUBLE`	`long double`

www.mpi-forum.org/docs

Vectors and matrices: 2.2.MPI_bcast_arrays.c

When communicating vectors, the first step is to communicate the number of elements to all ranks:

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

Next step is to bcast the vector:

```
MPI_Bcast( vector, number_of_elements, MPI_DOUBLE, 0, MPI_COMM_WORLD);
```

That is equivalent to:

```
MPI_Bcast( &vector[0], number_of_elements, MPI_DOUBLE, 0, MPI_COMM_WORLD);
```

Notice that the first entry is the pointer to the first element of the array.

Elements in an array or matrix are assumed to be stored in cosecutive memory allocations!

Error: 2.3.MPI_bcast_matrices_error.c

Allocation of matrices in C is a two step process

```
// 1. Allocate memory for the row pointers (an array of int*)
double **matrix = (double **)malloc(rows * sizeof(double *));

// 2. Allocate memory for the elements of each row (and initialize to 0
for (int i = 0; i < rows; i++)
{
   matrix[i] = (double *)calloc(columns, sizeof(double));
}
```

elements are not necessary contiguous, thus the simple bcast

```
MPI_Bcast(&matrix[0][0], number_of_columns*number_of_rows, MPI_DOUBLE, 0, MPI_COMM_WORLD);
```

fails in C (but works in Fortran).

www.mpi-forum.org/docs/, www.open-mpi.org

Solution 1: 2.4.MPI_bcast_matrices_vs1.c

Allocation of matrices in C is a two step process

```
// 1. Allocate memory for the row pointers (an array of int*)
double **matrix = (double **)malloc(rows * sizeof(double *));

// 2. Allocate memory for the elements of each row (and initialize to 0
for (int i = 0; i < rows; i++)
{
   matrix[i] = (double *)calloc(columns, sizeof(double));
}
```

elements are not necessary contiguous, consecutive bcasts

```
for (int i = 0; i < number_of_rows; i++)
{
   MPI_Bcast(&(matrix[i][0]), number_of_columns, MPI_DOUBLE, 0, MPI_COMM_WORLD);
}
```

solve the problem! Because the columns are contiguous elements in the memory!

Solution 2: 2.5.MPI_bcast_matrices_vs2.c

We can force the allocation to be contiguous by constructing the matrix as (arrays.h)

```
/* allocate the n*m contiguous items */
double *temp_matrix = (double *)malloc(rows * columns * sizeof(double));

/* allocate the row pointers into the memory */
double **matrix = (double **)malloc(rows * sizeof(double *));

/* set up the pointers into the contiguous memory */
for (int i = 0; i < rows; i++)
   matrix[i] = &(temp_matrix[i * columns]);
```

The pointers that were now defined through the temporary array are contiguous, thus the MPI_bcast call

```
MPI_Bcast(&matrix[0][0], number_of_columns*number_of_rows, MPI_DOUBLE, 0, MPI_COMM_WORLD);
```

Is succesful!

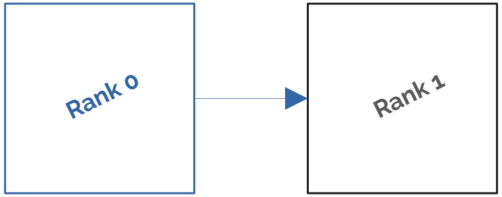
UNIVERSITY OF
LUXEMBOURG

MPI_Send/MPI_Recv

Introduction to MPI programming in C – 2025/2026
Lesson 1

Example: 3.1.MPI_send_recv_arrays.c

The commands MPI_Send/MPI_Recv are used for point-to-point communications between ranks



```
int MPI_Send(const void *buf,
                  int count,
       MPI_Datatype datatype,
                    int dest,
                     int tag,
               MPI_Comm comm )
```

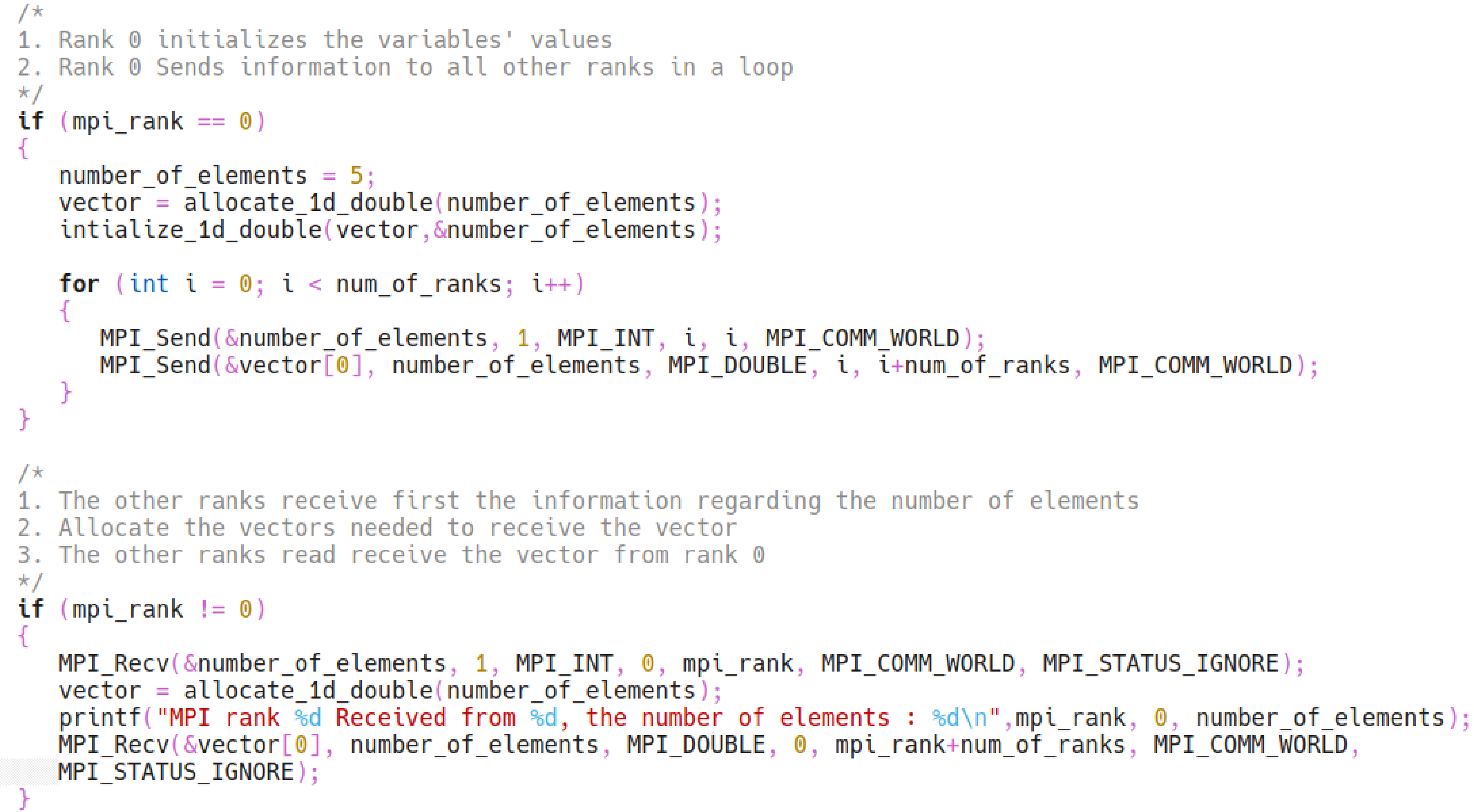
```
int MPI_Recv(void *buf,
             int count,
 MPI_Datatype datatype,
            int source,
               int tag,
         MPI_Comm comm,
     MPI_Status *status)
```

The MPI task that calles the command MPI_send, sends the buffer to the dest, with a tag that is used to recognize the communication (Many communications can be executed between the same ranks).

The **task that executes the MPI_recv command awaits for the communcation** from the source with a given tag.

www.mpi-forum.org/docs/, www.open-mpi.org

Example: 3.1.MPI_send_recv_arrays.c



```
/*
1. Rank 0 initializes the variables' values
2. Rank 0 Sends information to all other ranks in a loop
*/
if (mpi_rank == 0)
{
   number_of_elements = 5;
   vector = allocate_1d_double(number_of_elements);
   intialize_1d_double(vector,&number_of_elements);

   for (int i = 0; i < num_of_ranks; i++)
   {
      MPI_Send(&number_of_elements, 1, MPI_INT, i, i, MPI_COMM_WORLD);
      MPI_Send(&vector[0], number_of_elements, MPI_DOUBLE, i, i+num_of_ranks, MPI_COMM_WORLD);
   }
}

/*
1. The other ranks receive first the information regarding the number of elements
2. Allocate the vectors needed to receive the vector
3. The other ranks read receive the vector from rank 0
*/
if (mpi_rank != 0)
{
   MPI_Recv(&number_of_elements, 1, MPI_INT, 0, mpi_rank, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
   vector = allocate_1d_double(number_of_elements);
   printf("MPI rank %d Received from %d, the number of elements : %d\n",mpi_rank, 0, number_of_elements);
   MPI_Recv(&vector[0], number_of_elements, MPI_DOUBLE, 0, mpi_rank+num_of_ranks, MPI_COMM_WORLD,
   MPI_STATUS_IGNORE);
}
```

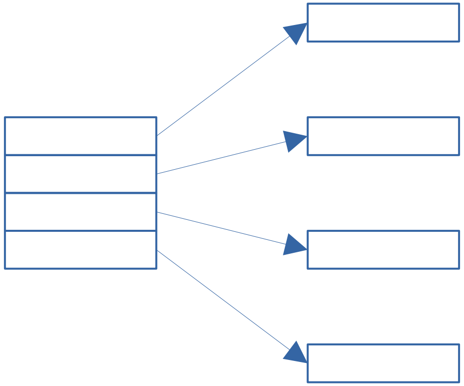
MPI_Scatter/MPI_Gather

Introduction to MPI programming in C – 2025/2026
Lesson 1

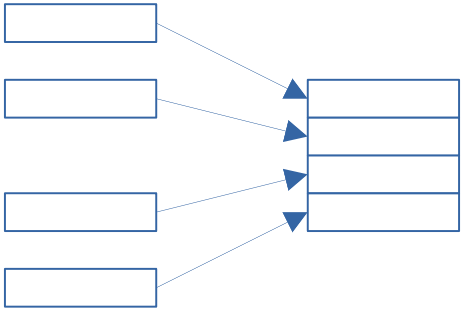
Example: 4.1.MPI_scatter_gather_arrays.c

The command MPI_Scatter is used to partition the sending buffer (usually array or matrix) on all ranks of the MPI_COMM_WORLD.

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



The command MPI_Gather is used to collect the sending buffers to a recivieving one.



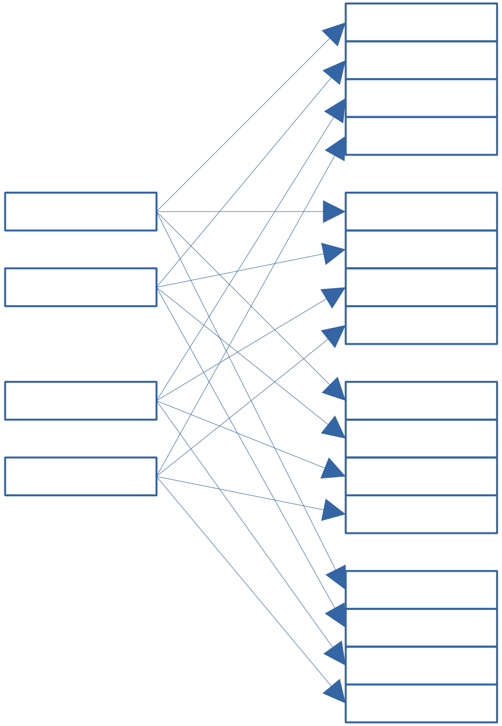
```
int MPI_Gather(const void *sendbuf,
                     int sendcount,
             MPI_Datatype sendtype,
                     void *recvbuf,
                     int recvcount,
             MPI_Datatype recvtype,
                          int root,
                     MPI_Comm comm)
```

www.mpi-forum.org/docs/, www.open-mpi.org

Example: 4.1.MPI_scatter_gather_arrays.c

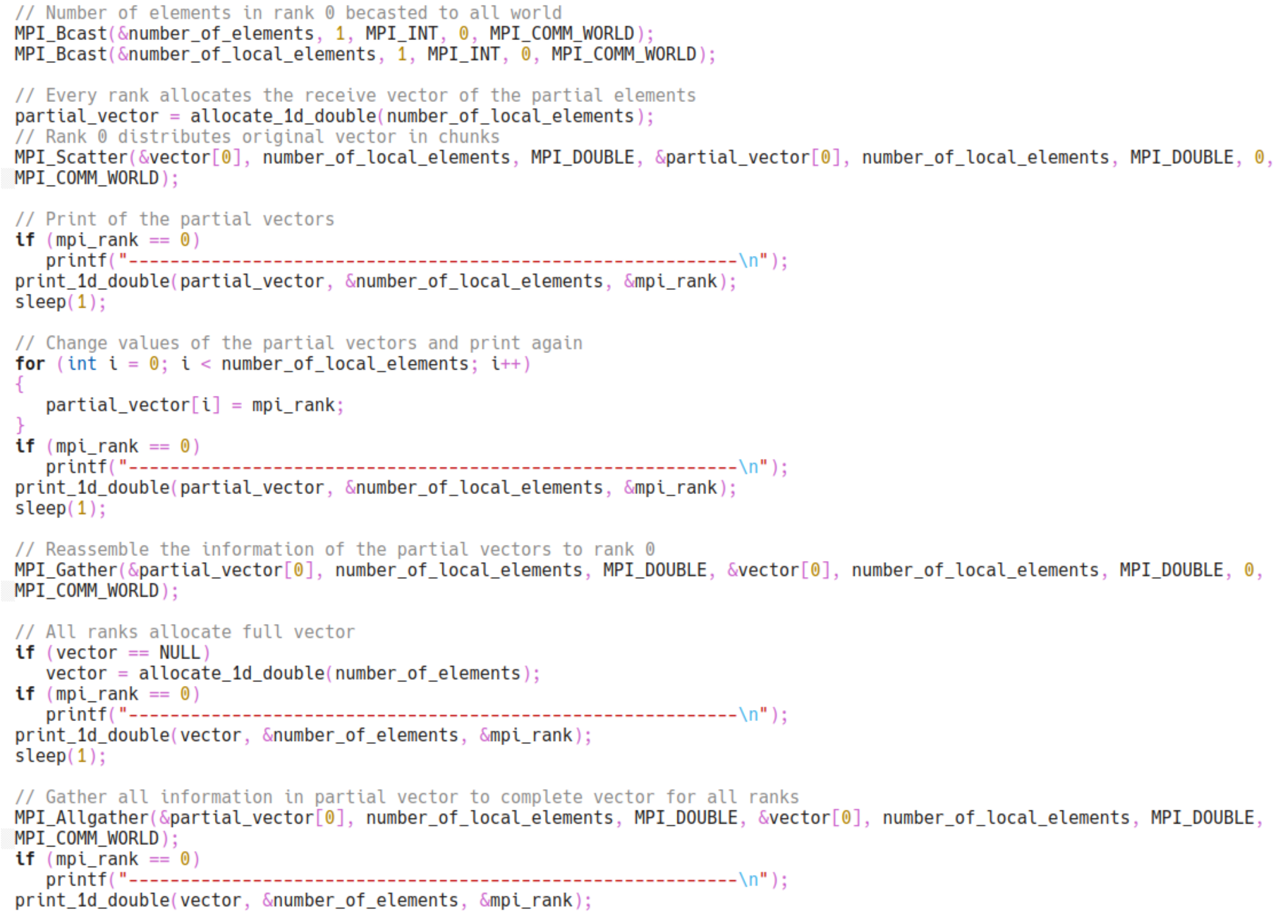
The command MPI_Allgather is used to collect the sending buffers for all ranks.

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



www.mpi-forum.org/docs/, www.open-mpi.org

Example: 4.1.MPI_scatter_gather_arrays.c



```
// Number of elements in rank 0 becasted to all world
MPI_Bcast(&number_of_elements, 1, MPI_INT, 0, MPI_COMM_WORLD);
MPI_Bcast(&number_of_local_elements, 1, MPI_INT, 0, MPI_COMM_WORLD);

// Every rank allocates the receive vector of the partial elements
partial_vector = allocate_1d_double(number_of_local_elements);
// Rank 0 distributes original vector in chunks
MPI_Scatter(&vector[0], number_of_local_elements, MPI_DOUBLE, &partial_vector[0], number_of_local_elements, MPI_DOUBLE, 0,
MPI_COMM_WORLD);

// Print of the partial vectors
if (mpi_rank == 0)
   printf("-----------------------------------------------------------\n");
print_1d_double(partial_vector, &number_of_local_elements, &mpi_rank);
sleep(1);

// Change values of the partial vectors and print again
for (int i = 0; i < number_of_local_elements; i++)
{
   partial_vector[i] = mpi_rank;
}
if (mpi_rank == 0)
   printf("-----------------------------------------------------------\n");
print_1d_double(partial_vector, &number_of_local_elements, &mpi_rank);
sleep(1);

// Reassemble the information of the partial vectors to rank 0
MPI_Gather(&partial_vector[0], number_of_local_elements, MPI_DOUBLE, &vector[0], number_of_local_elements, MPI_DOUBLE, 0,
MPI_COMM_WORLD);

// All ranks allocate full vector
if (vector == NULL)
   vector = allocate_1d_double(number_of_elements);
if (mpi_rank == 0)
   printf("-----------------------------------------------------------\n");
print_1d_double(vector, &number_of_elements, &mpi_rank);
sleep(1);

// Gather all information in partial vector to complete vector for all ranks
MPI_Allgather(&partial_vector[0], number_of_local_elements, MPI_DOUBLE, &vector[0], number_of_local_elements, MPI_DOUBLE,
MPI_COMM_WORLD);
if (mpi_rank == 0)
   printf("-----------------------------------------------------------\n");
print_1d_double(vector, &number_of_elements, &mpi_rank);
```

uni.lu
UNIVERSITY OF
LUXEMBOURG

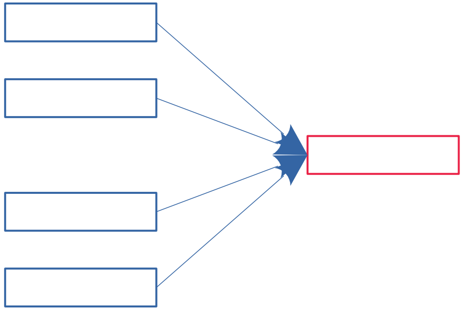
MPI_Reduce /MPI_Allreduce

Introduction to MPI programming in C – 2025/2026
Lesson 1

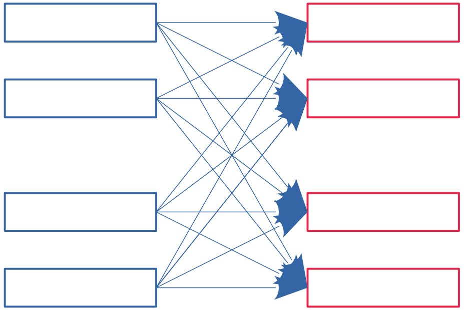
Example: 5.1.MPI_reduce_allreduce.c

The commands MPI_Reduce and MPI_Allreduce are used to perform the operation `MPI_Op op` executed over all MPI ranks, on the data from the sendbuf and collected on the recvbuf.

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



While in the case of MPI_Reduce the result is only made available to the rank 0, or root, in the case of `MPI_Allreduce` all MPI ranks receive the information:



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

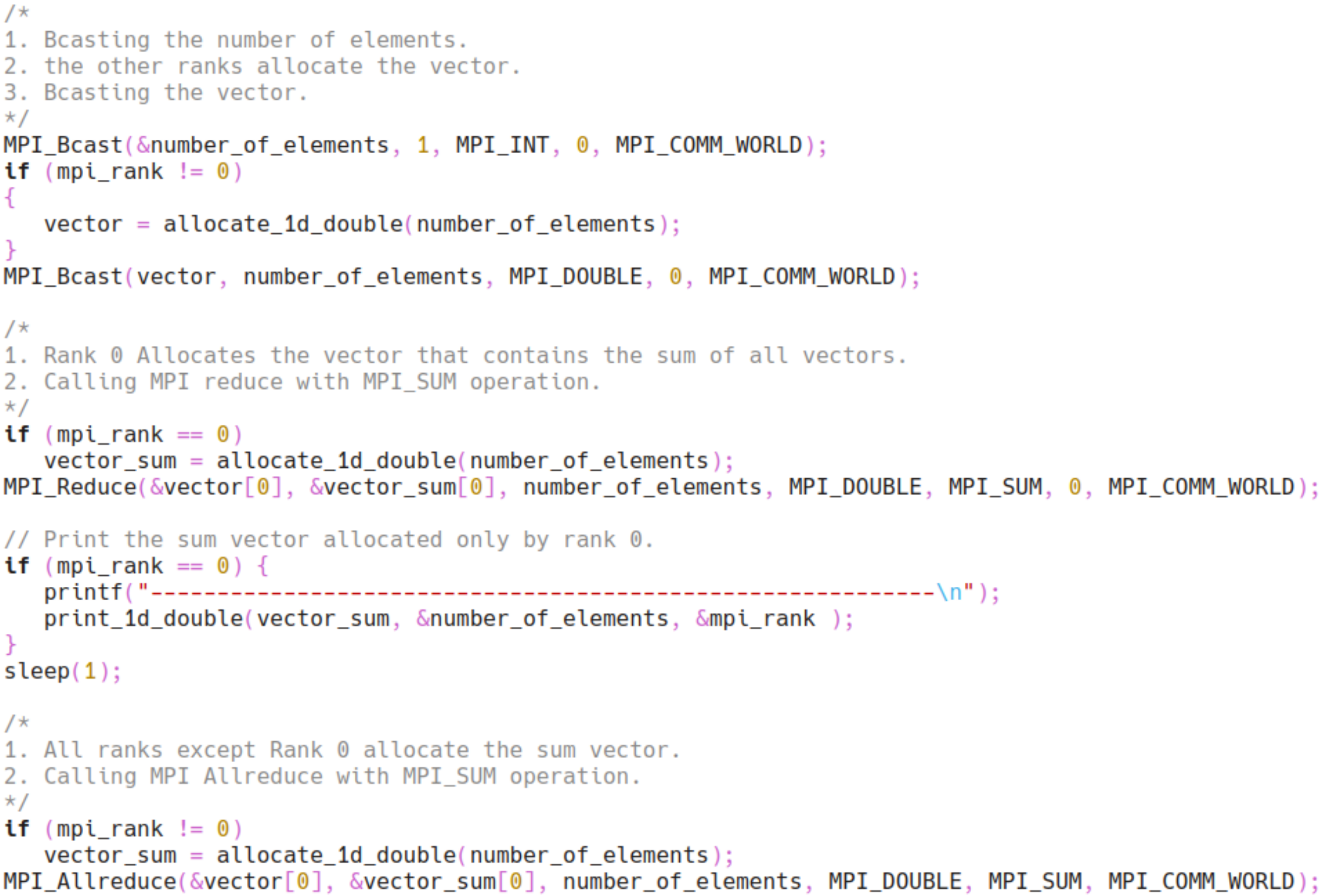
www.mpi-forum.org/docs/, www.open-mpi.org

MPI Operations

MPI Operations	Explanation
MPI_OP_NULL	dummy operation
MPI_MAX	Find the maximum value within ranks
MPI_MIN	Find the minimum value within ranks
MPI_SUM	Sum variables along all ranks
MPI_PROD	Mulitply variables along all ranks
MPI_LAND	Logical and
MPI_BAND	Bit-wise and
MPI_LOR	Logical or
MPI_BOR	Bit-wise or
MPI_LXOR	Logical xor (exclusive OR)
MPI_BXOR	Bit-wise xor
MPI_MINLOC	Computes min value and its location
MPI_MAXLOC	Computes max value and its location

www.mpi-forum.org/docs

Example: 5.1.MPI_reduce_allreduce.c



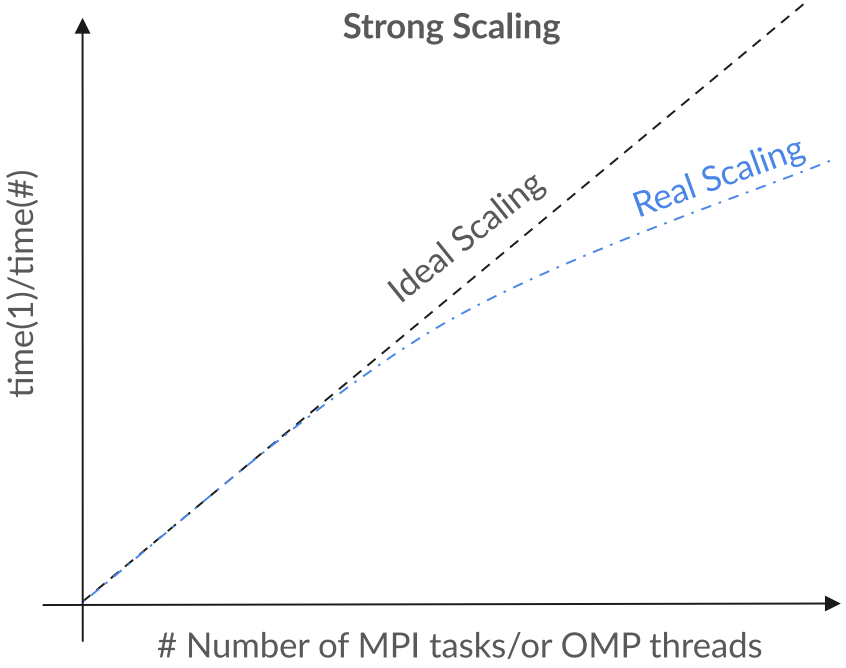
```
/*
1. Bcasting the number of elements.
2. the other ranks allocate the vector.
3. Bcasting the vector.
*/
MPI_Bcast(&number_of_elements, 1, MPI_INT, 0, MPI_COMM_WORLD);
if (mpi_rank != 0)
{
   vector = allocate_1d_double(number_of_elements);
}
MPI_Bcast(vector, number_of_elements, MPI_DOUBLE, 0, MPI_COMM_WORLD);

/*
1. Rank 0 Allocates the vector that contains the sum of all vectors.
2. Calling MPI reduce with MPI_SUM operation.
*/
if (mpi_rank == 0)
   vector_sum = allocate_1d_double(number_of_elements);
MPI_Reduce(&vector[0], &vector_sum[0], number_of_elements, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);

// Print the sum vector allocated only by rank 0.
if (mpi_rank == 0) {
   printf("-----------------------------------------------------------\n");
   print_1d_double(vector_sum, &number_of_elements, &mpi_rank );
}
sleep(1);

/*
1. All ranks except Rank 0 allocate the sum vector.
2. Calling MPI Allreduce with MPI_SUM operation.
*/
if (mpi_rank != 0)
   vector_sum = allocate_1d_double(number_of_elements);
MPI_Allreduce(&vector[0], &vector_sum[0], number_of_elements, MPI_DOUBLE, MPI_SUM, MPI_COMM_WORLD);
```

uni.lu
UNIVERSITY OF
LUXEMBOURG

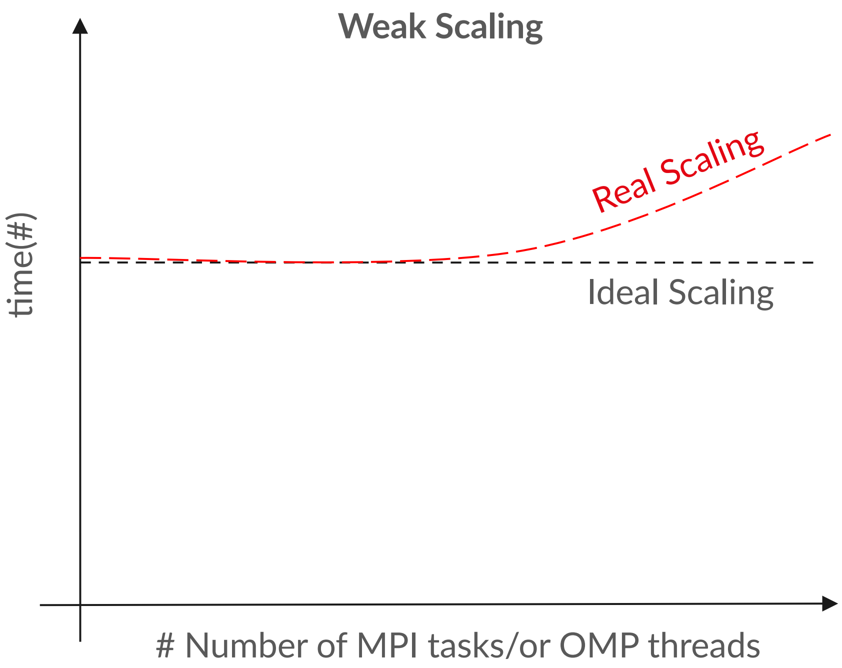
Performances

Introduction to MPI programming in C – 2025/2026
Lesson 1

Basic scaling laws



The total task is partitioned over the cores



Each core does the same task

Questions?

Section Title

Title

Footer notes