

# HPC.NRW

# MPI in Small Bites

## PPCES 2025

HPC.NRW Competence Network



**EDIH**  
Rheinland



THE COMPETENCE NETWORK FOR HIGH PERFORMANCE COMPUTING IN  
NRW

# MPI Concepts

HPC.NRW Competence Network

## MPI in Small Bites

- MPI is implemented as a library, not a compiler extension
  - Common (non-local) objects need coordinated construction
  - Library needs to be initialized explicitly
- Multiple methods exist to initialize MPI
  - Classic MPI (pre-MPI 4.0) without threads → `MPI_Init`
  - Classic MPI (pre-MPI 4.0) with threads → `MPI_Init_thread`
    - Covered in another part on hybrid programming
  - New MPI (MPI 4.0) with threads → `MPI_Session_init`
    - Covered in another part on the session model

# Library Initialization (classic MPI – no threads)

- Start-up, initialisation, finalization, and shutdown – C

- 1 Inclusion of the MPI header file
- 2 Pre-initialisation mode: uncoordinated
  - **No MPI function calls allowed with few exceptions**
  - **All program instances run exactly the same code**
- 3 Initialisation of the MPI environment with implicit synchronisation
- 4 Parallel MPI code
- 5 Finalisation of the MPI environment
- 6 Post-finalisation mode: uncoordinated
  - **No MPI function calls allowed with few exceptions**

```
1 #include <mpi.h>
2 {
  // ... some code ...
3 MPI_Init(&argc, &argv);
4 // ... computation & communication ...
5 MPI_Finalize();
  // ... wrap-up ...
6 return 0;
}
```

# Library Initialization (classic MPI – no threads)

- Start-up, initialisation, finalisation, and shutdown – Fortran

- 1 Using the MPI module
- 2 Pre-initialisation mode: uncoordinated
  - **No MPI function calls allowed with few exceptions**
  - **All program instances run exactly the same code**
- 3 Initialisation of the MPI environment with implicit synchronisation
- 4 Parallel MPI code
- 5 Finalisation of the MPI environment
- 6 Post-finalisation mode: uncoordinated
  - **No MPI function calls allowed with few exceptions**

```
1 PROGRAM example
  USE mpi_f08 ! USE mpi
2
3 ! ... some code ...
  INTEGER :: ierr
3 CALL MPI_Init(ierr)
4
5 ! ... computation & communication ...
5 CALL MPI_Finalize(ierr)
6
6 ! ... wrap-up ...
  END
```

Fortran

## - Initialization:

```
C:   ierr = MPI_Init(&argc, &argv);  
Fortran: CALL MPI_Init(ierr)
```

- Initializes the MPI library and makes the process member of MPI\_COMM\_WORLD
- [C] Both arguments must be either NULL or they *must* point to the arguments of main()
- **May not be called more than once for the duration of the program execution**
- Error code as return value in [C] and additional parameter in [F]

## - Finalization:

```
C:   ierr = MPI_Finalize();  
Fortran: CALL MPI_Finalize(ierr)
```

- Cleans up the MPI library and prepares the process for termination
- **Must be called once before the process terminates**
- Having other code after the finalisation call is not recommended

# General Structure of an MPI Program

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

1 Obtains the number of processes (ranks) in the MPI program

Example: if the job was started with 4 processes, then **numberOfProcs** will be set to 4 by the call

2 Obtains the identity of the calling process within the MPI program  
**NB: MPI processes are numbered starting from 0**

Example: if there are 4 processes in the job, then **rank** receives the value of 0 in the first process, 1 in the second process, etc.

```
#include <mpi.h>

int main(int argc, char **argv)
{
    // ... some code ...
    int ierr = MPI_Init(&argc, &argv);
    int numberOfProcs, rank;
    // ... more code ...
    1 ierr = MPI_Comm_size(MPI_COMM_WORLD,
        &numberOfProcs);
    2 ierr =
    MPI_Comm_rank(MPI_COMM_WORLD,
        &rank);
    // ... computation & communication ...
    ierr = MPI_Finalize();
    // ... wrap-up ...
    return 0;
}
```

# General Structure of an MPI Program

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

1 Obtains the number of processes (ranks) in the MPI program

Example: if the job was started with 4 processes, then **numberOfProcs** will be set to 4 by the call

2 Obtains the identity of the calling process within the MPI program  
**NB: MPI processes are numbered starting from 0**

Example: if there are 4 processes in the job, then **rank** receives the value of 0 in the first process, 1 in the second process, etc.

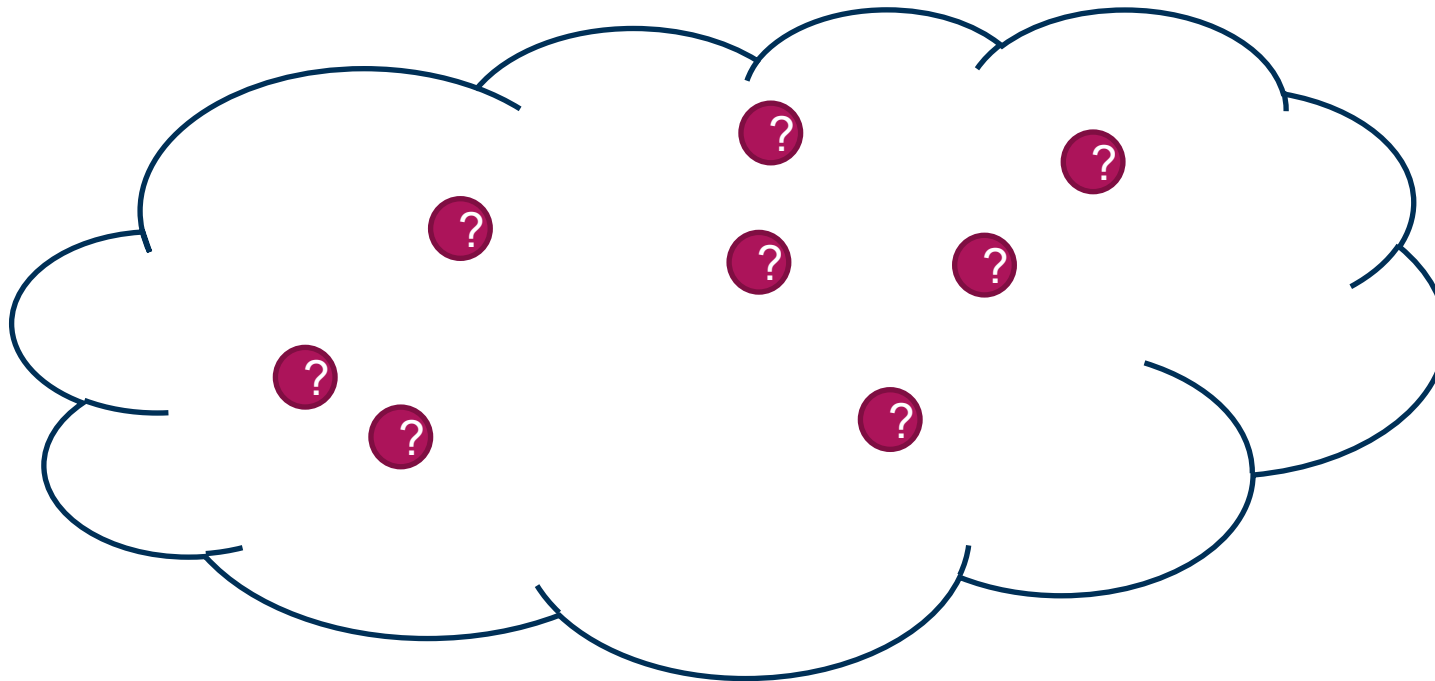
Fortran

## PROGRAM example

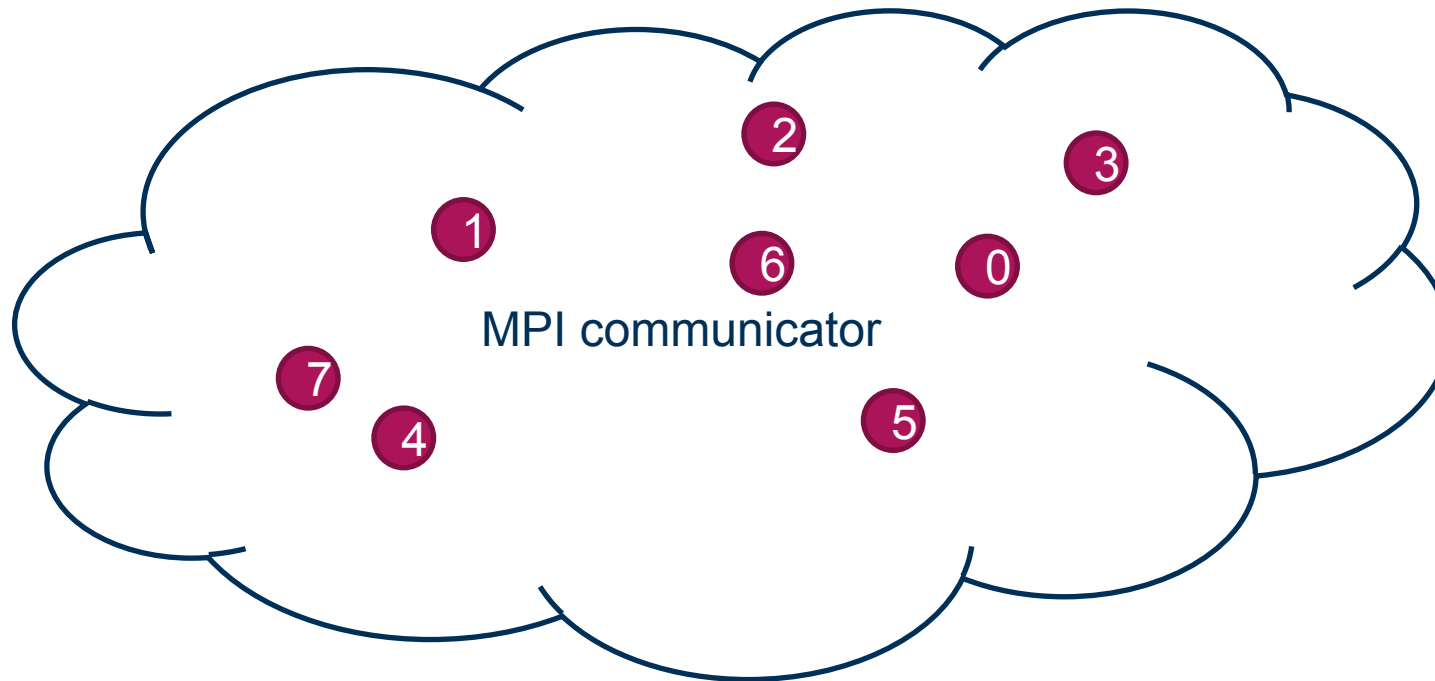
```
USE mpi_f08 ! USE mpi
INTEGER :: rank, numberOfProcs, ierr
! ... some code ...
CALL MPI_Init(ierr)
! ... other code ...
CALL
1 MPI_Comm_size(MPI_COMM_WORLD,&
  numberOfProcs, ierr)
2 CALL
  MPI_Comm_rank(MPI_COMM_WORLD,&
    rank, ierr)
! ... computation & communication ...
CALL MPI_Finalize(ierr)
! ... wrap-up ...
END PROGRAM example
```



- The processes in any MPI program are initially indistinguishable
- MPI assigns each process a unique identity (**rank**) in a communication context (**communicator**)



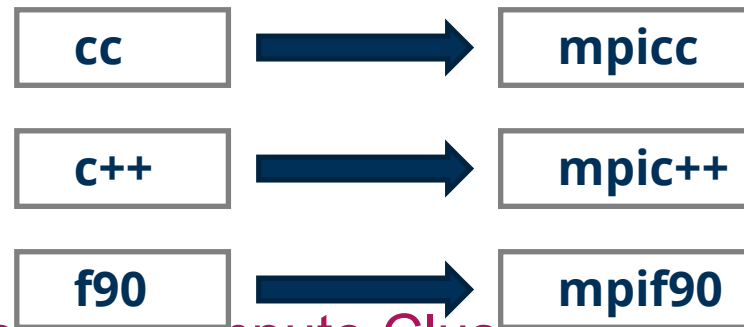
- The processes in any MPI program are initially indistinguishable
- MPI assigns each process a unique identity (**rank**) in a communication context (**communicator**)



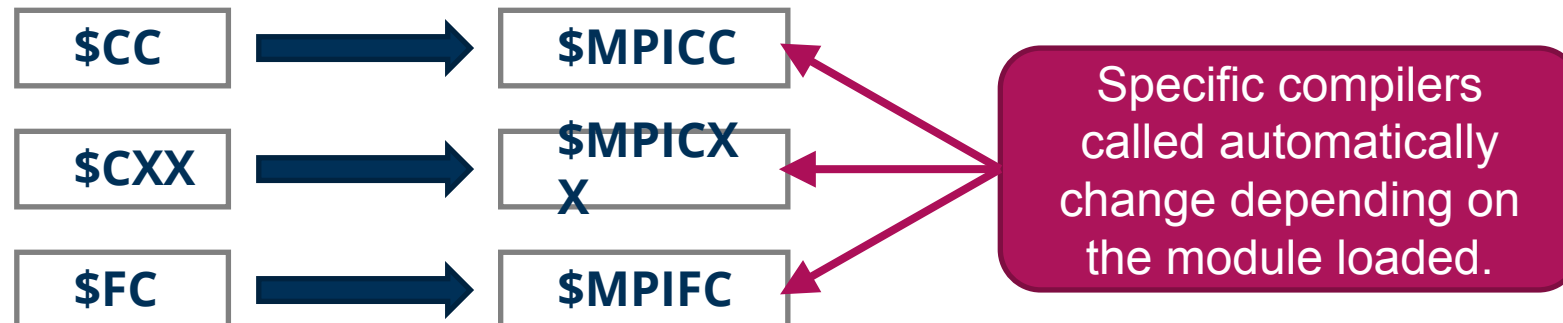
- The processes in any MPI program are initially indistinguishable (for the user)
- MPI assigns each process a unique identity (**rank**) in a communication context (**communicator**)
- Ranks
  - Range from 0 to n-1 (with n processes in the communicator)
  - An MPI process can have different ranks in different communicators
- Communicators
  - Logical contexts where communication takes place
    - Comprises a group of MPI processes with some additional information
  - **MPI\_COMM\_WORLD** is implicitly available
    - Comprises all processes initially started with the MPI program

- ✓ Provide dynamic identification of all peers
  - Who am I and who else is also working on this problem?
2. Provide robust mechanisms to exchange data
  - Whom to send data to / From whom to receive the data?
  - How much data?
  - What kind of data?
  - Has the data arrived?
3. Provide synchronisation mechanisms
  - Have all processes reached same point in the program execution flow?
4. Provide methods to launch and control a set of processes
  - How do we start multiple processes and get them to work together?
5. Portability

- MPI is a typical library with C header files, Fortran modules, etc.
- Most MPI vendors provide convenience compiler wrappers (names are not standardized!)



- On the RWTH Aachen Compute Cluster:



# Example: MPI Compiler Wrappers

- RWTH Aachen Cluster defines additional environment variables to minimize confusion

```
cluster:~> $MPICC -show # instruct wrapper to show compile line
icc \
-l"/cvmfs/[...]/impi/2021.6.0-intel-compilers-2022.1.0/mpi/2021.6.0/include" \
-L"/cvmfs/[...]/impi/2021.6.0-intel-compilers-2022.1.0/mpi/2021.6.0/lib/release" \
-L"/cvmfs/[...]/impi/2021.6.0-intel-compilers-2022.1.0/mpi/2021.6.0/lib" \
-Xlinker --enable-new-dtags -Xlinker -rpath \
-Xlinker "/cvmfs/[...]/impi/2021.6.0-intel-compilers-2022.1.0/mpi/2021.6.0/lib/release" \
-Xlinker -rpath \
-Xlinker "/cvmfs/[...]/impi/2021.6.0-intel-compilers-2022.1.0/mpi/2021.6.0/lib" \
-Impifort -Impi -ldl -lrt -lpthread
cluster:~> echo $MPICC # check compiler wrapper name
mpiicc
cluster:~> module purge; module load gompi # switch MPI implementation
cluster:~> echo $MPICC # check compiler wrapper name again
mpicc
```

# Example: MPI Compiler Wrappers

- RWTH Aachen Cluster defines additional environment variables to minimize confusion

```
cluster:~> module purge && module load gomp  
cluster:~> echo $MPICC  
mpicc  
cluster:~> $MPICC -show  
gcc \  
-I/cvmfs/[...]/OpenMPI/4.1.4-GCC-11.3.0/include \  
-L/cvmfs/[...]/OpenMPI/4.1.4-GCC-11.3.0/lib \  
-L/cvmfs/[...]/hwloc/2.7.1-GCCcore-11.3.0/lib \  
[...]  
-Wl,/cvmfs/[...]/libevent/2.1.12-GCCcore-11.3.0/lib -Wl,--enable-new-dtags -lmpi
```

- Most MPI implementations provide a special launcher program:

```
mpiexec -n nprocs ... program <arg1> <arg2> <arg3> ...
```

- Launches `nprocs` instances of **program** with command-line arguments **arg1, arg2, ...**
- The standard specifies the **mpiexec** program, but does not require it:
  - IBM BG/Q: **runjob --np 1024 ...**
  - SLURM resource manager: **srun -n 96 -N 1 ...**



- The launcher often performs more than simply launching processes:
  - Helps MPI processes find each other and establish the world communicator
  - Redirects the standard output of all ranks to the terminal
  - Redirects the terminal input to the standard input of rank 0
  - Forwards received signals (Unix-specific)

- ✓ Provide dynamic identification of all peers
  - Who am I and who else is also working on this problem?
- 2. Provide robust mechanisms to exchange data
  - Whom to send data to / From whom to receive the data?
  - How much data?
  - What kind of data?
  - Has the data arrived?
- 3. Provide synchronisation mechanisms
  - Have all processes reached same point in the program execution flow?
- ✓ Provide methods to launch and control a set of processes
  - How do we start multiple processes and get them to work together?
- ✓ Portability

- Error codes indicate the success of the operation:
  - Failure is indicated by error codes other than **MPI\_SUCCESS**

```
if (MPI_SUCCESS != MPI_Init(&argc, &argv))  
...
```

C

```
CALL MPI_Init(ierr)  
IF (ierr /= MPI_SUCCESS) ...
```

Fortran

- An MPI error handler is called first before the call returns
  - **The default error handler for non-I/O calls aborts the entire MPI program!**
  - Error checking in simple programs is redundant
- Actual MPI error code values are implementation specific
  - Use **MPI\_Error\_string** to derive human readable information

- MPI objects (e.g., communicators) are referenced via handles
  - **Process-local values**
    - Cannot be passed from one process to another
  - **Objects referenced by handles are opaque**
    - Structure is implementation dependent
    - Blackbox for the user
- C (mpi.h)
  - **typedef'd handle types: MPI\_Comm, MPI\_Datatype, MPI\_File, etc.**

- Fortran (USE mpi)
  - All handles are INTEGER values
  - Easy to pass the wrong handle type
- Fortran 2008 (USE mpi\_f08)
  - Wrapped INTEGER values: TYPE(MPI\_Comm), TYPE(MPI\_File), etc.
  - The INTEGER handle is still available: comm%MPI\_VAL

- MPI is a library
  - Cannot infer datatypes of supplied buffers at runtime
  - User needs to provide additional information on buffer type
- MPI datatype handles tell the MPI library how to:
  - read binary values from the send buffer
  - write binary values into the receive buffer
  - correctly apply value alignments
  - convert between machine representations in heterogeneous environments

- MPI datatypes are **handles**
  - Cannot be used to declare variables of a specific language type
  - `sizeof(MPI_INT)` provides the size of a datatype handle **NOT** the size of an int in C
- Type Signatures
  - Sequence of basic datatypes in a buffer
  - Basic datatypes correspond to native language datatypes
- Type Maps
  - Sequence of basic datatypes **AND** their location in a buffer
- **Type signatures of associated operations have to match; Type map may differ!**

- MPI provides predefined datatypes for each language binding:

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

MPI data type	Fortran data type
MPI_INTEGER	INTEGER
MPI_REAL	REAL
MPI_REAL8	REAL(KIND=8)
MPI_DOUBLE_PRECISION	DOUBLE PRECISION
MPI_COMPLEX	COMPLEX
MPI_LOGICAL	LOGICAL
MPI_CHARACTER	CHARACTER(1)
...	...
MPI_BYTE	-

8 binary digits  
no conversion  
used for untyped data



- Non-local procedures may require,
  - during its execution,
  - some specific, semantically-related MPI procedure
  - to be called on another MPI process”
- Local procedure are not non-local

- MPI defines several **operations**, which are
  - a sequence of steps
  - performed by the MPI library
  - to establish and enable
    - data transfer
    - and/or synchronization
- Four stages
  1. Initialization – Resources (argument lists, buffer address, etc.) are handed to the MPI library
  2. Starting – The operation takes over control of the resources (buffer contents)
  3. Completion – Return control of the resources (buffer contents)
  4. Freeing – Return control of the remaining resources

- Blocking procedures return when the associated operation is complete locally
  - Any input argument can be safely reused or deallocated
  - Operation may not be completed remotely
- Non-blocking procedures return before associated operation is complete locally
  - One or more additional calls are needed to complete operation
  - Input arguments may not be written or deallocated until operation is complete
- Synchronous operations complete locally only with specific remote intervention
  - Asynchronous operations may complete locally without remote intervention



Point-to-Point  
Communication



Collective  
Communication



One-sided  
Communication