Computação em Larga Escala

Introduction to Message Passage Interface (MPI)

Eurico Pedrosa

António Rui Borges

Universidade de Aveiro - DETI

2025-03-22

What is MPI?

MPI: Message Passing Interface

- **MPI** is a **library specification** that supports the **message-passing programming model**, where data is explicitly transferred between processes through cooperative operations.
- It is **not a programming language**, but a **standardized API**. In C/C++ and Fortran, MPI functions are defined in the header file `mpi.h` and provided through language bindings.
- MPI enables **inter-process communication** by moving data from the memory space of one process to another.
- It provides a **portable and efficient foundation** for vendors to implement optimized communication routines — sometimes with hardware acceleration — to ensure performance and **scalability**.
- All MPI constants and functions are prefixed with `MPI_`. For example, in C/C++, MPI programs include the header `mpi.h`.

Anatomy of a MPI program

Example

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

int main (int argc, char *argv[])
{
    MPI_Init(&argc, &argv); // <- Always the first instruction

    int i, rank, size;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);  // <- number of the process
    MPI_Comm_size(MPI_COMM_WORLD, &size);  // <- number of processes spawn

    std::cout << "Hello! I am " << rank << " of " << size << std::endl;

    MPI_Finalize(); // <- Always the last instruction
    return EXIT_SUCCESS;
}
```

Compilation and Execution

To compile use `mpic++`:

```
$ mpic++ -Wall -O3 hello.cpp -o hello
```

To spawn 4 processes:

```
$ mpiexec -n 4 ./hello
```

```
Hello! I am 0 of 4
Hello! I am 1 of 4
Hello! I am 2 of 4
Hello! I am 3 of 4
```

Error Handling

- **All MPI functions in C/C++ return an error code** indicating the status of the operation.
- **Error handlers** can be attached to:
  - **Communicators**
  - **Windows**
  - **Files**
- **Unhandled function calls** are assumed to be associated with `MPI_COMM_WORLD`.

Error Handling

Predefined Error Handlers:

- `MPI_ERRORS_ARE_FATAL` (default): Aborts **all processes** upon error — equivalent to `MPI_Abort`.
- `MPI_ERRORS_RETURN`: Returns the error code to the user — **program continues execution**.

⚠️ Each process can set its own error handler for each object — the association is **local**, not collective.

With MPI_ERRORS_ARE_FATAL


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

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

    int rank, size;
    MPI_Comm_rank (MPI_COMM_WORLD, &rank);
    MPI_Comm_size (MPI_COMM_WORLD, &size);

    if (rank == 1)
        MPI_Init (&argc, &argv); // <- error! can only be called once

    std::cout << "Hello! I am " << rank << " of " << size << std::endl;

    MPI_Finalize();
    return EXIT_SUCCESS;
}
```

With MPI_ERRORS_ARE_FATAL

```
$ mpic++ -Wall -O3 error.cpp -o error
$ mpiexec -n 2 ./error
```

```
Hello! I am 0 of 2
Abort(1068099599) on node 1: Fatal error in internal_Init: Other MPI error,
error stack:
internal_Init(70): MPI_Init(argc=0x16bd82ee4, argv=0x16bd82ed8) failed
internal_Init(50): Cannot call MPI_INIT or MPI_INIT_THREAD more than once
```

With MPI_ERRORS_RETURN

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

int main (int argc, char *argv[])
{
    int stat, rank, size;
    char errMessage[100]; int errMessLen;
    MPI_Init (&argc, &argv);
    MPI_Comm_rank (MPI_COMM_WORLD, &rank);
    MPI_Comm_size (MPI_COMM_WORLD, &size);
    if (rank == 1) {
        MPI_Comm_set_errhandler (MPI_COMM_WORLD, MPI_ERRORS_RETURN);
        if ((stat = MPI_Init (&argc, &argv) & 0xFF) != MPI_SUCCESS) {
            switch (stat) {
                case MPI_ERR_COMM: std::cerr << "Invalid communicator!" << std::endl;
                    break;
                case MPI_ERR_OTHER: MPI_Error_string (stat, errMessage, &errMessLen);
                    std::cerr << errMessage << ": MPI_Init called more than once!" << std::endl;
                    break;
            }
            MPI_Abort (MPI_COMM_WORLD, EXIT_FAILURE);
        }
    }
    std::cout << "Hello! I am " << rank << " of " << size << std::endl;
    MPI_Finalize ();
    return EXIT_SUCCESS;
}
```

With MPI_ERRORS_RETURN


```
$ mpic++ -Wall -O3 error-return.cpp -o error-return
$ mpiexec -n 2 ./error-return
```

```
Hello! I am 0 of 4
Hello! I am 2 of 4
Hello! I am 3 of 4
Hello! I am 1 of 4
Other MPI error: MPI_Init called more than once!
Abort(1) on node 1 (rank 1 in comm 0): application called
MPI_Abort(MPI_COMM_WORLD, 1) - process 1
```

Programming Language Data Types

- **MPI is a library interface specification** with **language bindings** for C/C++ and Fortran.
- It defines a set of **predefined data types** used in communication routines (e.g., `MPI_INT`, `MPI_FLOAT`).

Mapping MPI Types to Language Types:

- **MPI data types** serve as **formal types** describing the data passed in MPI functions.
- **Programming language types** (e.g., `int`, `float` in C/C++) are the **actual types** used to store values in user code.
- This mapping ensures **type safety** and **portable message exchange** across different platforms.

Programming Language Data Types

Matching MPI Predefined Data Types to C/C++ Basic Types

MPI Data Type	C Data Type
`MPI_CHAR`	`char`
`MPI_SHORT`	`short`
`MPI_INT`	`int`
`MPI_LONG`	`long`
`MPI_UNSIGNED_CHAR`	`unsigned char`
`MPI_UNSIGNED_SHORT`	`unsigned short`
`MPI_UNSIGNED`	`unsigned int`
`MPI_UNSIGNED_LONG`	`unsigned long`
`MPI_FLOAT`	`float`
`MPI_DOUBLE`	`double`

✅ **Use these MPI datatypes in communication calls** (e.g., `MPI_Send`, `MPI_Recv`) to correctly describe the memory layout of the corresponding C/C++ data.

Special MPI Data Types

MPI provides two special predefined data types that **do not map directly** to standard C or Fortran types:

`MPI_BYTE`

- Represents a **raw 8-bit byte**.
- Often used for **binary data** or file I/O.
- In C/C++, can be thought of as `unsigned char`.

Special MPI Data Types

📦 `MPI_PACKED`

- Used for **manual packing** of complex, non-contiguous data structures.
- Treated as a **byte array** containing serialized data.
- Enables sending **heterogeneous data** from scattered memory regions in a **single message**.

> *i* Info
>
> Useful for advanced scenarios where performance or layout control is needed — e.g., manual message composition.

Communicator Concept

Communicators in MPI

- A **communicator** defines a **communication context** — an isolated "universe" where communication occurs.
- **Messages are matched only within the same communicator**; contexts prevent interference between unrelated communications.

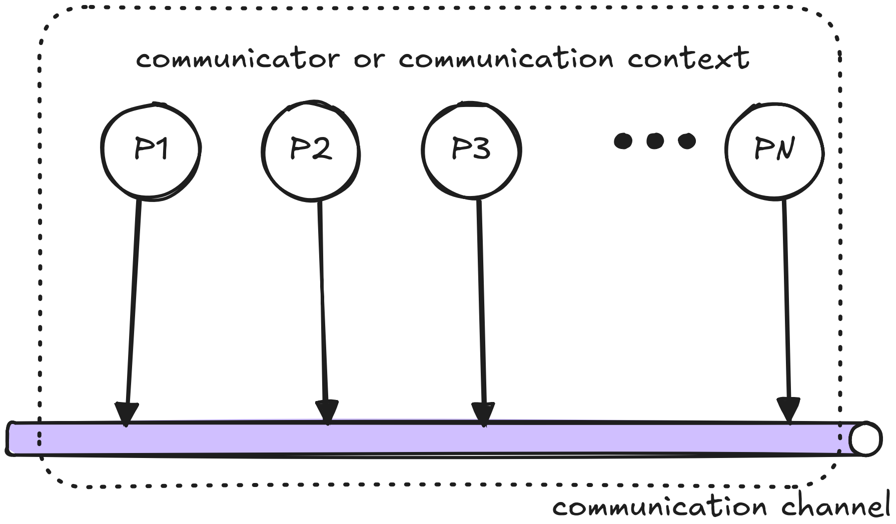
Key Properties:

- A communicator includes a **group of processes**.
- Each process in the group has a unique **rank** (0 to `size - 1`), used for addressing.
- The **process group is ordered** and fixed for the communicator.

Communicators in MPI

🧩 MPI_COMM_WORLD

- A **predefined communicator** available after `MPI_Init`.
- Includes **all processes** launched in the MPI job.
- Most basic MPI programs operate within this communicator.



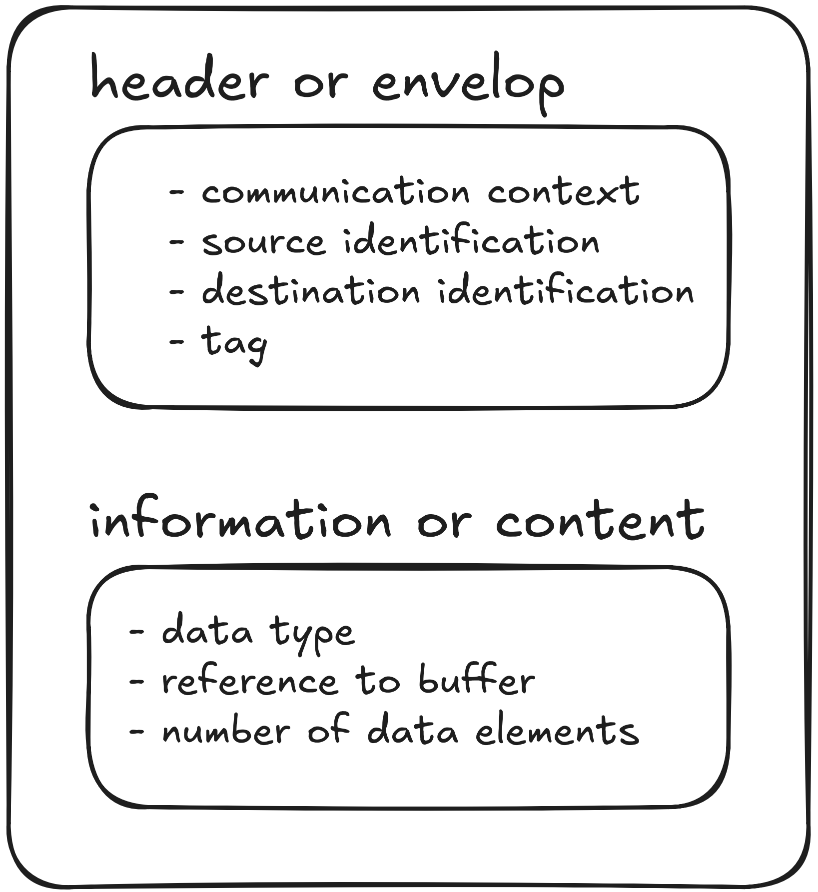
Communicator Size and Process Identification

- After `MPI_Init`, the **size of the communicator** is determined by the number of **spawned processes** (e.g., via `mpiexec -n N`).
- Within a communicator, each process is identified by a unique **rank**:
  - Ranges from `0` to `size - 1`
  - Used in **point-to-point and collective operations** to specify source/destination

These ranks are **local to the communicator's process group** — the same process may have different ranks in different communicators.

MPI Messsages

Format


header or envelop
- communication context
- source identification
- destination identification
- tag
information or content
- data type
- reference to buffer
- number of data elements

MPI Message Header (Envelope)

Each MPI message includes a **header**, also called the **envelope**, which contains metadata used for message matching:

Header Fields:

- **Communication Context** Specifies the **communicator** — the set of processes within which the message exchange occurs.
- **Source Identification** The **rank** of the sending process within the communicator.
  - Implicit in **send** operations
  - Explicitly used in **receive** operations

MPI Message Header (Envelope)

- **Destination Identification** The **rank** of the target process that receives the message.
  - Implicit in **receive** operations
  - Explicitly used in **send** operations
- **Tag** An **integer label** used to distinguish between different types of messages.
  - Valid range: `0` to `MPI_TAG_UB`
  - `MPI_TAG_UB` is **implementation-defined**, retrievable via `MPI_Comm_get_attr`.

MPI Message: Information Content

In addition to the header, each MPI message contains **payload data**, described by the following parameters:

Message Content Parameters

- **Data Type** The **MPI datatype** (e.g., `MPI_INT`, `MPI_FLOAT`) describing the type of elements in the message.
- **Buffer Reference** A **pointer** to the memory location where data is **stored (send)** or **received into**.
- **Count** The **number of elements** of the given datatype to be transferred.
  - Can be `0` — in which case the **message is empty**, but still valid.

Point-to-point Communication

P2P Communication in MPI

- A **point-to-point communication** occurs when one process (**source**) sends a message to another (**destination**).

Standard Communication Mode

- **`MPI_Send` and `MPI_Recv`** are the most basic forms of point-to-point communication.
- They are **blocking operations**:
  - `MPI_Send` **blocks** until the message is **safely sent** (typically, until it is received or buffered).
  - `MPI_Recv` **blocks** until the message is **received** and the buffer is filled.

MPI P2P Communication Primitives

📨 MPI_Send

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

📥 MPI_Recv

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

Parameter Descriptions

Parameter	**Description**
`buf`	Pointer to the **data buffer** (send or receive)
`count`	Number of elements to send/receive
`datatype`	MPI datatype (e.g., `MPI_INT`, `MPI_FLOAT`)
`dest`	Rank of the **destination** process (for `MPI_Send`)
`source`	Rank of the **source** process (for `MPI_Recv`) Can use `MPI_ANY_SOURCE` to match any
`tag`	Message **tag** to differentiate types of messages Can use `MPI_ANY_TAG` to match any
`comm`	Communicator identifying the **communication context**
`status`	Pointer to an `MPI_Status` struct Includes `MPI_SOURCE`, `MPI_TAG`, `MPI_ERROR` Use `MPI_STATUS_IGNORE` if not needed

Notes

- `MPI_Send` and `MPI_Recv` are **blocking** by default.
- `MPI_Status` is useful for determining **who sent the message**, what **tag** it had, and whether **errors** occurred.

Example

```cpp
#include <iostream>
#include <cstring>
#include <mpi.h>

int main (int argc, char *argv[])
{
    MPI_Init(&argc, &argv);
    int rank;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);

    if (rank == 0) {
        char data[] = "I am here!";
        std::cout << "Transmitted message: " << data << std::endl;
        MPI_Send(data, std::strlen (data), MPI_CHAR, 1, 0, MPI_COMM_WORLD);
    } else if (rank == 1) {
        int i;
        char* recData = new char(100);
        for (i = 0; i < 100; i++)
            recData[i] = '\0';

        MPI_Recv(recData, 100, MPI_CHAR, 0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        std::cout << "Received message: " << recData << std::endl;
    }

    MPI_Finalize ();
    return EXIT_SUCCESS;
}
```

Example

```
$ mpic++ -Wall -O3 sendRecData.cpp -o error
$ mpiexec -n 2 ./sendRecData

Transmitted message: I am here!
Received message: I am here!
```

Suggested Reading

- The Art of HPC by Victor Eijkhout of TACC
  - Volume 2: Parallel Programming for Science and Engineering
- MPICH User's Guide
  - https://www.mpich.org/documentation/guides/