Week-09 MPI Basics

What is MPI (Message Passing Interface)?

MPI is a **standardized and portable message-passing system** used to program **parallel computers**. It's mainly designed for **distributed memory systems**, like clusters, where each processor has its own local memory.

- MPI is used in **high-performance computing (HPC)** environments.
- It enables **communication between separate processes** that may be on the same or different machines.
- Programs using MPI are often written in **C, C++, or Fortran**.
- It offers explicit control over communication, which gives you **power and performance**, but also **complexity**.

What is OpenMP (Open Multi-Processing)?

OpenMP is an **API for shared memory multiprocessing**, typically used on **multi-core systems** where all threads share the same memory.

- It's a set of **compiler directives**, library routines, and environment variables.
- Used mainly with **C, C++, and Fortran**.
- Ideal for **parallelizing loops** and other CPU-intensive operations.
- Easier to implement than MPI — just add `#pragma omp` directives and go.

✅ MPI vs OpenMP – Head-to-Head

Feature	MPI	OpenMP
Memory Model	Distributed memory	Shared memory
Communication	Explicit message passing	Implicit through shared memory
Scalability	Very scalable (can run across clusters)	Limited to single-node, shared-memory systems
Ease of Use	Complex – need to manage messages & processes	Simple – compiler directives handle most of it
Best Use Case	Large-scale HPC apps (e.g., weather modeling)	Multithreaded apps on multi-core CPUs
Languages	C, C++, Fortran	C, C++, Fortran

Feature	MPI	OpenMP
Parallelism Type	Process-based	Thread-based
Fault Tolerance	Harder to manage	Easier (due to shared memory)
Portability	High – works across machines	Limited – needs shared memory

---

🔑 TL;DR:

- Use **MPI** when working on **distributed systems or clusters** and you need **fine-grained control**.
- Use **OpenMP** when you're on a **single machine with multiple cores** and want **quick parallelism** without the communication headache.

🔧 MPI Hello World in C

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

int main(int argc, char** argv) {
    int world_rank;
    int world_size;

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

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

    // Get the rank (ID) of this process
    MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);

    // Print out a message from each process
    printf("Hello from process %d of %d\n", world_rank, world_size);

    // Finalize the MPI environment
    MPI_Finalize();
    return 0;
}
```

💡 What's Happening Here?

- `MPI_Init` sets up the MPI environment.
- `MPI_Comm_size` gives you the total number of processes running.
- `MPI_Comm_rank` gives you the unique ID of the current process.
- Each process prints its own message.
- `MPI_Finalize` shuts down the MPI environment.

🧪 How to Compile and Run

1. **Compile:**

   ```
   mpicc hello_mpi.c -o hello_mpi
   ```

2. **Run with 4 processes:**

   ```
   mpirun -np 4 ./hello_mpi
   ```

✅ Sample Output

```
Hello from process 0 of 4
Hello from process 1 of 4
Hello from process 2 of 4
Hello from process 3 of 4
```

Each line comes from a different process running in parallel.

✅ Step-by-Step: Setting Up MPI on WSL2 (Ubuntu)

🧰 1. Update Your System

```
sudo apt update && sudo apt upgrade -y
```

📦 2. Install OpenMPI

```
sudo apt install -y openmpi-bin libopenmpi-dev
```

This installs:

- `mpicc` — the MPI C compiler wrapper
- `mpirun` — used to launch MPI programs
- All core MPI libraries

🧪 3. Test It Works

Create a sample MPI program:

```
nano hello_mpi.c
```

Paste this code:

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

int main(int argc, char** argv) {
    int world_rank, world_size;
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);
    MPI_Comm_size(MPI_COMM_WORLD, &world_size);
    printf("Hello from process %d of %d\n", world_rank, world_size);
    MPI_Finalize();
    return 0;
}
```

⚙️ 4. Compile It

```
mpicc hello_mpi.c -o hello_mpi
```

🚀 5. Run It

```
mpirun -np 4 ./hello_mpi
```

Expected output:

```
Hello from process 0 of 4
Hello from process 1 of 4
Hello from process 2 of 4
Hello from process 3 of 4
```

If you see that, your MPI setup is good to go.

✅ Minimal MPI Skeleton:

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

int main(int argc, char** argv) {
    MPI_Init(&argc, &argv); // Start MPI
```

```
    // Your parallel logic goes here...

    MPI_Finalize();         // End MPI
    return 0;
}
```

`MPI_Init(&argc, &argv);` is the **first required call** in every MPI program. It initializes the MPI environment and prepares the system for communication between processes.

🔍 What it does:

- **Initializes** the MPI runtime.
- **Sets up communication** between all participating processes.
- Accepts the command-line arguments (`argc`, `argv`) so MPI can process any MPI-specific flags (though you usually don't pass any).

✅ Syntax:

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

💡 Key Points:

- Must be called **once and only once** by each process.
- **Must be called before any other MPI function.**
- Often followed by:

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

- Ends with `MPI_Finalize();` to clean up.

🔁 Minimal MPI Program Using `MPI_Init`:

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

int main(int argc, char** argv) {
    MPI_Init(&argc, &argv); // Start MPI

    int rank;
```

```
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    printf("Hello from process %d\n", rank);

    MPI_Finalize(); // Clean up
    return 0;
}
```

🔷 `MPI_COMM_WORLD`

This is a **predefined communicator** in MPI that includes **all the processes** in your MPI program.

Think of it as:

💬 *"The global group where every process can talk to each other."*

- When you call `mpirun -np 4`, `MPI_COMM_WORLD` represents all 4 of those processes.
- Every process gets a **unique ID** (called **rank**) within this communicator.

🔷 `MPI_Comm_size(MPI_COMM_WORLD, &world_size);`

This function tells you **how many total processes** are in a communicator.

✅ Usage:

```
int world_size;
MPI_Comm_size(MPI_COMM_WORLD, &world_size);
```

- Fills `world_size` with the number of processes in `MPI_COMM_WORLD`.
- If you launched with `mpirun -np 4`, then `world_size == 4`.

🔷 Related: `MPI_Comm_rank(MPI_COMM_WORLD, &rank);`

This tells you **which process you are** (your ID).

- `rank == 0` → First process
- `rank == 1` → Second process
- etc.

🧠 Analogy:

Think of `MPI_COMM_WORLD` like a classroom of students:

- `MPI_Comm_size()` tells you **how many students are in the room**.
- `MPI_Comm_rank()` tells you **who you are (your seat number)**.

📌 Summary:

MPI Function	Purpose
`MPI_COMM_WORLD`	Default communicator (all processes)
`MPI_Comm_size(comm, &size)`	Get number of processes in communicator
`MPI_Comm_rank(comm, &rank)`	Get ID (rank) of the calling process
