

MPI Tutorial 1: Getting Started with MPI

Introduction to MPI

Processes, Ranks, Program Launch, and Point-to-Point Communication

Learning outcomes: compile, launch, inspect ranks, and exchange a value with `MPI_Send` and `MPI_Recv`

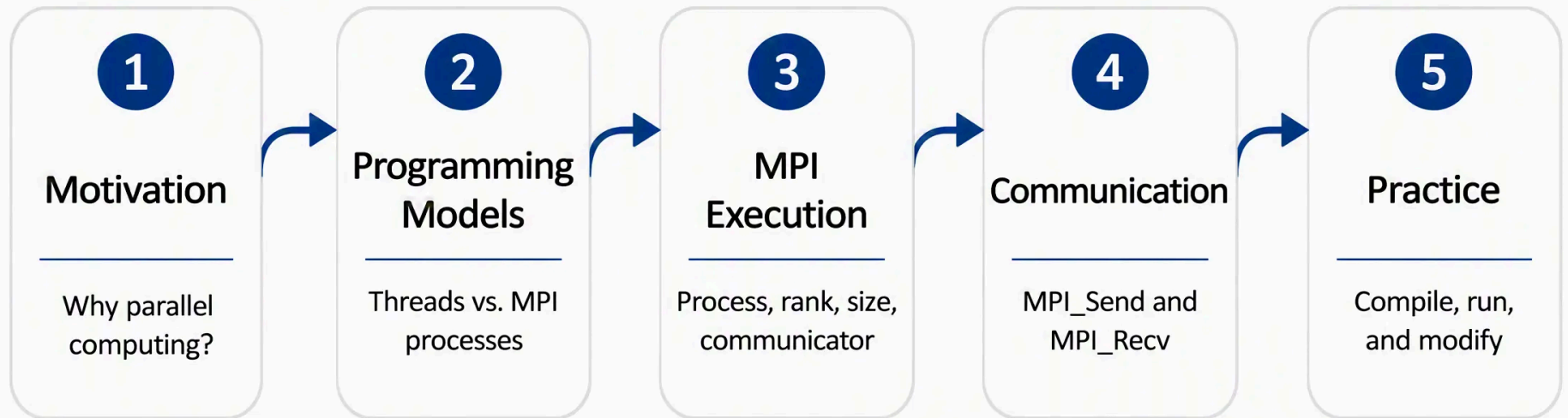
Tutorial 1 of 2 — collective communication is covered in Tutorial 2

What Will You Learn Today?

By the end of this tutorial, you should be able to:

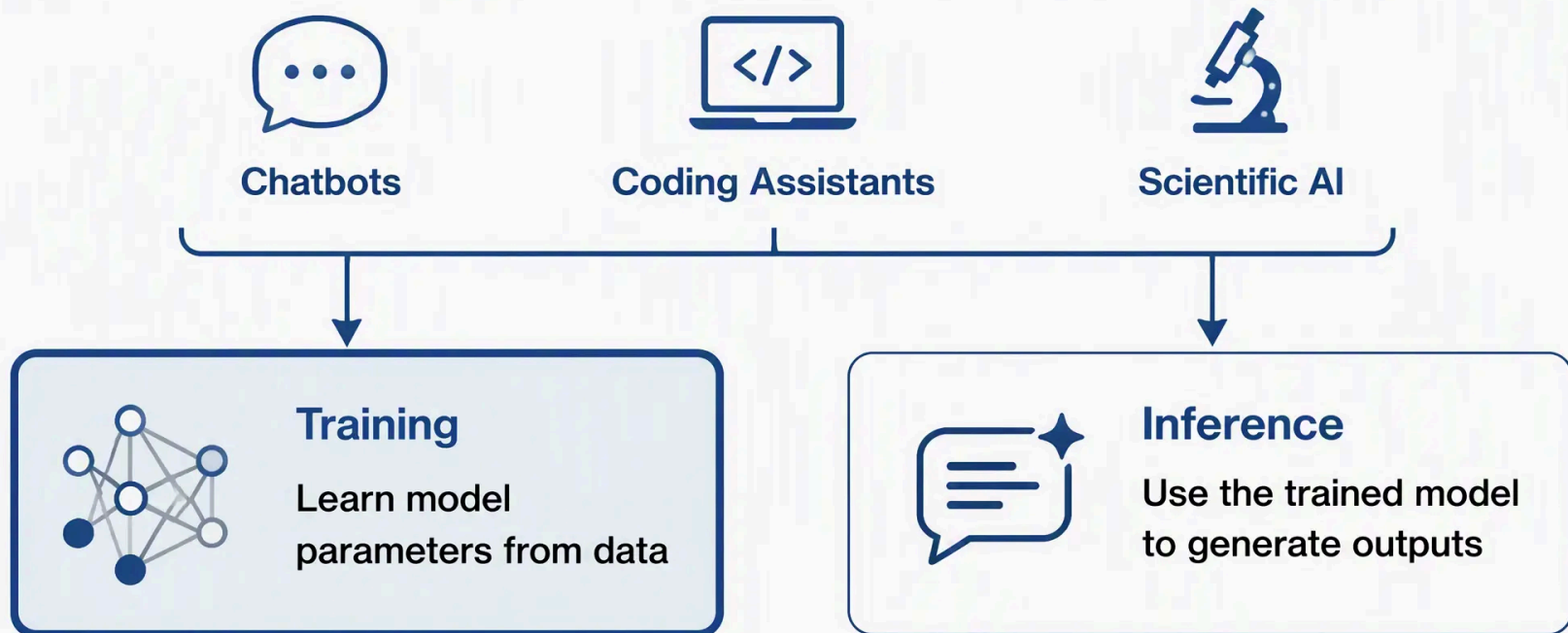
- Explain why MPI is useful for parallel computing
- Compare shared-memory threads with MPI processes
- Compile and launch one MPI program as multiple processes
- Exchange data using `MPI_Send` and `MPI_Recv`

Our Learning Roadmap



Why Parallel Computing?

Modern AI systems rely on large language models.

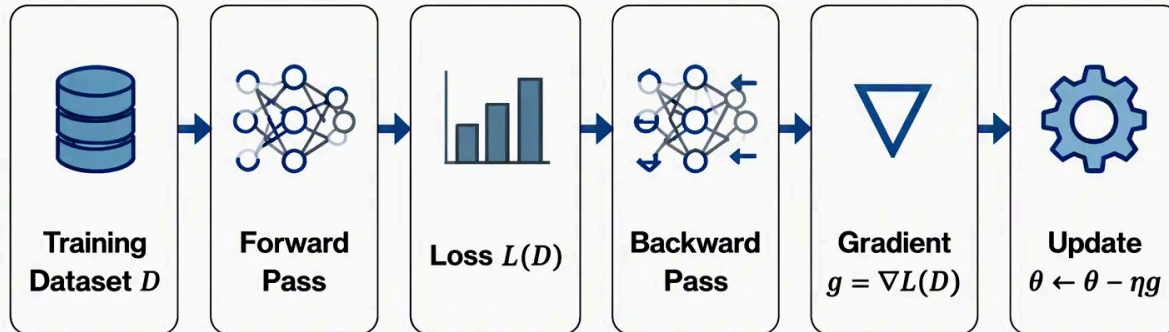


Both training and inference use parallel computing.

We focus on training because gradient computation makes task decomposition easy to see.

Why Is LLM Training Expensive?

One Training Step



Training at Scale



Large training dataset



Repeated training iterations

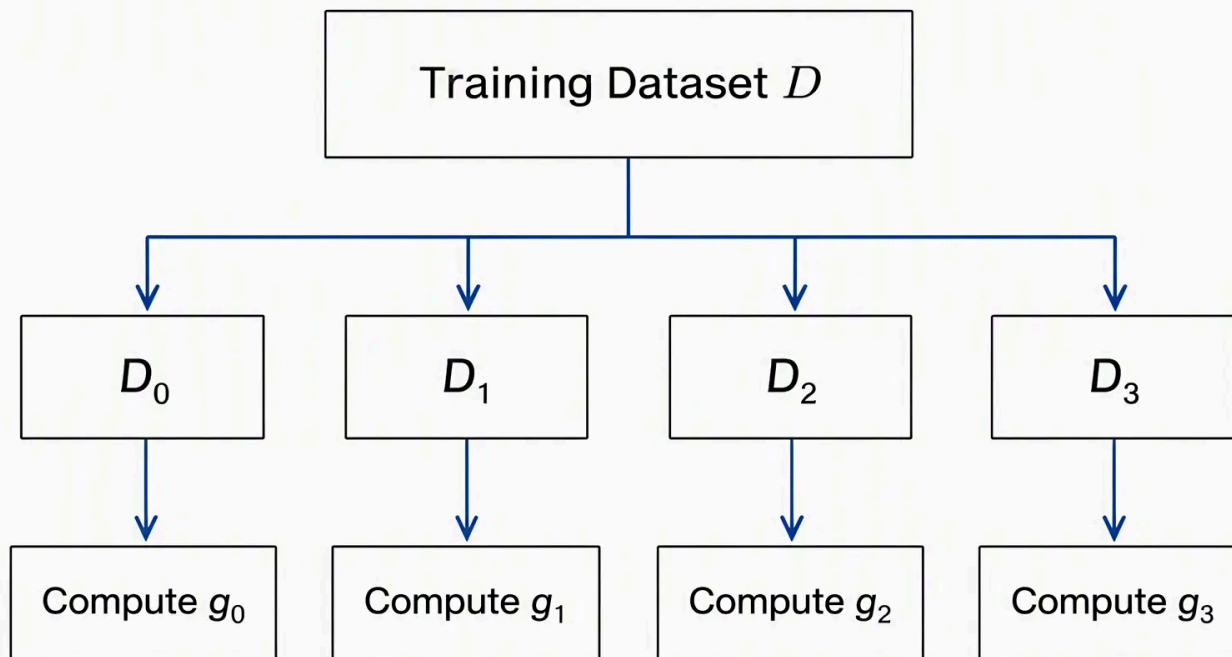


Large model parameters

Can we split the gradient computation across multiple CPU workers?

Large, repeated computation creates the need for parallelism.

Where Is the Parallelism?

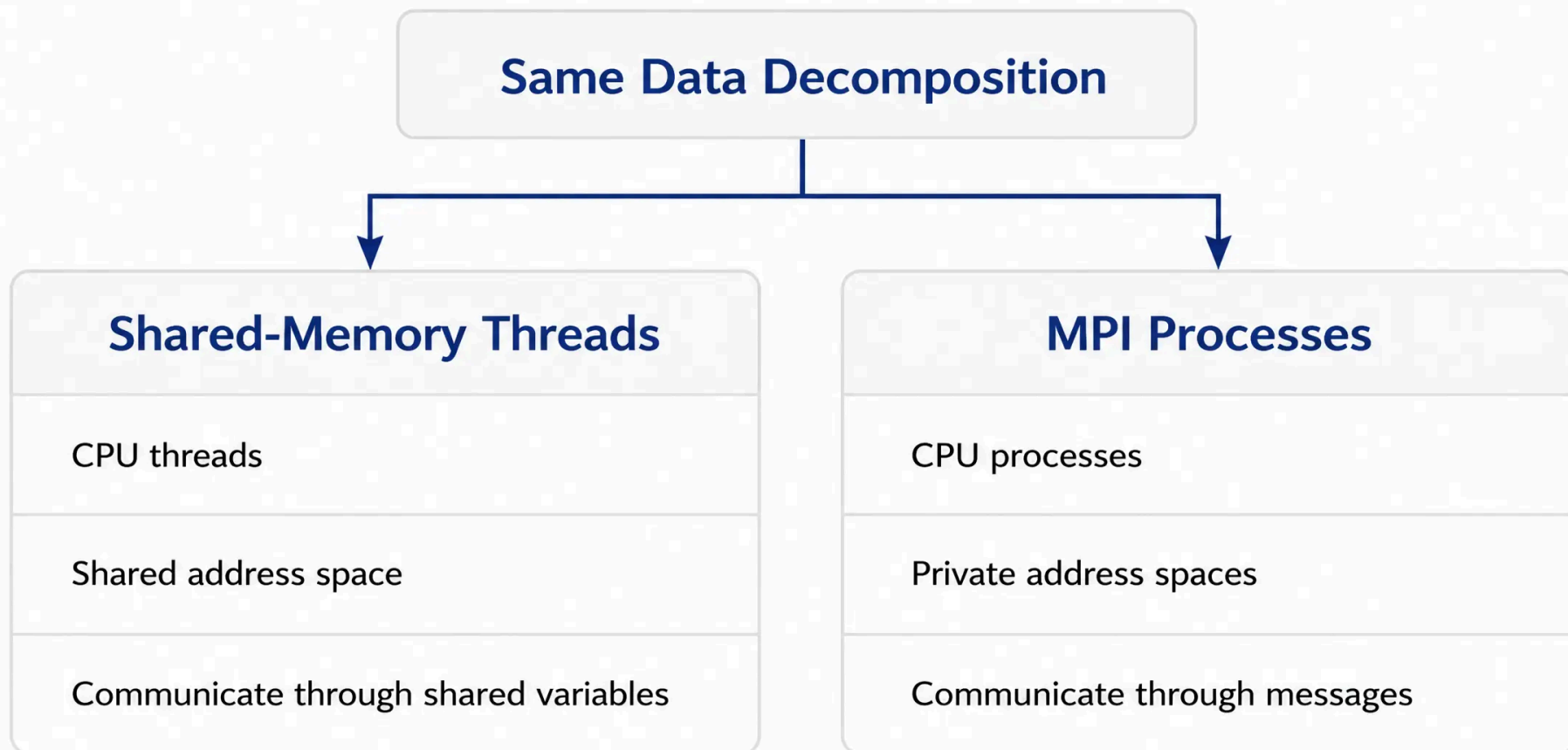


Can these local gradients be computed at the same time?

$$g_i = \nabla L(D_i)$$

Parallel programming begins by identifying independent work.

Same Work, Different Programming Models

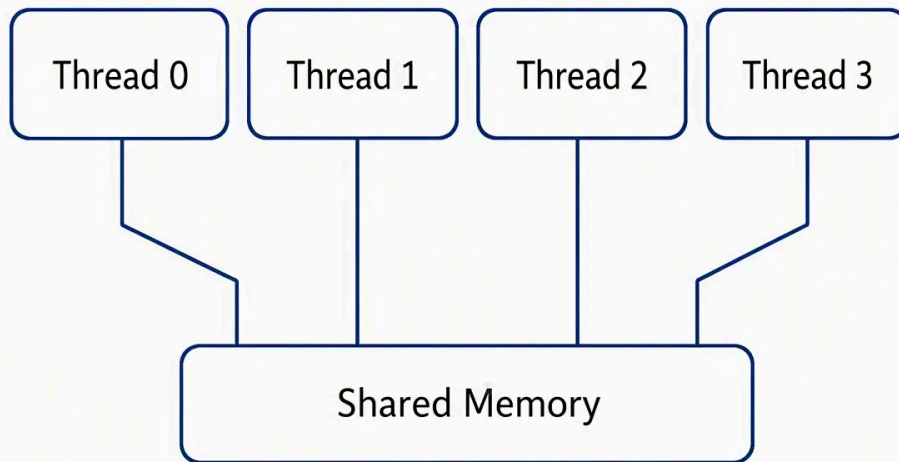


Same decomposition; different rules for data access and communication.

Shared-Memory Threads

Multiple threads execute within one process.

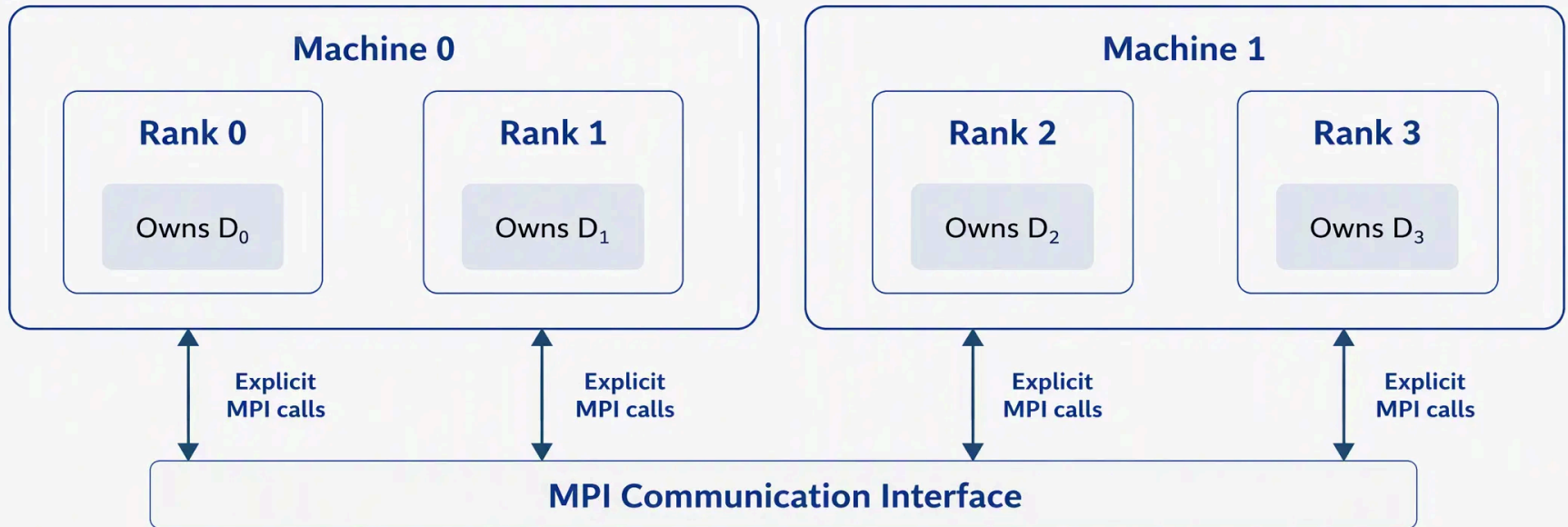
Process



- **Shared address space** — threads can access the same variables
- **Local execution state** — each thread has its own stack
- **Synchronization** — shared updates must be coordinated

Threads share memory; coordination is the main programming challenge.

Why MPI Processes?



Scale Beyond One Machine

Processes can run across multiple hosts



Explicit Data Ownership

Each rank owns its local data



Standard Communication Abstractions

Send / Recv / Broadcast / Reduce

Communication is explicit, but MPI hides low-level network details.

MPI combines private process memory with portable communication across one or many machines.

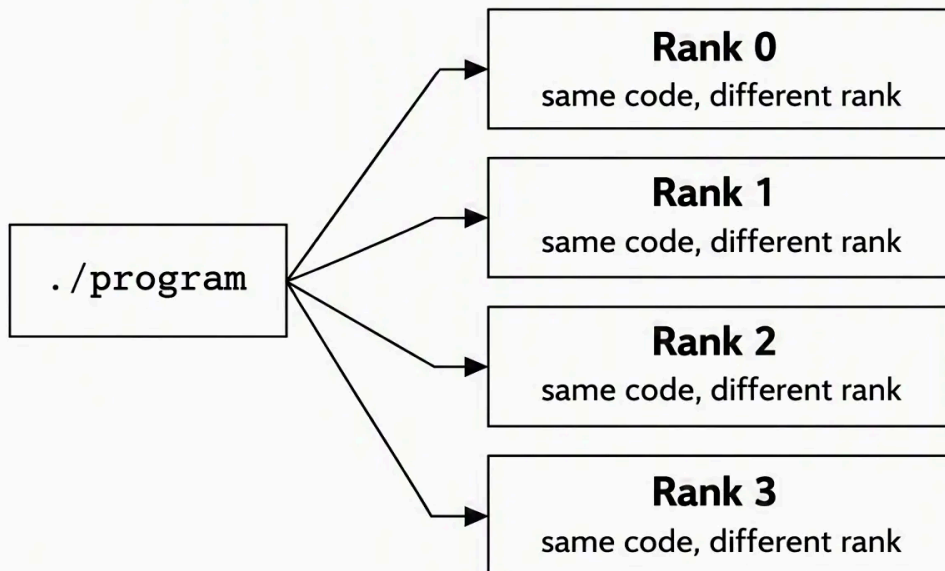
One Program, Many MPI Processes

```
mpiexec -n <number of processes> <executable>
```

```
mpiexec -n 4 ./program
```

launch 4 processes

one executable



Process: one independent MPI worker

Rank: process identifier inside a communicator

Size: number of processes in that communicator

`MPI_COMM_WORLD` = default communicator

SPMD: every process runs the same program, but uses its rank to choose local work.

MPI Program Lifecycle

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

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

```
/* local computation */  
/* communication */
```

```
MPI_Finalize();
```

- ▶ MPI calls occur after MPI_Init.
- ▶ Every process executes the same program.



Demo 1: Hello MPI

Compile

```
$ mpicc hello.c -o hello
```

Run

```
$ mpiexec -n 1 ./hello
```

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

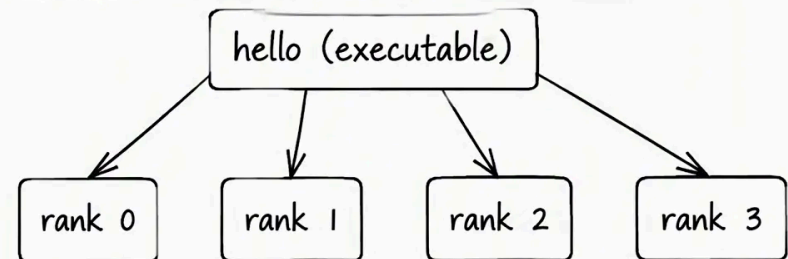
Hello from rank 0 of 4

Hello from rank 1 of 4

Hello from rank 2 of 4

Hello from rank 3 of 4

↑ output order is not guaranteed



Compiled executables?

One

Processes with -n 4?

Four

How identify itself?

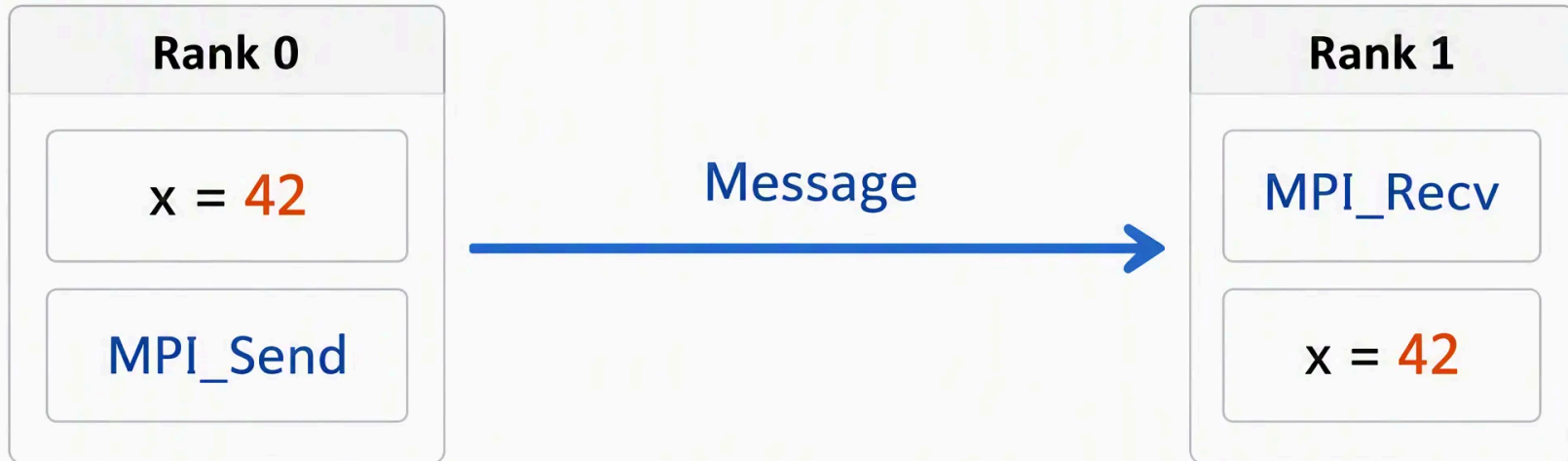
Rank

Output order?

Not guaranteed

Point-to-Point Communication

One sender, one receiver, one matching message



Message matching

- ✓ buffer
- ✓ count
- ✓ datatype
- ✓ source / destination
- ✓ tag
- ✓ communicator

Message = Data + Endpoints + Tag + Communicator

What must match before Rank 1 can receive the message?

Demo 2: Send / Recv in Code

Core rank-specific branches after MPI initialization

```
if (rank == 0) {  
    int x = 42;  
    MPI_Send(&x, 1, MPI_INT, 1, 0, MPI_COMM_WORLD);  
} else if (rank == 1) {  
    int x;  
    MPI_Recv(&x, 1, MPI_INT, 0, 0, MPI_COMM_WORLD,  
            MPI_STATUS_IGNORE);  
}
```

Argument Guide

buffer	Pointer to data
count	Number of elements
datatype	Data type of elements
destination / source	Rank of peer
tag	User-defined identifier
communicator	Communication context

Walkthrough

1. Rank 0 creates x
2. Send and receive descriptions correspond
3. Rank 1 obtains x = 42

Message Matching

Must match before Rank 1 can receive:

- ✓ buffer
- ✓ count
- ✓ datatype
- ✓ destination / source
- ✓ tag
- ✓ communicator

Same message description, different endpoint roles.

Practice: Run and Modify the Programs

SHORT LAB – NOT A FORMAL ASSIGNMENT

✓ Task A — Hello MPI

- ✓ Compile the Hello MPI program.
- ✓ Run with ``-n 1``, ``-n 2``, and ``-n 4``.
- ✓ Record the rank and size printed by each process.
- ✓ Observe whether output order changes.

✓ Task B — Send / Recv

- ✓ Rank 0 sends an integer to Rank 1.
- ✓ Rank 1 prints the received value.
- ✓ Change the value and tag.
- ✓ Run with at least two processes.

★ Optional extension

Rank 1 modifies the value and sends it back to Rank 0.

➔ **Next tutorial:** how multiple local values become a global result using MPI collectives.

Next Class: Interactive MPI Demo

BEFORE NEXT CLASS



COURSE CODE ON GITHUB

<https://github.com/Daoping-Zhang/Experiment.git>

Clone the repository before class.



Before class, browse the README, the Teacher / Worker run modes, and the MiniMPI structure. Run Environment Check / Verification to confirm Python and the local environment are ready.



In the next class, students can join as different ranks and observe Collective Communication in action.



BRING YOUR LAPTOP

Join and participate in the interactive demo.

Clone → Check Environment → Explore MiniMPI → Join the Interactive Demo