CPSC 4240/5240: Parallel Programming Techniques

Lecture 11: Optimistic Locking and MPI

Lecturer: Quanquan C. Liu

Spring 2026

Homework 2 due **Next Thursday**

- Grades for Homework 1 are out
  - 125 for Code
  - Performance Tests will be put up on a (password protected) leaderboard
- Gradescope submission sites are up (Written, Code and Performance Tests)
  - We will test larger tests **offline!**
  - Gradescope memory is too small for these larger tests

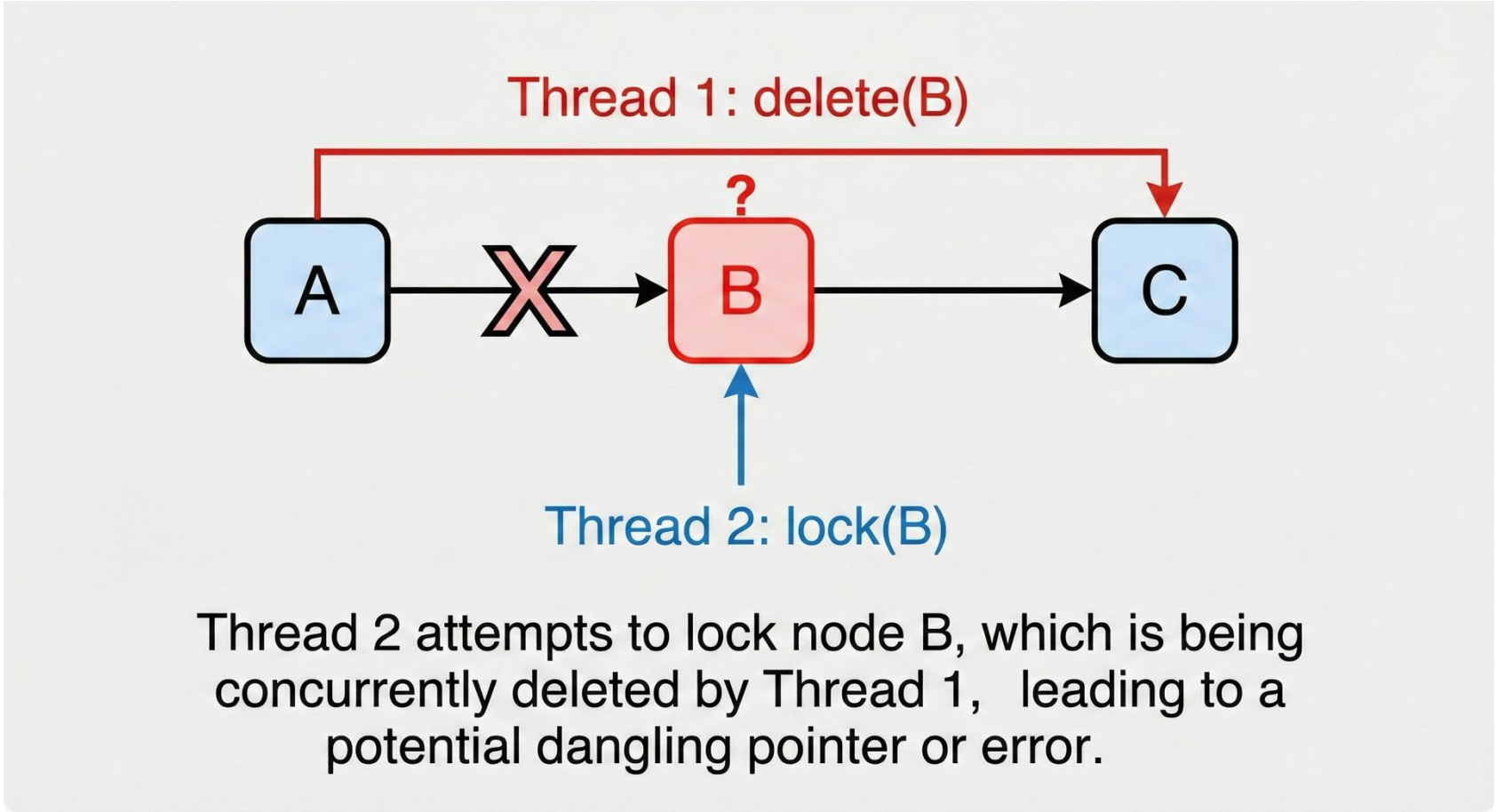
The "Unbounded Memory" Problem

The Flaw in Basic Optimistic Locking: Require all nodes to be kept in memory permanently

The Danger of Immediate Deletion: If we immediately free a node's memory upon deletion, we risk creating dangling pointers

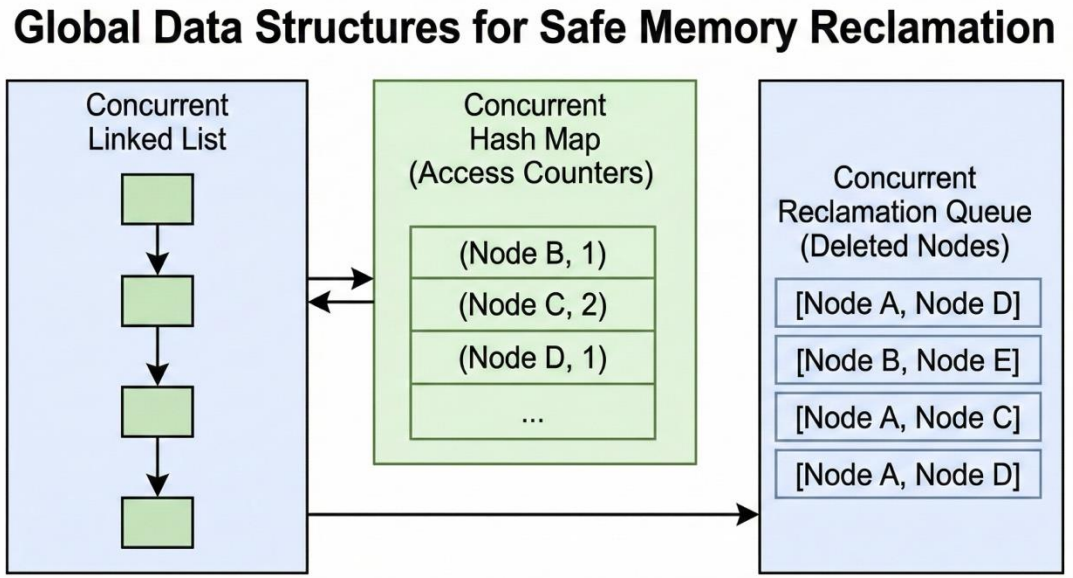
Our Objective: Design a mechanism for safe memory reclamation where deleted nodes are freed

The "Unbounded Memory" Problem

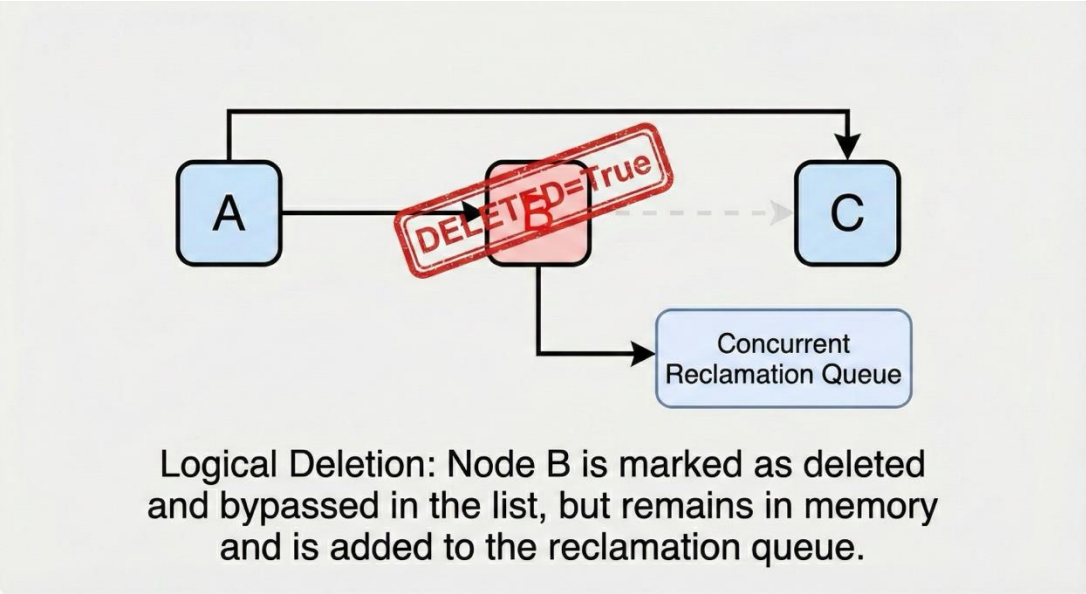


A New Architecture for Safe Deletion

- **The Core Concept:** If an element is actively being accessed by *any* thread, it **cannot be safely deleted** from memory.
- **Tracking Access:** We introduce a secondary **Concurrent Hash Map** that associates each node with an atomic integer
- **The Waiting Room:** We introduce a **Concurrent Reclamation Queue**



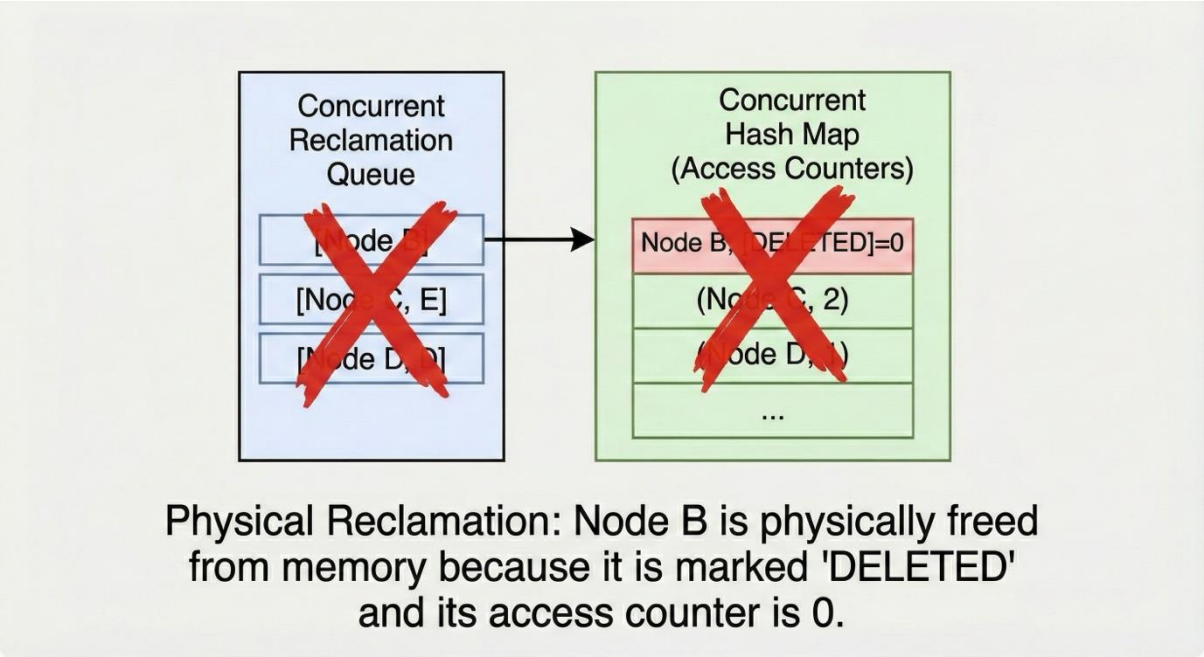
Deletion Phase 1 - Logical Deletion



Logical Deletion: Node B is marked as deleted and bypassed in the list, but remains in memory and is added to the reclamation queue.

- **Step 1:** The deleting thread finds the target node and its predecessor, and acquires locks on both.
- **Step 2:** Instead of freeing the memory, the thread sets a special flag (v.deleted = True) to indicate the node is no longer part of the active list.
- **Step 3:** The node's pointers are bypassed, and the node is pushed into the **Reclamation Queue**. The node is now *logically* deleted, but physically safe.

Deletion Phase 2 - Physical Reclamation



Physical Reclamation: Node B is physically freed from memory because it is marked 'DELETED' and its access counter is 0.

- **The Cleanup Process:** Threads periodically iterate through the Reclamation Queue to check for nodes that can be permanently destroyed.
- **The Safety Check:** A node is only physically freed from memory if its entry in the Hash Map is marked as DELETED and its access counter has reached exactly zero.
- **The Guarantee:** Because threads must increment the counter *before* accessing a node, a counter of zero guarantees no thread holds or can acquire a valid reference, ensuring completely safe memory reclamation.

Message Passing Interface (MPI)

- Standardized and portable **message-passing library** interface designed for parallel computing

Message Passing Interface (MPI)

- Standardized and portable **message-passing library** interface designed for parallel computing
- It is a **standard** rather than a single implementation—multiple libraries (e.g., Open MPI, MPICH, Intel MPI, MVAPICH) adhere to the MPI standard

Message Passing Interface (MPI)

- Standardized and portable **message-passing library** interface designed for parallel computing
- It is a **standard** rather than a single implementation—multiple libraries (e.g., Open MPI, MPICH, Intel MPI, MVAPICH) adhere to the MPI standard
- Designed primarily for **distributed-memory** systems but can also run on shared-memory machines

Message Passing Interface (MPI)

- Standardized and portable **message-passing library** interface designed for parallel computing
- It is a **standard** rather than a single implementation—multiple libraries (e.g., Open MPI, MPICH, Intel MPI, MVAPICH) adhere to the MPI standard
- Designed primarily for **distributed-memory** systems but can also run on shared-memory machines
- MPI focuses on **process-level parallelism**, where each process has its own memory space, and processes communicate by sending/receiving messages

Message Passing Interface (MPI)

MPI is a **message-passing standard for communication across processes:**

- **Synchronization**
- **Data movement between address spaces**

Why Use MPI?

- **Scalability**: MPI is designed to scale to thousands or even millions of processes

Why Use MPI?

- **Scalability**: MPI is designed to scale to thousands or even millions of processes
- **Performance**: Implementations are highly optimized for various network interconnects

Why Use MPI?

- **Scalability**: MPI is designed to scale to thousands or even millions of processes
- **Performance**: Implementations are highly optimized for various network interconnects
- **Flexibility**: Programs can run on different architectures with minimal changes

Why Use MPI?

- **Scalability**: MPI is designed to scale to thousands or even millions of processes
- **Performance**: Implementations are highly optimized for various network interconnects
- **Flexibility**: Programs can run on different architectures with minimal changes
- **Portability**: Adherence to a standard guarantees that MPI programs can be compiled and run on any system with an MPI implementation

Why Use MPI?

- **Scalability**: MPI is designed to scale to thousands or even millions of processes
- **Performance**: Implementations are highly optimized for various network interconnects
- **Flexibility**: Programs can run on different architectures with minimal changes
- **Portability**: Adherence to a standard guarantees that MPI programs can be compiled and run on any system with an MPI implementation
- **Rich Communication Model**: Point-to-point, collective, one-sided communication (in newer versions), and more

Distributed-Memory vs. Shared-Memory

- **Distributed-Memory (MPI)**
  - Each process has its *own* private address space.

Distributed-Memory vs. Shared-Memory

- **Distributed-Memory (MPI)**
  - Each process has its *own* private address space.
  - Processes communicate by explicitly sending and receiving messages.

Distributed-Memory vs. Shared-Memory

- **Distributed-Memory (MPI)**
  - Each process has its *own* private address space.
  - Processes communicate by explicitly sending and receiving messages.
  - Ideal for clusters or supercomputers where you have multiple nodes interconnected by a network.

Distributed-Memory vs. Shared-Memory

- **Distributed-Memory (MPI)**
  - Each process has its *own* private address space.
  - Processes communicate by explicitly sending and receiving messages.
  - Ideal for clusters or supercomputers where you have multiple nodes interconnected by a network.
  - Scales very well to large numbers of nodes (thousands or even millions of processes).

Distributed-Memory vs. Shared-Memory

- **Distributed-Memory (MPI)**
  - Each process has its *own* private address space.
  - Processes communicate by explicitly sending and receiving messages.
  - Ideal for clusters or supercomputers where you have multiple nodes interconnected by a network.
  - Scales very well to large numbers of nodes (thousands or even millions of processes).
- **Shared-Memory (OpenMP)**
  - Threads share a *single* address space.

Distributed-Memory vs. Shared-Memory

- **Distributed-Memory (MPI)**
  - Each process has its *own* private address space.
  - Processes communicate by explicitly sending and receiving messages.
  - Ideal for clusters or supercomputers where you have multiple nodes interconnected by a network.
  - Scales very well to large numbers of nodes (thousands or even millions of processes).
- **Shared-Memory (OpenMP)**
  - Threads share a *single* address space.
  - Communication happens implicitly by reading and writing shared variables.

Distributed-Memory vs. Shared-Memory

- **Distributed-Memory (MPI)**
  - Each process has its *own* private address space.
  - Processes communicate by explicitly sending and receiving messages.
  - Ideal for clusters or supercomputers where you have multiple nodes interconnected by a network.
  - Scales very well to large numbers of nodes (thousands or even millions of processes).
- **Shared-Memory (OpenMP)**
  - Threads share a *single* address space.
  - Communication happens implicitly by reading and writing shared variables.
  - Primarily designed for multi-core or many-core *within* a single node (or a shared-memory machine).

MPI Basics

- **Communicator**
  - A group of processes that can communicate with each other

MPI Basics

- **Communicator**
  - A group of processes that can communicate with each other
  - The most common communicator is **MPI_COMM_WORLD**, which includes all MPI processes by default

MPI Basics

- **Communicator**
  - A group of processes that can communicate with each other
  - The most common communicator is **MPI_COMM_WORLD**, which includes all MPI processes by default
- **Rank**:
  - Each process within a communicator is assigned a unique integer identifier (**rank**)

MPI Basics

- **Communicator**
  - A group of processes that can communicate with each other
  - The most common communicator is **MPI_COMM_WORLD**, which includes all MPI processes by default
- **Rank**:
  - Each process within a communicator is assigned a unique integer identifier (**rank**)
  - Ranks typically range from 0 to N-1 for N processes

MPI Basics

- **Communicator**
  - A group of processes that can communicate with each other
  - The most common communicator is **MPI_COMM_WORLD**, which includes all MPI processes by default
- **Rank**:
  - Each process within a communicator is assigned a unique integer identifier (**rank**)
  - Ranks typically range from 0 to N-1 for N processes
  - **MPI assigns** each process a rank (0, 1, 2, …) at startup

MPI Basics

- **Initialization and Finalization**:
  - MPI programs start by calling **MPI_Init** (or MPI_Init_thread for multithreading support)

MPI Basics

- **Initialization and Finalization**:
  - MPI programs start by calling **MPI_Init** (or MPI_Init_thread for multithreading support)
  - MPI programs must end with **MPI_Finalize**

MPI Basics

- **Initialization and Finalization**:
  - MPI programs start by calling **MPI_Init** (or MPI_Init_thread for multithreading support)
  - MPI programs must end with **MPI_Finalize**
  - After MPI_Finalize, no further MPI calls should be made

MPI Basics

- **Initialization and Finalization**:
  - MPI programs start by calling **MPI_Init** (or MPI_Init_thread for multithreading support)
  - MPI programs must end with **MPI_Finalize**
  - After MPI_Finalize, no further MPI calls should be made
- **Point-to-Point Communication**:
  - **Send/Receive**: **MPI_Send** / **MPI_Recv** are blocking operations

MPI Basics

- **Initialization and Finalization**:
  - MPI programs start by calling **MPI_Init** (or MPI_Init_thread for multithreading support)
  - MPI programs must end with **MPI_Finalize**
  - After MPI_Finalize, no further MPI calls should be made
- **Point-to-Point Communication**:
  - **Send/Receive**: **MPI_Send** / **MPI_Recv** are blocking operations
  - **Non-blocking**: **MPI_Isend / MPI_Irecv** allow communication to occur in the background while computation proceeds

MPI Basics

- **Collective Communication**:
  - Involves a group of processes in a communicator
  - Examples include:
    - **Broadcast**: MPI_Bcast
    - **Gather** / **Scatter**: MPI_Gather, MPI_Scatter
    - **Reduction**: MPI_Reduce, MPI_Allreduce
    - **All-to-All**: MPI_Alltoall
    - **Barrier**: MPI_Barrier

MPI Simple Setup

- Many MPI programs can be written using six functions:

MPI Simple Setup

- Many MPI programs can be written using six functions:
  - **MPI_Init**: Startup

MPI Simple Setup

- Many MPI programs can be written using six functions:
  - **MPI_Init**: Startup
  - **MPI_Finalize**: End

MPI Simple Setup

- Many MPI programs can be written using six functions:
  - **MPI_Init**: Startup
  - **MPI_Finalize**: End
  - **MPI_Comm_Size**: Number of processes

MPI Simple Setup

- Many MPI programs can be written using six functions:
  - **MPI_Init**: Startup
  - **MPI_Finalize**: End
  - **MPI_Comm_Size**: Number of processes
  - **MPI_Comm_Rank**: Rank of processes

MPI Simple Setup

- Many MPI programs can be written using six functions:
  - **MPI_Init**: Startup
  - **MPI_Finalize**: End
  - **MPI_Comm_Size**: Number of processes
  - **MPI_Comm_Rank**: Rank of processes
  - **MPI_Send**: Sending messages

MPI Simple Setup

- Many MPI programs can be written using six functions:
  - **MPI_Init**: Startup
  - **MPI_Finalize**: End
  - **MPI_Comm_Size**: Number of processes
  - **MPI_Comm_Rank**: Rank of processes
  - **MPI_Send**: Sending messages
  - **MPI_Recv**: Receiving messages

MPI Simple Example: Hello World

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

int main(int argc, char** argv) {
    // 1) Initialize the MPI environment
    MPI_Init(&argc, &argv);

    // 2) Get the number of processes
    int world_size;
    MPI_Comm_size(MPI_COMM_WORLD, &world_size);

    // 3) Get the rank of the process
    int world_rank;
    MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);

    // 4) Print a hello message with iostream
    std::cout << "Hello from rank " << world_rank
            << " of " << world_size << std::endl;

    // 5) Finalize the MPI environment
    MPI_Finalize();
    return 0;
}
```

MPI Simple Example: Hello World

```
mpicxx -o hello_mpi hello_mpi.cpp
```

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

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

int main(int argc, char** argv) {
    // 1) Initialize the MPI environment
    MPI_Init(&argc, &argv);

    // 2) Get the number of processes
    int world_size;
    MPI_Comm_size(MPI_COMM_WORLD, &world_size);

    // 3) Get the rank of the process
    int world_rank;
    MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);

    // 4) Print a hello message with iostream
    std::cout << "Hello from rank " << world_rank
              << " of " << world_size << std::endl;

    // 5) Finalize the MPI environment
    MPI_Finalize();
    return 0;
}
```

MPI Simple Example: Hello World

```
mpicxx -o hello_mpi hello_mpi.cpp
```

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

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

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

int main(int argc, char** argv) {
    // 1) Initialize the MPI environment
    MPI_Init(&argc, &argv);

    // 2) Get the number of processes
    int world_size;
    MPI_Comm_size(MPI_COMM_WORLD, &world_size);

    // 3) Get the rank of the process
    int world_rank;
    MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);

    // 4) Print a hello message with iostream
    std::cout << "Hello from rank " << world_rank
              << " of " << world_size << std::endl;

    // 5) Finalize the MPI environment
    MPI_Finalize();
    return 0;
}
```

MPI Simple Example: Hello World

```
mpicxx -o hello_mpi hello_mpi.cpp
```

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

Number of processes you want to run

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

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

int main(int argc, char** argv) {
    // 1) Initialize the MPI environment
    MPI_Init(&argc, &argv);

    // 2) Get the number of processes
    int world_size;
    MPI_Comm_size(MPI_COMM_WORLD, &world_size);

    // 3) Get the rank of the process
    int world_rank;
    MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);

    // 4) Print a hello message with iostream
    std::cout << "Hello from rank " << world_rank
              << " of " << world_size << std::endl;

    // 5) Finalize the MPI environment
    MPI_Finalize();
    return 0;
}
```

MPI Simple Example: Hello World

```
mpicxx -o hello_mpi hello_mpi.cpp
```

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

Each process **runs the same program with a different rank**

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

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

int main(int argc, char** argv) {
    // 1) Initialize the MPI environment
    MPI_Init(&argc, &argv);

    // 2) Get the number of processes
    int world_size;
    MPI_Comm_size(MPI_COMM_WORLD, &world_size);

    // 3) Get the rank of the process
    int world_rank;
    MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);

    // 4) Print a hello message with iostream
    std::cout << "Hello from rank " << world_rank
              << " of " << world_size << std::endl;

    // 5) Finalize the MPI environment
    MPI_Finalize();
    return 0;
}
```

MPI Simple Example: Hello World

Include the MPI library

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

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

int main(int argc, char** argv) {
    // 1) Initialize the MPI environment
    MPI_Init(&argc, &argv);

    // 2) Get the number of processes
    int world_size;
    MPI_Comm_size(MPI_COMM_WORLD, &world_size);

    // 3) Get the rank of the process
    int world_rank;
    MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);

    // 4) Print a hello message with iostream
    std::cout << "Hello from rank " << world_rank
              << " of " << world_size << std::endl;

    // 5) Finalize the MPI environment
    MPI_Finalize();
    return 0;
}
```

MPI Simple Example: Hello World

Initializes MPI Environment

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

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

int main(int argc, char** argv) {
    // 1) Initialize the MPI environment
    MPI_Init(&argc, &argv);

    // 2) Get the number of processes
    int world_size;
    MPI_Comm_size(MPI_COMM_WORLD, &world_size);

    // 3) Get the rank of the process
    int world_rank;
    MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);

    // 4) Print a hello message with iostream
    std::cout << "Hello from rank " << world_rank
              << " of " << world_size << std::endl;

    // 5) Finalize the MPI environment
    MPI_Finalize();
    return 0;
}
```

MPI Simple Example: Hello World

Initializes MPI Environment:

Must call exactly once at the start

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

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

int main(int argc, char** argv) {
    // 1) Initialize the MPI environment
    MPI_Init(&argc, &argv);

    // 2) Get the number of processes
    int world_size;
    MPI_Comm_size(MPI_COMM_WORLD, &world_size);

    // 3) Get the rank of the process
    int world_rank;
    MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);

    // 4) Print a hello message with iostream
    std::cout << "Hello from rank " << world_rank
              << " of " << world_size << std::endl;

    // 5) Finalize the MPI environment
    MPI_Finalize();
    return 0;
}
```

MPI Simple Example: Hello World

Integer variable storing number of processes in MPI_COMM_WORLD

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

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

int main(int argc, char** argv) {
    // 1) Initialize the MPI environment
    MPI_Init(&argc, &argv);

    // 2) Get the number of processes
     int world_size;
    MPI_Comm_size(MPI_COMM_WORLD, &world_size);

    // 3) Get the rank of the process
    int world_rank;
    MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);

    // 4) Print a hello message with iostream
    std::cout << "Hello from rank " << world_rank
              << " of " << world_size << std::endl;

    // 5) Finalize the MPI environment
    MPI_Finalize();
    return 0;
}
```

MPI Simple Example: Hello World

Retrieves total num of processes in MPI_COMM_WORLD; stores in world_size

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

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

int main(int argc, char** argv) {
    // 1) Initialize the MPI environment
    MPI_Init(&argc, &argv);

    // 2) Get the number of processes
    int world_size;
    MPI_Comm_size(MPI_COMM_WORLD, &world_size);

    // 3) Get the rank of the process
    int world_rank;
    MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);

    // 4) Print a hello message with iostream
    std::cout << "Hello from rank " << world_rank
              << " of " << world_size << std::endl;

    // 5) Finalize the MPI environment
    MPI_Finalize();
    return 0;
}
```

MPI Simple Example: Hello World

Store the rank of the particular process

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

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

int main(int argc, char** argv) {
    // 1) Initialize the MPI environment
    MPI_Init(&argc, &argv);

    // 2) Get the number of processes
    int world_size;
    MPI_Comm_size(MPI_COMM_WORLD, &world_size);

    // 3) Get the rank of the process
     int world_rank;
    MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);

    // 4) Print a hello message with iostream
    std::cout << "Hello from rank " << world_rank
            << " of " << world_size << std::endl;

    // 5) Finalize the MPI environment
    MPI_Finalize();
    return 0;
}
```

MPI Simple Example: Hello World

Retrieves the rank of calling process and store in world_rank

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

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

int main(int argc, char** argv) {
    // 1) Initialize the MPI environment
    MPI_Init(&argc, &argv);

    // 2) Get the number of processes
    int world_size;
    MPI_Comm_size(MPI_COMM_WORLD, &world_size);

    // 3) Get the rank of the process
    int world_rank;
    MPI_Comm_rank(MPI_COMM_WORLD,
&world_rank);

    // 4) Print a hello message with iostream
    std::cout << "Hello from rank " << world_rank
              << " of " << world_size << std::endl;

    // 5) Finalize the MPI environment
    MPI_Finalize();
    return 0;
}
```

MPI Simple Example: Hello World

Retrieves the rank of calling process and store in world_rank

Typically 0 to world_size-1

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

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

int main(int argc, char** argv) {
    // 1) Initialize the MPI environment
    MPI_Init(&argc, &argv);

    // 2) Get the number of processes
    int world_size;
    MPI_Comm_size(MPI_COMM_WORLD, &world_size);

    // 3) Get the rank of the process
    int world_rank;
    MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);

    // 4) Print a hello message with iostream
    std::cout << "Hello from rank " << world_rank
              << " of " << world_size << std::endl;

    // 5) Finalize the MPI environment
    MPI_Finalize();
    return 0;
}
```

MPI Simple Example: Hello World

Print out the message using world_rank and world_size

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

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

int main(int argc, char** argv) {
    // 1) Initialize the MPI environment
    MPI_Init(&argc, &argv);

    // 2) Get the number of processes
    int world_size;
    MPI_Comm_size(MPI_COMM_WORLD, &world_size);

    // 3) Get the rank of the process
    int world_rank;
    MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);

    // 4) Print a hello message with iostream
    std::cout << "Hello from rank " << world_rank
              << " of " << world_size << std::endl;

    // 5) Finalize the MPI environment
    MPI_Finalize();
    return 0;
}
```

MPI Simple Example: Hello World

Cleans up the MPI environment

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

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

int main(int argc, char** argv) {
    // 1) Initialize the MPI environment
    MPI_Init(&argc, &argv);

    // 2) Get the number of processes
    int world_size;
    MPI_Comm_size(MPI_COMM_WORLD, &world_size);

    // 3) Get the rank of the process
    int world_rank;
    MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);

    // 4) Print a hello message with iostream
    std::cout << "Hello from rank " << world_rank
              << " of " << world_size << std::endl;

    // 5) Finalize the MPI environment
    MPI_Finalize();
    return 0;
}
```

MPI Simple Example: Hello World

Cleans up the MPI environment: must be called **exactly once** by every process at exit

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

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

int main(int argc, char** argv) {
    // 1) Initialize the MPI environment
    MPI_Init(&argc, &argv);

    // 2) Get the number of processes
    int world_size;
    MPI_Comm_size(MPI_COMM_WORLD, &world_size);

    // 3) Get the rank of the process
    int world_rank;
    MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);

    // 4) Print a hello message with iostream
    std::cout << "Hello from rank " << world_rank
              << " of " << world_size << std::endl;

    // 5) Finalize the MPI environment
    MPI_Finalize();
    return 0;
}
```

Finding Out About the Environment

- Think of each process as an **entity running a copy of the code**

Finding Out About the Environment

- Think of each process as an **entity running a copy of the code**
- Two important questions for each entity:

Finding Out About the Environment

- Think of each process as an **entity running a copy of the code**
- Two important questions for each entity:
  - How many processes are participating?

Finding Out About the Environment

- Think of each process as an **entity running a copy of the code**
- Two important questions for each entity:
  - How many processes are participating?
  - Which one am I?

Finding Out About the Environment

- Think of each process as an **entity running a copy of the code**
- Two important questions for each entity:
  - How many processes are participating?
  - Which one am I?
- MPI uses the following commands to answer these questions:
  - **MPI_Comm_size** reports number of processes

Finding Out About the Environment

- Think of each process as an **entity running a copy of the code**
- Two important questions for each entity:
  - How many processes are participating?
  - Which one am I?
- MPI uses the following commands to answer these questions:
  - **MPI_Comm_size** reports number of processes
  - **MPI_Comm_rank** reports the rank of the calling entity

MPI is Simple but Painful for Shared-Memory

- In OpenMP, only needed a few #pragmas to make sequential code parallel

MPI is Simple but Painful for Shared-Memory

- In OpenMP, only needed a few #pragmas to make sequential code parallel
  - Easy because hardware takes care of data movement **implicitly**

MPI is Simple but Painful for Shared-Memory

- In OpenMP, only needed a few #pragmas to make sequential code parallel
  - Easy because hardware takes care of data movement **implicitly**
  - Threads get the data they need when they need it automatically

MPI is Simple but Painful for Shared-Memory

- In OpenMP, only needed a few #pragmas to make sequential code parallel
  - Easy because hardware takes care of data movement **implicitly**
  - Threads get the data they need when they need it automatically
- MPI requires **explicit data movement**

MPI is Simple but Painful for Shared-Memory

- In OpenMP, only needed a few #pragmas to make sequential code parallel
  - Easy because hardware takes care of data movement **implicitly**
  - Threads get the data they need when they need it automatically
- MPI requires **explicit data movement**
  - Programmer (you!) must say exactly what data goes where and when

MPI Simple Example: Hello World

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

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

int main(int argc, char** argv) {
    // 1) Initialize the MPI environment
    MPI_Init(&argc, &argv);

    // 2) Get the number of processes
    int world_size;
    MPI_Comm_size(MPI_COMM_WORLD, &world_size);

    // 3) Get the rank of the process
    int world_rank;
    MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);

    // 4) Print a hello message with iostream
    std::cout << "Hello from rank " << world_rank
            << " of " << world_size << std::endl;

    // 5) Finalize the MPI environment
    MPI_Finalize();
    return 0;
}
```

MPI Simple Example: Hello World

In this code, **every process is identical**

No **main thread**

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

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

int main(int argc, char** argv) {
    // 1) Initialize the MPI environment
    MPI_Init(&argc, &argv);

    // 2) Get the number of processes
    int world_size;
    MPI_Comm_size(MPI_COMM_WORLD, &world_size);

    // 3) Get the rank of the process
    int world_rank;
    MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);

    // 4) Print a hello message with iostream
    std::cout << "Hello from rank " << world_rank
              << " of " << world_size << std::endl;

    // 5) Finalize the MPI environment
    MPI_Finalize();
    return 0;
}
```

MPI Simple Example: Hello World

In this code, **every process is identical**

No **main thread**

Also no message passing

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

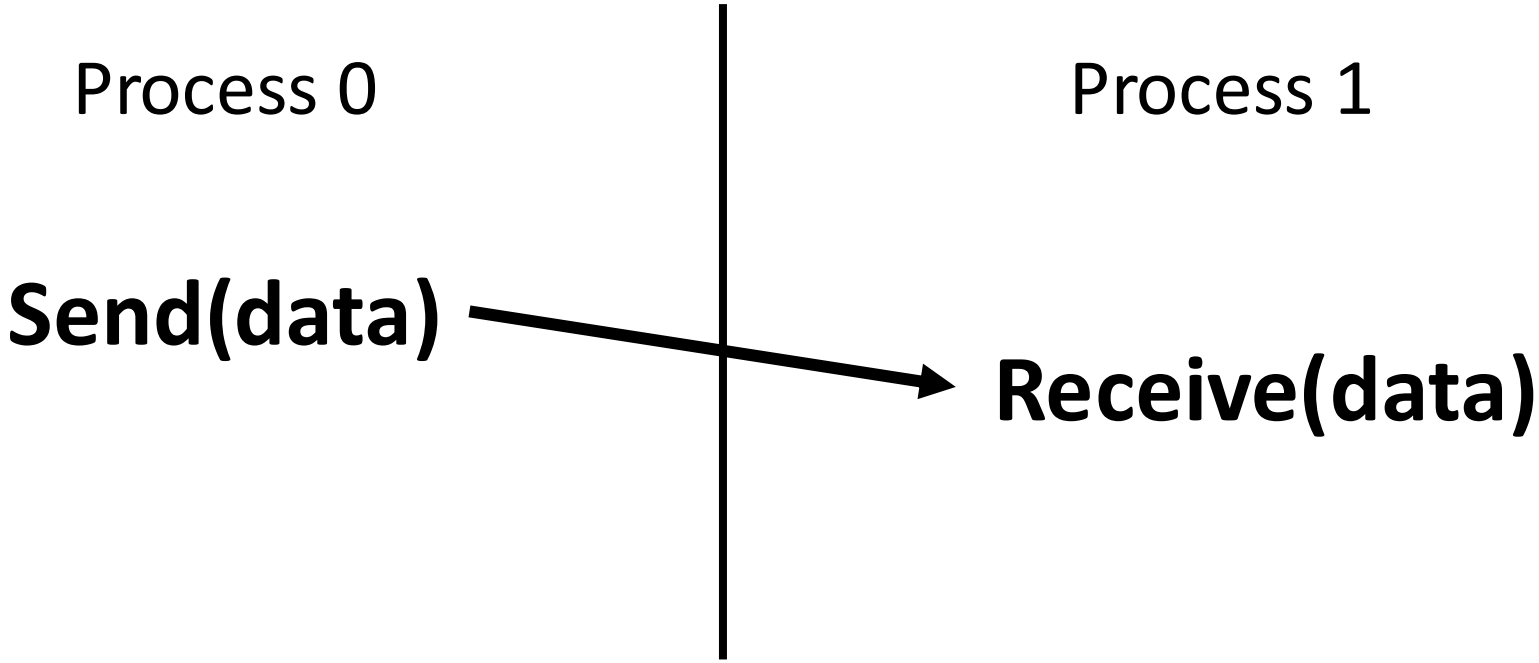
```
#include <mpi.h>
#include <iostream>

int main(int argc, char** argv) {
    // 1) Initialize the MPI environment
    MPI_Init(&argc, &argv);

    // 2) Get the number of processes
    int world_size;
    MPI_Comm_size(MPI_COMM_WORLD, &world_size);

    // 3) Get the rank of the process
    int world_rank;
    MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);

    // 4) Print a hello message with iostream
    std::cout << "Hello from rank " << world_rank
              << " of " << world_size << std::endl;

    // 5) Finalize the MPI environment
    MPI_Finalize();
    return 0;
}
```

MPI Basic Send/Receive

- You! (the programmer) must specify message send and receive instructions

MPI Basic Send/Receive

- You! (the programmer) must specify message send and receive instructions



MPI Basic Send/Receive

- You! (the programmer) must specify message send and receive instructions
- Things that need to be specified:

MPI Basic Send/Receive

- You! (the programmer) must specify message send and receive instructions
- Things that need to be specified:
  - How will “data” be described?

MPI Basic Send/Receive

- You! (the programmer) must specify message send and receive instructions
- Things that need to be specified:
  - How will “data” be described?
  - How will processes be identified?

MPI Basic Send/Receive

- You! (the programmer) must specify message send and receive instructions
- Things that need to be specified:
  - How will “data” be described?
  - How will processes be identified?
  - How will the receiver recognize/screen message?

MPI Basic Send/Receive

- You! (the programmer) must specify message send and receive instructions
- Things that need to be specified:
  - How will “data” be described?
  - How will processes be identified?
  - How will the receiver recognize/screen message?
  - What will it mean for these operations to be complete?

What is Message Passing?

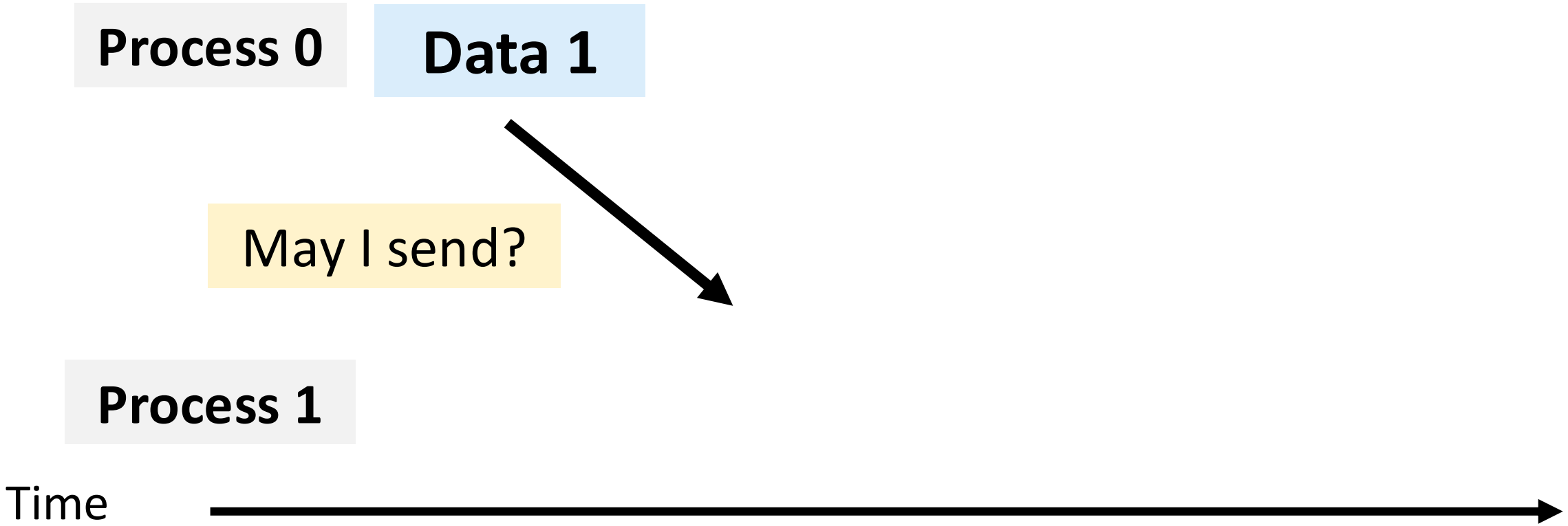
- **Data transfer plus synchronization**

Process 0 **Data 1**

Time 

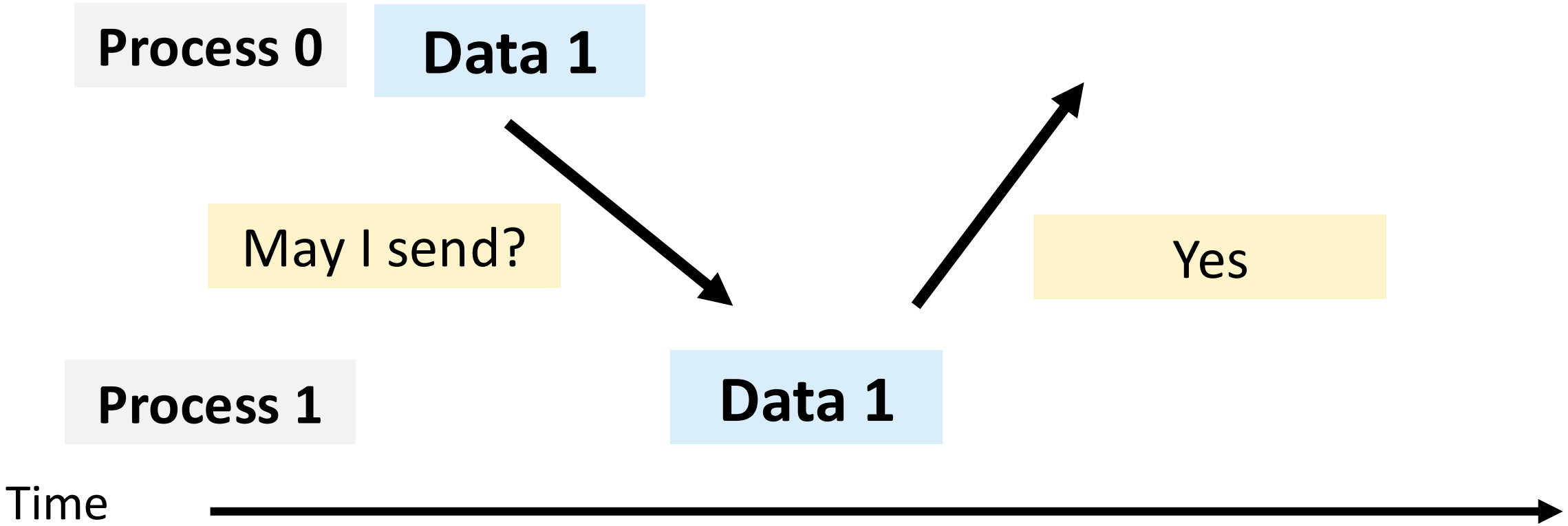
What is Message Passing?

- **Data transfer plus synchronization**



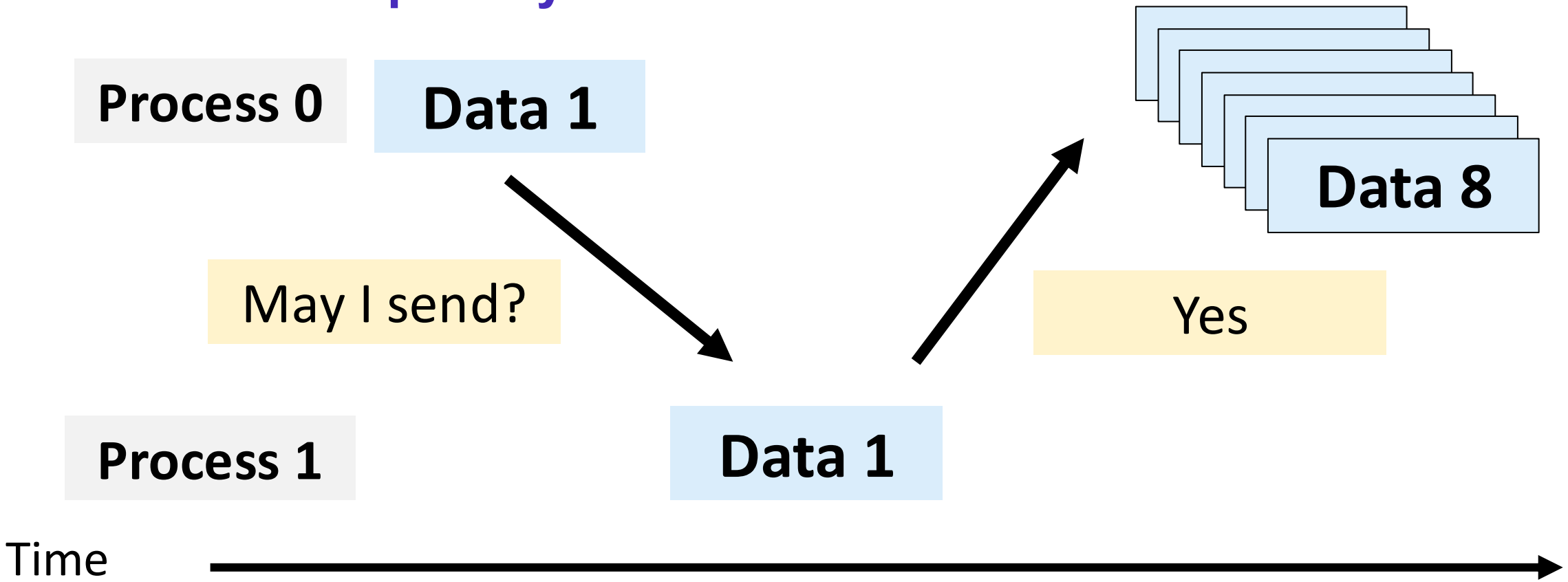
What is Message Passing?

- **Data transfer plus synchronization**



What is Message Passing?

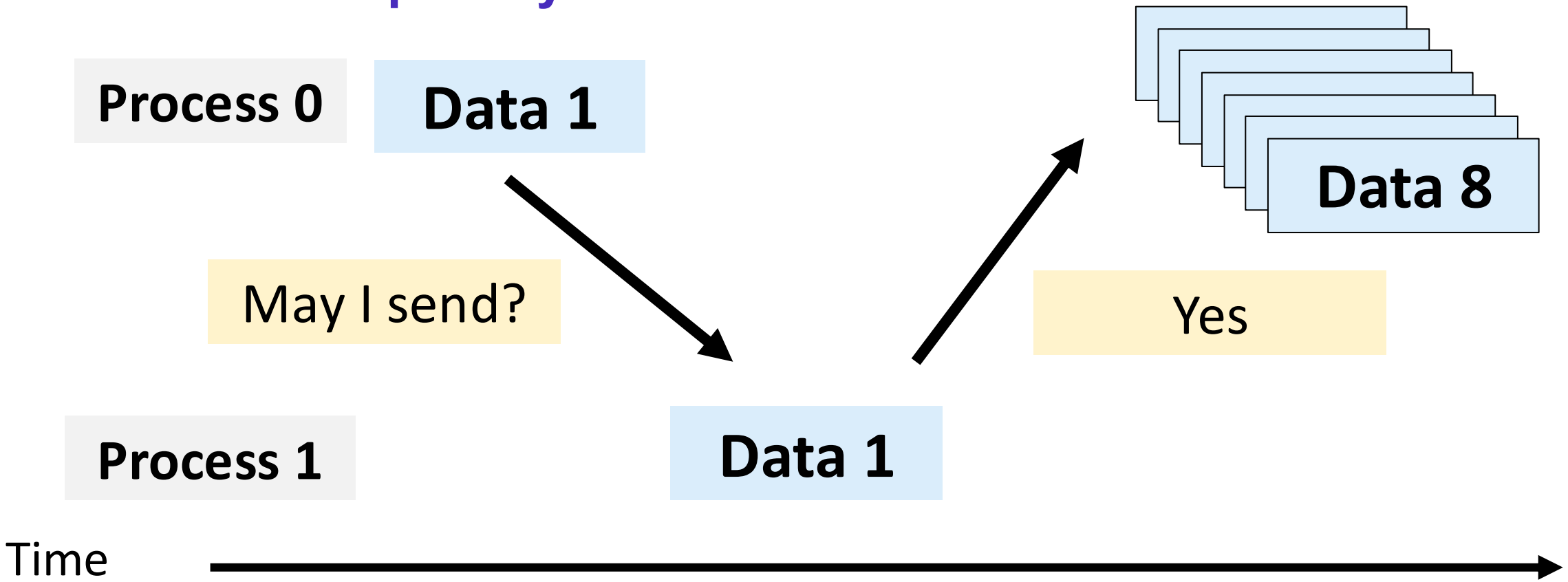
- **Data transfer plus synchronization**



What is Message Passing?

Cooperation between sender and receiver

- **Data transfer plus synchronization**



Message Passing Datatypes

- Messages are described by a triple (address, count, datatype) where:

Message Passing Datatypes

- Messages are described by a triple (address, count, datatype) where:
  - datatype defined as:

Message Passing Datatypes

- Messages are described by a triple (address, count, datatype) where:
  - datatype defined as:
    - Predefined datatype from MPI (e.g., **MPI_INT**, **MPI_DOUBLE**)

Message Passing Datatypes

- Messages are described by a triple (address, count, datatype) where:
  - datatype defined as:
    - Predefined datatype from MPI (e.g., **MPI_INT**, **MPI_DOUBLE**)
    - Contiguous array of MPI datatypes

Message Passing Datatypes

- Messages are described by a triple (address, count, datatype) where:
  - datatype defined as:
    - Predefined datatype from MPI (e.g., **MPI_INT**, **MPI_DOUBLE**)
    - Contiguous array of MPI datatypes
    - An arbitrary structure of datatypes

Message Passing Datatypes

- Messages are described by a triple (address, count, datatype) where:
  - datatype defined as:
    - Predefined datatype from MPI (e.g., **MPI_INT**, **MPI_DOUBLE**)
    - Contiguous array of MPI datatypes
    - An arbitrary structure of datatypes
- MPI functions for constructing custom datatypes, such as array of (int, float) pairs, row of matrix, etc.

MPI Message Tags

- Messages are sent with an accompanying user-defined integer tag, to assist the receiving process in identifying the message

MPI Message Tags

- Messages are sent with an accompanying user-defined integer tag, to assist the receiving process in identifying the message
- Messages can be screened at the receiving end by specifying a specific tag, or not screened by specifying **MPI_ANY_TAG** as the tag in a receive.

MPI Message Tags

- Messages are sent with an accompanying user-defined integer tag, to assist the receiving process in identifying the message
- Messages can be screened at the receiving end by specifying a specific tag, or not screened by specifying **MPI_ANY_TAG** as the tag in a receive.
- Some non-MPI message-passing systems have called tags "message types". MPI calls them tags to avoid confusion with datatypes.

MPI Basic (Blocking) Send

- A **blocking send** does not return until safely reuse or modify the send buffer
  - Program is blocked until MPI has transfer send data
  - Or confirmed that the receiving side has started receiving it

MPI Basic (Blocking) Send

- A **blocking send** does not return until safely reuse or modify the send buffer
  - Program is blocked until MPI has transfer send data
  - Or confirmed that the receiving side has started receiving it

MPI_Send(start, count, datatype, dest, tag, comm)

MPI Basic (Blocking) Send

MPI_Send(start, count, datatype, dest, tag, comm)

- Message buffer described by (start, count, datatype)

MPI Basic (Blocking) Send

MPI_Send(start, count, datatype, dest, tag, comm)

- Message buffer described by (start, count, datatype)
- Target process is specified by dest, rank of the target process in the communicator specified by comm

MPI Basic (Blocking) Send

MPI_Send(start, count, datatype, dest, tag, comm)

- Message buffer described by (start, count, datatype)
- Target process is specified by dest, rank of the target process in the communicator specified by comm
- When this function returns, **data has been delivered**

MPI Basic (Blocking) Send

MPI_Send(start, count, datatype, dest, tag, comm)

- Message buffer described by (start, count, datatype)
- Target process is specified by dest, rank of the target process in the communicator specified by comm
- When this function returns, **data has been delivered**
  - **Not necessarily received**

MPI Basic (Blocking) Send

MPI_Send(start, count, datatype, dest, tag, comm)

- Example:
  - float data[5] = {1.0, 2.0, 3.0, 4.0, 5.0};
  - MPI_Send(data, 5, MPI_FLOAT, 1, 99, MPI_COMM_WORLD);

MPI Basic (Blocking) Send

MPI_Send(start, count, datatype, dest, tag, comm)

- Example:
  - float data[5] = {1.0, 2.0, 3.0, 4.0, 5.0};
  - MPI_Send(**data**, 5, MPI_FLOAT, 1, 99, MPI_COMM_WORLD);

Pointer to start of the array

MPI Basic (Blocking) Send

MPI_Send(start, count, datatype, dest, tag, comm)

- Example:
  - float data[5] = {1.0, 2.0, 3.0, 4.0, 5.0};
  - MPI_Send(**data**, **5**, MPI_FLOAT, 1, 99, MPI_COMM_WORLD);

Number of elements to send

MPI Basic (Blocking) Send

MPI_Send(start, count, datatype, dest, tag, comm)

- Example:
  - float data[5] = {1.0, 2.0, 3.0, 4.0, 5.0};
  - MPI_Send(**data**, **5**, **MPI_FLOAT**, 1, 99, MPI_COMM_WORLD);

Type of each element

MPI Basic (Blocking) Send

MPI_Send(start, count, datatype, dest, tag, comm)

- Example:
  - float data[5] = {1.0, 2.0, 3.0, 4.0, 5.0};
  - MPI_Send(**data**, **5**, **MPI_FLOAT**, **1**, 99, MPI_COMM_WORLD);

Rank of the receiving process

MPI Basic (Blocking) Send

MPI_Send(start, count, datatype, dest, tag, comm)

- Example:
  - float data[5] = {1.0, 2.0, 3.0, 4.0, 5.0};
  - MPI_Send(**data**, **5**, **MPI_FLOAT**, **1**, **99**, MPI_COMM_WORLD);

Message tag used for matching the message on receiver end

MPI Basic (Blocking) Send

MPI_Send(start, count, datatype, dest, tag, comm)

- Example:
  - float data[5] = {1.0, 2.0, 3.0, 4.0, 5.0};
  - MPI_Send(**data**, **5**, **MPI_FLOAT**, **1**, **99**, **MPI_COMM_WORLD**);

Communicator

MPI Basic (Blocking) Receive

- A **blocking receive** does not return until **entire incoming message** has been fully received

MPI Basic (Blocking) Receive

- A **blocking receive** does not return until **entire incoming message** has been fully received

MPI_Recv(start, count, datatype, source, tag, comm, status)

MPI Basic (Blocking) Receive

- A **blocking receive** does not return until **entire incoming message** has been fully received

MPI_Recv(start, count, datatype, source, tag, comm, status)

- Waits until matching source and tag message is received

MPI Basic (Blocking) Receive

- A **blocking receive** does not return until **entire incoming message** has been fully received

MPI_Recv(start, count, datatype, source, tag, comm, status)

- Waits until matching source and tag message is received
- source is rank in communicator specified by comm, or MPI_ANY_SOURCE

MPI Basic (Blocking) Receive

- A **blocking receive** does not return until **entire incoming message** has been fully received

MPI_Recv(start, count, datatype, source, tag, comm, status)

- Waits until matching source and tag message is received
- source is rank in communicator specified by comm, or MPI_ANY_SOURCE
- status contains further information

MPI Basic (Blocking) Receive

- A **blocking receive** does not return until **entire incoming message** has been fully received

MPI_Recv(start, count, datatype, source, tag, comm, status)

- Waits until matching source and tag message is received
- source is rank in communicator specified by comm, or MPI_ANY_SOURCE
- status contains further information
- Receiving fewer than count occurrences of datatype is ok, but receiving more is an error

MPI Basic (Blocking) Receive

MPI_Recv(start, count, datatype, source, tag, comm, status)

- Example:
  - float buffer[5];
  - MPI_Recv(**buffer**, 5, MPI_FLOAT, 0, tag, MPI_COMM_WORLD, MPI_STATUS_IGNORE);

A pointer to the receive buffer where data will be stored

MPI Basic (Blocking) Receive

MPI_Recv(start, count, datatype, source, tag, comm, status)

- Example:
  - float buffer[5];
  - MPI_Recv(**buffer**, **5**, MPI_FLOAT, 0, tag, MPI_COMM_WORLD, MPI_STATUS_IGNORE);

Count of the maximum number of elements you expect to receive

MPI Basic (Blocking) Receive

MPI_Recv(start, count, datatype, source, tag, comm, status)

- Example:
  - float buffer[5];
  - MPI_Recv(**buffer**, **5**, **MPI_FLOAT**, 0, tag, MPI_COMM_WORLD, MPI_STATUS_IGNORE);

Datatype of each element in the buffer; must match sender's datatype

MPI Basic (Blocking) Receive

MPI_Recv(start, count, datatype, source, tag, comm, status)

- Example:
  - float buffer[5];
  - MPI_Recv(**buffer**, **5**, **MPI_FLOAT**, **0**, tag, MPI_COMM_WORLD, MPI_STATUS_IGNORE);

Rank of process sending data; can be MPI_ANY_SOURCE

MPI Basic (Blocking) Receive

MPI_Recv(start, count, datatype, source, tag, comm, status)

- Example:
  - float buffer[5];
  - MPI_Recv(**buffer**, **5**, **MPI_FLOAT**, **0**, **tag**, MPI_COMM_WORLD, MPI_STATUS_IGNORE);

Message identifier; MPI_ANY_TAG

MPI Basic (Blocking) Receive

MPI_Recv(start, count, datatype, source, tag, comm, status)

- Example:
  - float buffer[5];
  - MPI_Recv(**buffer**, **5**, **MPI_FLOAT**, **0**, **tag**, **MPI_COMM_WORLD**, MPI_STATUS_IGNORE);

Communicator

MPI Basic (Blocking) Receive

MPI_Recv(start, count, datatype, source, tag, comm, status)

- Example:
  - float buffer[5];
  - MPI_Recv(**buffer**, **5**, **MPI_FLOAT**, **0**, **tag**, **MPI_COMM_WORLD**, **MPI_STATUS_IGNORE**);

Status object or special constant; MPI_Status gives you other information like num of elements received

Send and Receive Example Program

```
#include <iostream>
#include <cassert>

int main(int argc, char* argv[]) {
   int N = 32;
   double fibs[N + 2];

   // Initialize the first two Fibonacci values
   fibs[0] = 1;
   fibs[1] = 1;

   // Compute and print the sequence
   for (int i = 2; i < N; i++) {
      fibs[i] = fibs[i - 1] + fibs[i - 2];
      std::cout << "The " << i << "th Fibonacci number is "
                << fibs[i] << "." << std::endl;
   }

   return 0;
}
```

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

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

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

   MPI_Status status;
   // Starting message: first two Fibonacci numbers.
   double msg[2] = {1, 1};

   // For processes with rank > 0, receive a message from any
source.
   if (rank > 0) {
      double fibs[2];

MPI_Recv(fibs, 2, MPI_DOUBLE, rank - 1, 0,
MPI_COMM_WORLD, MPI_STATUS_IGNORE);
      // Compute the next Fibonacci number.
      double next = fibs[0] + fibs[1];
      // Update the message: the new pair is (previous second,
next).
      msg[0] = fibs[1];
      msg[1] = next;

      // Output the Fibonacci number corresponding to this
process.
      std::cout << "The " << (rank + 2)
                << "th Fibonacci number is " << next << ".\n";
   }

   // If there is a next process, send the updated message.
   if (rank + 1 < size) {
      int ret = MPI_Send(msg, 2, MPI_DOUBLE, rank + 1, 0,
MPI_COMM_WORLD);
      assert(ret == MPI_SUCCESS);
   }

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

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

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

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

    MPI_Status status;
    // Starting message: first two Fibonacci numbers.
    double msg[2] = {1, 1};

    // For processes with rank > 0, receive a message from any
source.
    if (rank > 0) {
        double fibs[2];

MPI_Recv(fibs, 2, MPI_DOUBLE, rank - 1, 0,
MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        // Compute the next Fibonacci number.
        double next = fibs[0] + fibs[1];
        // Update the message: the new pair is (previous second,
next).
        msg[0] = fibs[1];
        msg[1] = next;

        // Output the Fibonacci number corresponding to this
process.
        std::cout << "The " << (rank + 2)
                  << "th Fibonacci number is " << next << ".\n";
    }

    // If there is a next process, send the updated message.
    if (rank + 1 < size) {
        int ret = MPI_Send(msg, 2, MPI_DOUBLE, rank + 1, 0,
MPI_COMM_WORLD);
        assert(ret == MPI_SUCCESS);
    }

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

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

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

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

   MPI_Status status;
   // Starting message: first two Fibonacci numbers.
   double msg[2] = {1, 1};

   // For processes with rank > 0, receive a message from any
source.
   if (rank > 0) {
      double fibs[2];

MPI_Recv(fibs, 2, MPI_DOUBLE, rank - 1, 0,
      // Compute the next Fibonacci number.
      double next = fibs[0] + fibs[1];
      // Update the message: the new pair is (previous second,
next).
      msg[0] = fibs[1];
      msg[1] = next;

      // Output the Fibonacci number corresponding to this
process.
      std::cout << "The " << (rank + 2)
            << "th Fibonacci number is " << next << ".\n";
   }

   // If there is a next process, send the updated message.
   if (rank + 1 < size) {
      int ret = MPI_Send(msg, 2, MPI_DOUBLE, rank + 1, 0,
MPI_COMM_WORLD);
      assert(ret == MPI_SUCCESS);
   }

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

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

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

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

   MPI_Status status;
   // Starting message: first two Fibonacci numbers.
   double msg[2] = {1, 1};

   // For processes with rank > 0, receive a message from any
source.
   if (rank > 0) {
      double fibs[2];

MPI_Recv(fibs, 2, MPI_DOUBLE, rank - 1, 0,
MPI_COMM_WORLD, MPI_STATUS_IGNORE);
      // Compute the next Fibonacci number.
      double next = fibs[0] + fibs[1];
      // Update the message: the new pair is (previous second,
next).
      msg[0] = fibs[1];
      msg[1] = next;

      // Output the Fibonacci number corresponding to this
process.
      std::cout << "The " << (rank + 2)
                << "th Fibonacci number is " << next << ".\n";
   }

   // If there is a next process, send the updated message.
   if (rank + 1 < size) {
      int ret = MPI_Send(msg, 2, MPI_DOUBLE, rank + 1, 0,
MPI_COMM_WORLD);
      assert(ret == MPI_SUCCESS);
   }

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

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

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

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

   MPI_Status status;
   // Starting message: first two Fibonacci numbers.
   double msg[2] = {1, 1};

   // For processes with rank > 0, receive a message from any
source.
   if (rank > 0) {
      double fibs[2];

MPI_Recv(fibs, 2, MPI_DOUBLE, rank - 1, 0,
MPI_COMM_WORLD, MPI_STATUS_IGNORE);
      // Compute the next Fibonacci number.
      double next = fibs[0] + fibs[1];
      // Update the message: the new pair is (previous second,
next).
      msg[0] = fibs[1];
      msg[1] = next;

      // Output the Fibonacci number corresponding to this
process.
      std::cout << "The " << (rank + 2)
                << "th Fibonacci number is " << next << ".\n";
   }

   // If there is a next process, send the updated message.
   if (rank + 1 < size) {
      int ret = MPI_Send(msg, 2, MPI_DOUBLE, rank + 1, 0,
MPI_COMM_WORLD);
      assert(ret == MPI_SUCCESS);
   }

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

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

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

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

   MPI_Status status;
   // Starting message: first two Fibonacci numbers.
   double msg[2] = {1, 1};

   // For processes with rank > 0, receive a message from any source.
   if (rank > 0) {
      double fibs[2];
```

MPI_Recv(fibs, 2, MPI_DOUBLE, rank - 1, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);

```
      // Compute the next Fibonacci number.
      double next = fibs[0] + fibs[1];
      // Update the message: the new pair is (previous second, next).
      msg[0] = fibs[1];
      msg[1] = next;

      // Output the Fibonacci number corresponding to this process.
      std::cout << "The " << (rank + 2)
                << "th Fibonacci number is " << next << ".\n";
   }

   // If there is a next process, send the updated message.
   if (rank + 1 < size) {
      int ret = MPI_Send(msg, 2, MPI_DOUBLE, rank + 1, 0, MPI_COMM_WORLD);
      assert(ret == MPI_SUCCESS);
   }

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

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

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

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

   MPI_Status status;
   // Starting message: first two Fibonacci numbers.
   double msg[2] = {1, 1};

   // For processes with rank > 0, receive a message from any
source.
   if (rank > 0) {
      double fibs[2];

MPI_Recv(fibs, 2, MPI_DOUBLE, rank - 1, 0,
MPI_COMM_WORLD, MPI_STATUS_IGNORE);
      // Compute the next Fibonacci number.
      double next = fibs[0] + fibs[1];
      // Update the message: the new pair is (previous second,
next).
      msg[0] = fibs[1];
      msg[1] = next;

      // Output the Fibonacci number corresponding to this
process.
      std::cout << "The " << (rank + 2)
                << "th Fibonacci number is " << next << ".\n";
   }

   // If there is a next process, send the updated message.
   if (rank + 1 < size) {
      int ret = MPI_Send(msg, 2, MPI_DOUBLE, rank + 1, 0,
MPI_COMM_WORLD);
      assert(ret == MPI_SUCCESS);
   }

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

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

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

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

   MPI_Status status;
   // Starting message: first two Fibonacci numbers.
   double msg[2] = {1, 1};

   // For processes with rank > 0, receive a message from any
source.
   if (rank > 0) {
      double fibs[2];

MPI_Recv(fibs, 2, MPI_DOUBLE, rank - 1, 0,
MPI_COMM_WORLD, MPI_STATUS_IGNORE);
      // Compute the next Fibonacci number.
      double next = fibs[0] + fibs[1];
      // Update the message: the new pair is (previous second,
next).
         msg[0] = fibs[1];
         msg[1] = next;

      // Output the Fibonacci number corresponding to this
process.
      std::cout << "The " << (rank + 2)
            << "th Fibonacci number is " << next << ".\n";
   }

   // If there is a next process, send the updated message.
   if (rank + 1 < size) {
      int ret = MPI_Send(msg, 2, MPI_DOUBLE, rank + 1, 0,
MPI_COMM_WORLD);
      assert(ret == MPI_SUCCESS);
   }

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

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

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

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

   MPI_Status status;
   // Starting message: first two Fibonacci numbers.
   double msg[2] = {1, 1};

   // For processes with rank > 0, receive a message from any source.
   if (rank > 0) {
      double fibs[2];

MPI_Recv(fibs, 2, MPI_DOUBLE, rank - 1, 0,
MPI_COMM_WORLD, MPI_STATUS_IGNORE);
      // Compute the next Fibonacci number.
      double next = fibs[0] + fibs[1];
      // Update the message: the new pair is (previous second, next).
      msg[0] = fibs[1];
      msg[1] = next;

      // Output the Fibonacci number corresponding to this process.
      std::cout << "The " << (rank + 2)
                << "th Fibonacci number is " << next << ".\n";
   }

   // If there is a next process, send the updated message.
   if (rank + 1 < size) {
      int ret = MPI_Send(msg, 2, MPI_DOUBLE, rank + 1, 0,
MPI_COMM_WORLD);
      assert(ret == MPI_SUCCESS);
   }

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

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

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

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

   MPI_Status status;
   // Starting message: first two Fibonacci numbers.
   double msg[2] = {1, 1};

   // For processes with rank > 0, receive a message from any
source.
   if (rank > 0) {
      double fibs[2];

MPI_Recv(fibs, 2, MPI_DOUBLE, rank - 1, 0,
MPI_COMM_WORLD, MPI_STATUS_IGNORE);
      // Compute the next Fibonacci number.
      double next = fibs[0] + fibs[1];
      // Update the message: the new pair is (previous second,
next).
      msg[0] = fibs[1];
      msg[1] = next;

      // Output the Fibonacci number corresponding to this
process.
      std::cout << "The " << (rank + 2)
                << "th Fibonacci number is " << next << ".\n";
   }

   // If there is a next process, send the updated message.
   if (rank + 1 < size) {
      int ret = MPI_Send(msg, 2, MPI_DOUBLE,
rank + 1, 0, MPI_COMM_WORLD);
      assert(ret == MPI_SUCCESS);
   }

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

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

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

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

   MPI_Status status;
   // Starting message: first two Fibonacci numbers.
   double msg[2] = {1, 1};

   // For processes with rank > 0, receive a message from any
source.
   if (rank > 0) {
      double fibs[2];

MPI_Recv(fibs, 2, MPI_DOUBLE, rank - 1, 0,
MPI_COMM_WORLD, MPI_STATUS_IGNORE);
      // Compute the next Fibonacci number.
      double next = fibs[0] + fibs[1];
      // Update the message: the new pair is (previous second,
next).
      msg[0] = fibs[1];
      msg[1] = next;

      // Output the Fibonacci number corresponding to this
process.
      std::cout << "The " << (rank + 2)
                << "th Fibonacci number is " << next << ".\n";
   }

   // If there is a next process, send the updated message.
   if (rank + 1 < size) {
      int ret = MPI_Send(msg, 2, MPI_DOUBLE, rank + 1, 0,
MPI_COMM_WORLD);
      assert(ret == MPI_SUCCESS);
   }

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

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

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

    int rank, size;
    MPI_Comm_size(MPI_C
    MPI_Comm_rank(MPI_

    MPI_Status status;
    // Starting message: firs
    double msg[2] = {1, 1};

    // For processes with ra
source.
    if (rank > 0) {
        double fibs[2];

MPI_Recv(fibs, 2, MPI_DOUBLE, rank - 1, 0,
MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        // Compute the next Fibonacci number.
        double next = fibs[0] + fibs[1];
        // Update the message: the new pair is (previous second,
next).
        msg[0] = fibs[1];
        msg[1] = next;
```

```
orresponding to this

' << next << ".\n";

e updated message.

OUBLE, rank + 1, 0,
```

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

```
mpirun -np 8 ./fibonacci

The 3th Fibonacci number is 2.
The 4th Fibonacci number is 3.
The 5th Fibonacci number is 5.
The 6th Fibonacci number is 8.
The 7th Fibonacci number is 13.
The 8th Fibonacci number is 21.
The 9th Fibonacci number is 34.
```

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

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

   int rank, size;
   MI
   MI

   MI
   //
   do

   //
sour
   if (


MPI_
MPI_
```

```
      // Compute the next Fibonacci number.
      double next = fibs[0] + fibs[1];
      // Update the message: the new pair is (previous second,
next).
      msg[0] = fibs[1];
      msg[1] = next;

      // Output the Fibonacci number corresponding to this
ess.
      std::cout << "The " << (rank + 2)
                << "th Fibonacci number is " << next << ".\n";


      If there is a next process, send the updated message.
      rank + 1 < size) {
      int ret = MPI_Send(msg, 2, MPI_DOUBLE, rank + 1, 0,
      _COMM_WORLD);
      assert(ret == MPI_SUCCESS);


      Finalize the MPI environment.
      MPI_Finalize();
      turn 0;
}
```

Could result in deadlock in certain situations!

Briefly discuss next class

Lecture 11 Quiz