Distributed Memory Programming with MPI

Costa Rica High Performance Computing School 2025

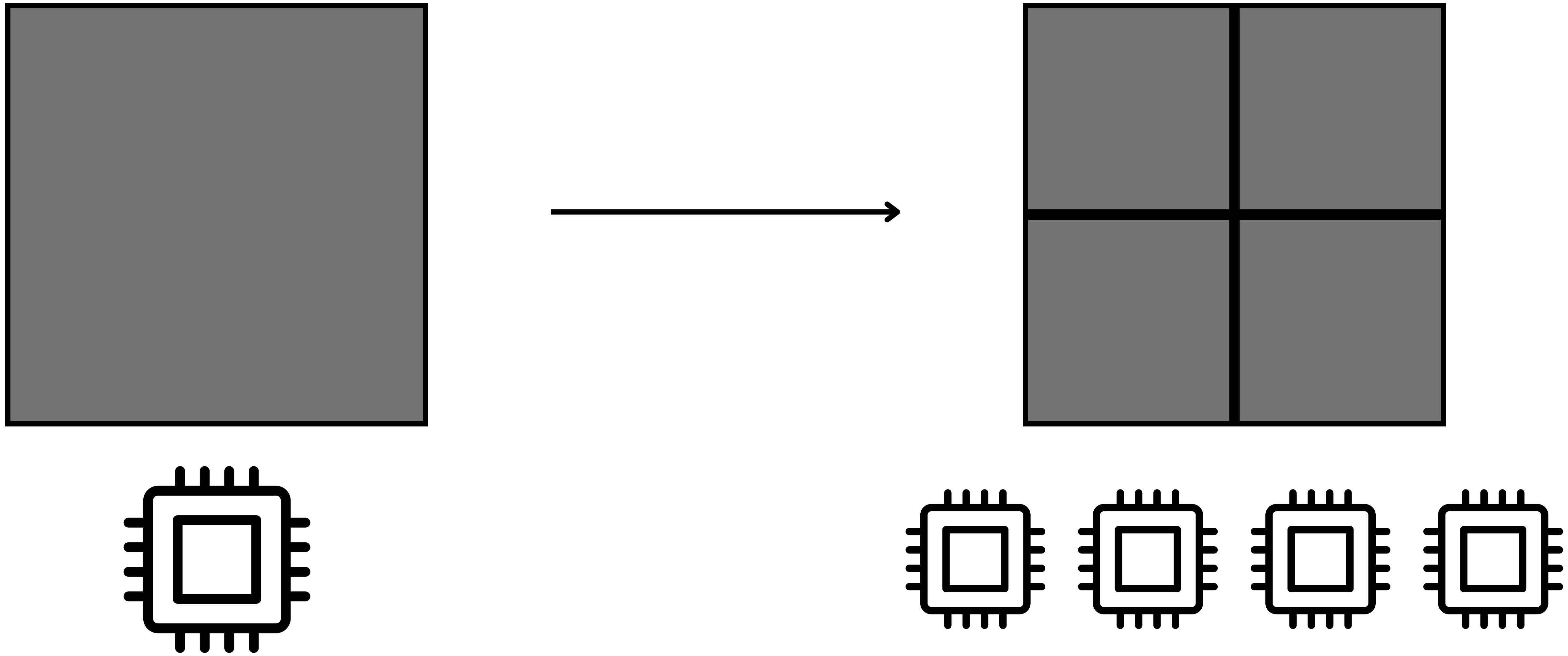
- Introduction to distributed memory programming
- Message passing basics
- Message Passing Interface
- MPI Applications
- Practice

Introduction to Distributed Memory Programming

What is Distributed Memory Programming?

Introduction to Distributed Memory Programming

Distributed Memory Programming refers to a programming paradigm used to write software for systems where each processor has its own local memory for problems that do not fit in one processor.



Why Distributed Memory Programming?

Introduction to Distributed Memory Programming

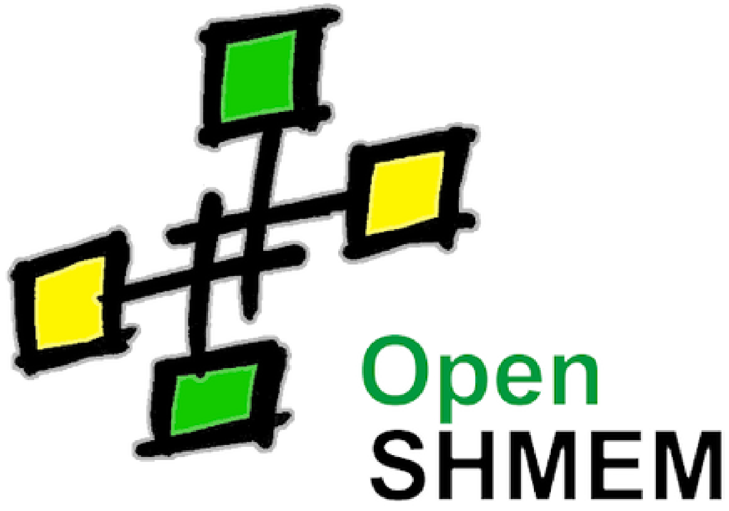
- Simulations either require great resolution or are comprehend physical cases that happen in great extents.

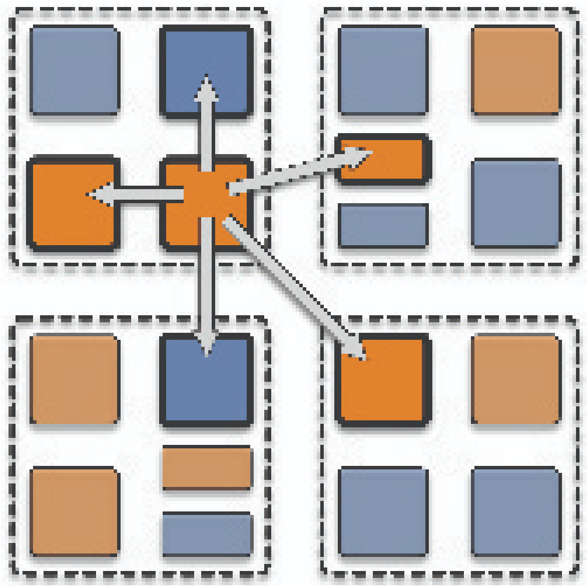
- In Big Data, the size of modern datasets far exceeds the capacity of single machine memory.

- Current Deep Learning models have billions of parameters, and are currently trained on big datasets.

How is Distributed Memory Programming implemented?

Introduction to Distributed Memory Programming

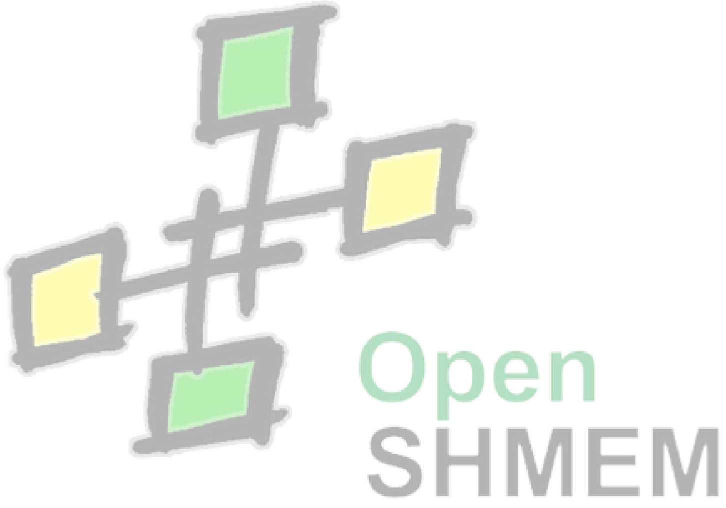


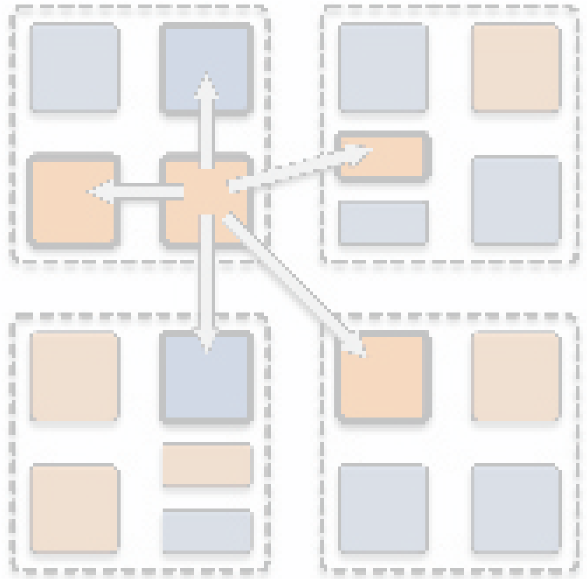


Charm++

How is Distributed Memory Programming implemented?

Introduction to Distributed Memory Programming





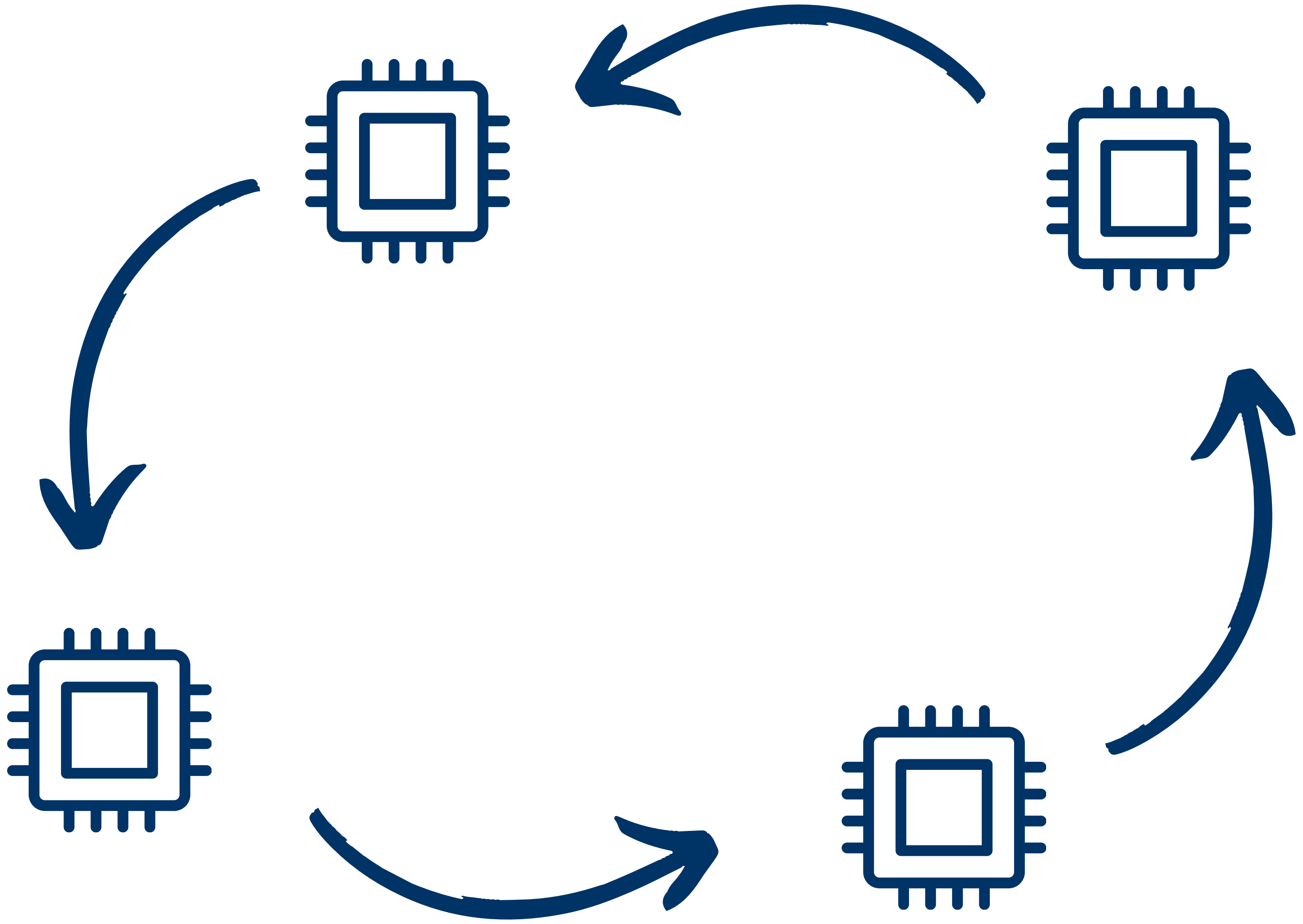
Charm++

CeNAT

What is Message Passing?

Introduction to Distributed Memory Programming

Message Passing is a method of communication used in distributed memory systems where processes running on separate memory spaces exchange data explicitly by sending and receiving messages.



Message Passing Interface - (MPI Standard)

Introduction to Distributed Memory Programming

Message Passing Interface - (MPI Standard)

Introduction to Distributed Memory Programming

Message Passing Basics

Terminology

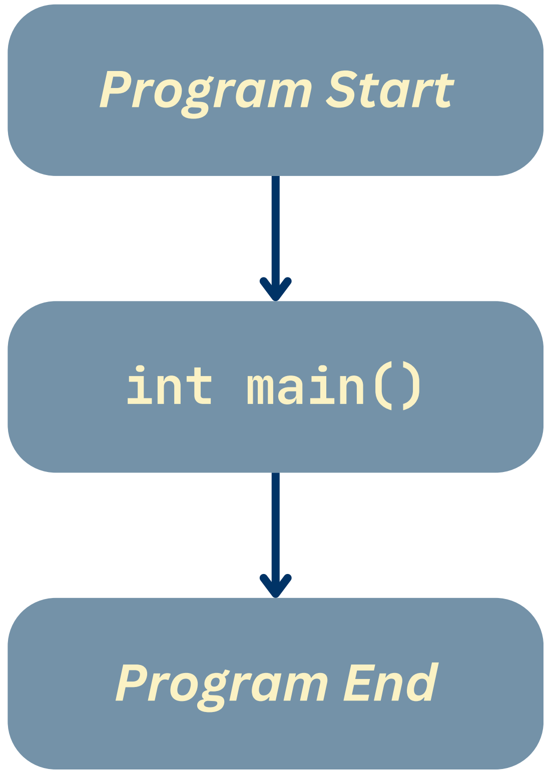
Message Passing Basics

- **MPI**: A standard API for parallel programming in distributed-memory systems.
- **Rank**: A unique ID for a process in a communicator, typically starting at 0.
- **Communicator**: A group of processes that can communicate amongst each other, e.g., MPI_COMM_WORLD.
- **Process**: An independent program instance with its own memory, communicating via MPI.
- **Blocking**: Operations that wait until data is sent/received.
- **Non-blocking**: Operations that start data transfer and return immediately.
- **Group**: A set of processes in a communicator.
- **Topology**: Logical arrangement of processes, e.g., cartesian or graph.

Program Structure

Message Passing Basics

Serial

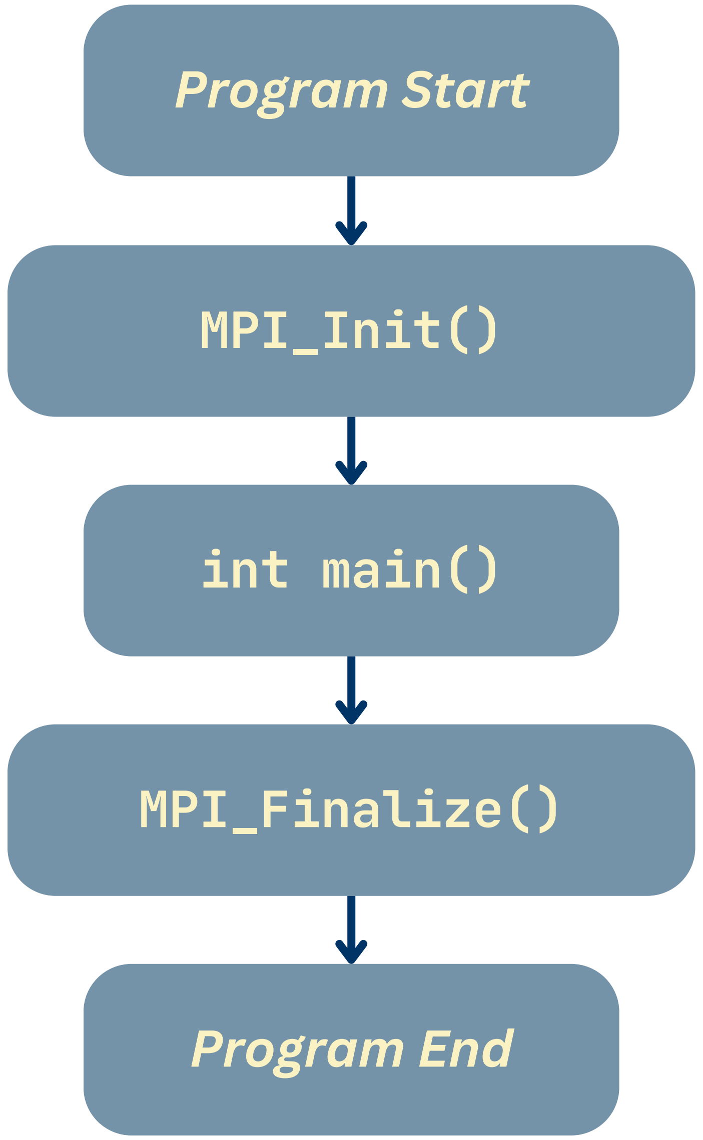


Program Start

`int main()`

Program End

MPI



Program Start

`MPI_Init()`

`int main()`

`MPI_Finalize()`

Program End

Program Structure

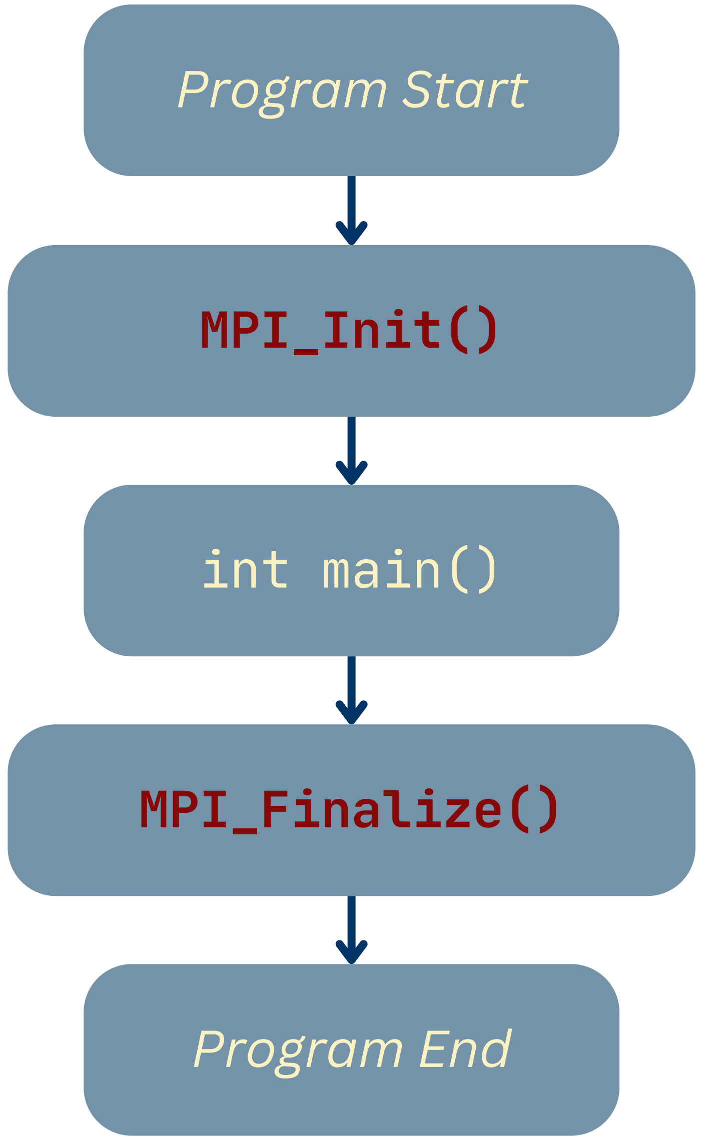
Message Passing Basics

`MPI_Init()`

- Command-line argument processing
- Initialization of internal data structures for MPI management
- Initialization of comms subsystem (TCP/IP, Infiniband, NVLink)
- Initialization of rank mapping
- Initial synchronization of processes

`MPI_Finalize()`

- Final synchronization or processes
- Network connection close
- Resource release
- Initialization of comms subsystem (TCP/IP, Infiniband, NVLink)
- Initialization of rank mapping



Program Structure

Message Passing Basics

helloworld.cpp

```
#include <iostream>

int main(int argc, char*argv[])
{
    std::cout << “Hello World!\n” << std::endl;

    return 1;

}
```

mpi_helloworld.cpp

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

int main(int argc, char*argv[])
{
    int error;
    error = MPI_Init(&argc, &argv); // error-checking

    std::cout << “Hello World!\n” << std::endl;

    error = MPI_Finalize();

    return 1;

}
```

Compilation

Message Passing Basics

helloworld.cpp

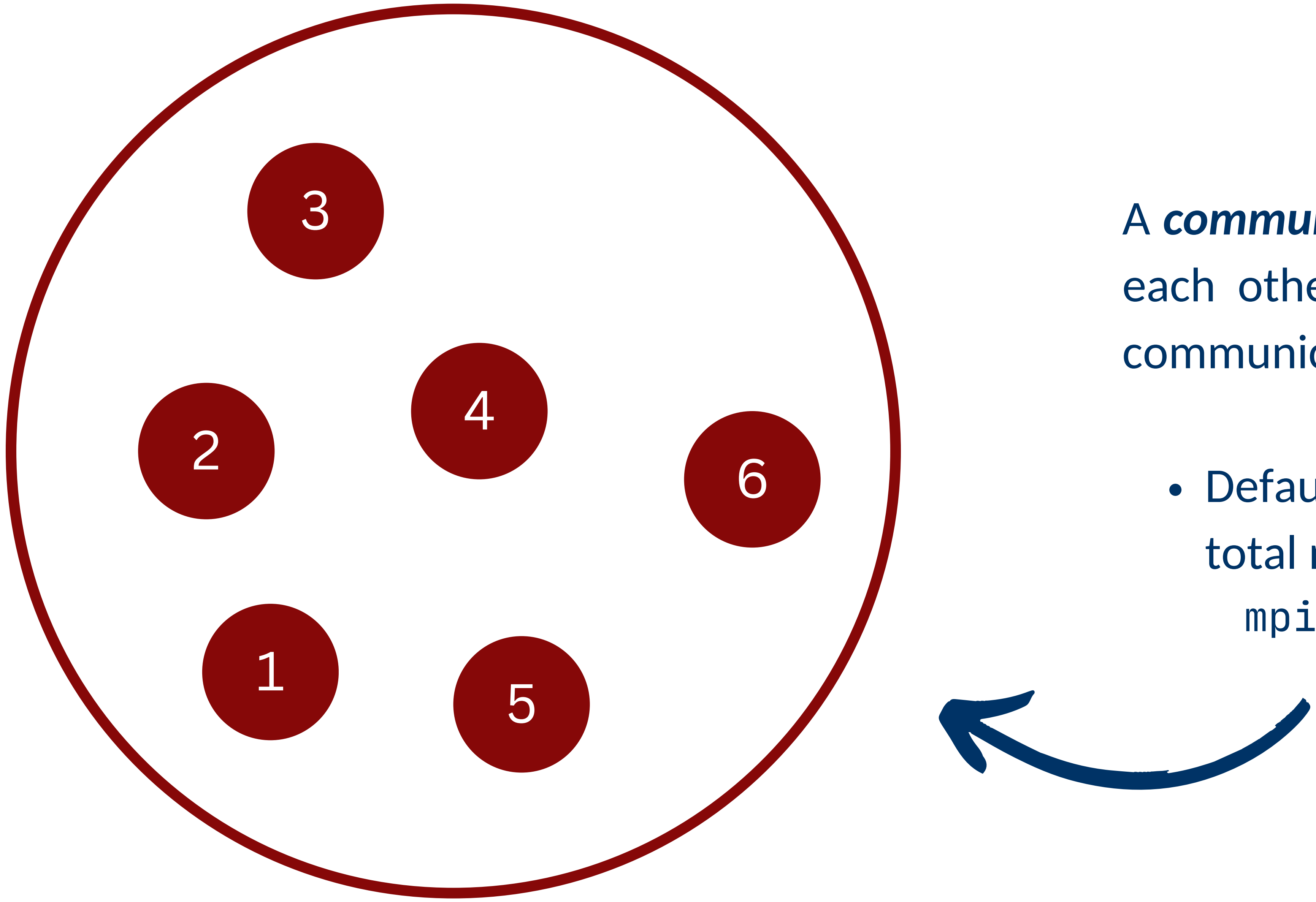
```
#include <iostream>

int main(int argc, char*argv[])
{
    std::cout << “Hello World!\n” << std::endl;

    return 0;

}

$ g++ -o helloworld helloworld.cpp
$ ./helloworld
Hello World!

$
```

mpi_helloworld.cpp

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

int main(int argc, char*argv[])
{
    int err;
    err = MPI_Init(&argc, &argv); // error-checking

    std::cout << “Hello World!\n” << std::endl;

    err = MPI_Finalize();

    return 0;

}


$ mpic++ -o mpi_helloworld mpi_helloworld.cpp
$ mpirun -n 2 ./mpi_helloworld
Hello World!

Hello World!

$
```

MPI Communicators

Message Passing Basics



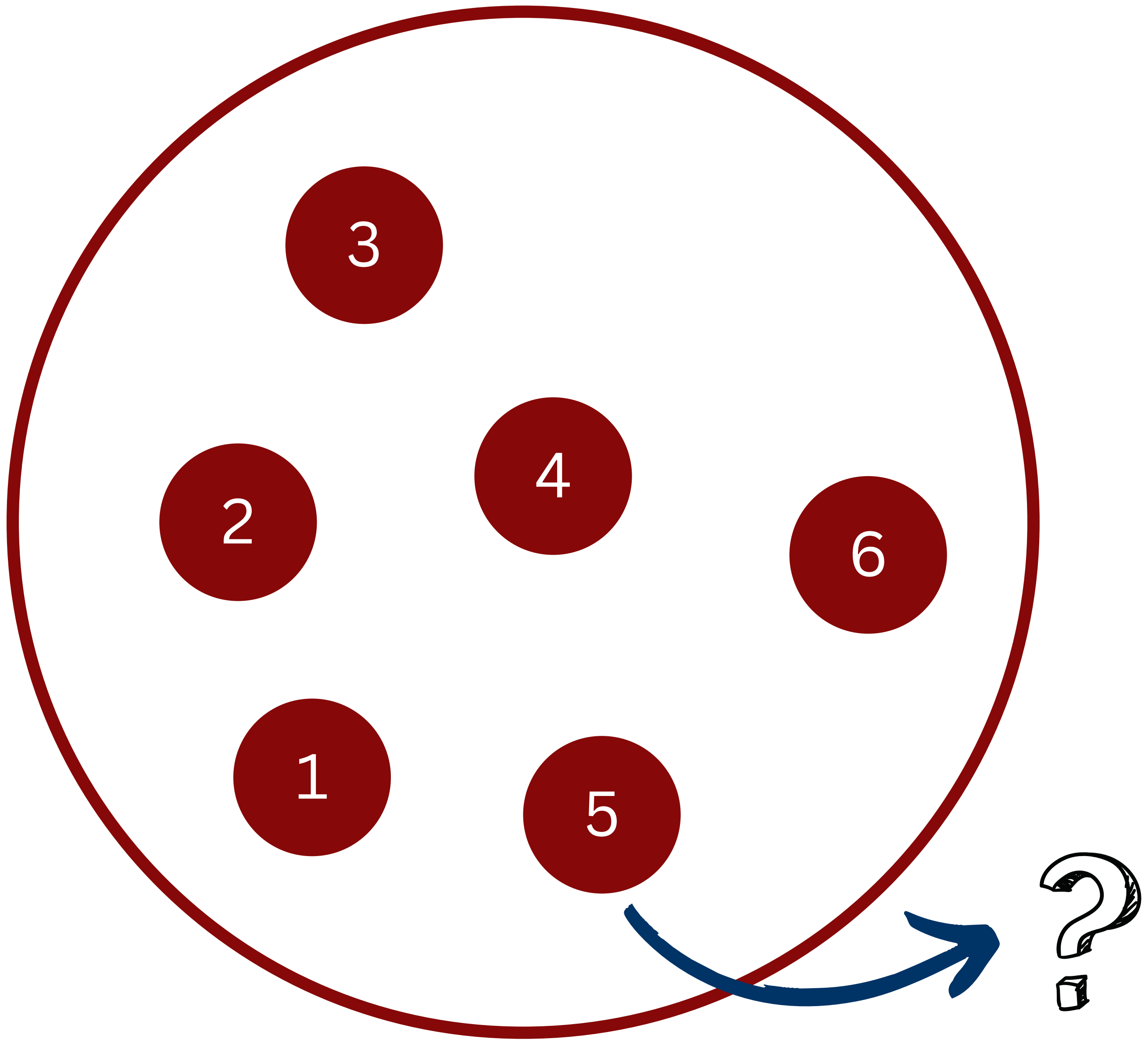
A ***communicator*** defines a group of processes that can communicate with each other. Every process in an MPI program belongs to at least one communicator.

- Default Communicator: **MPI_COMM_WORLD** - it is made up of the total number of processes specified on program call:

```
mpirun -n 6 ./mpi_helloworld
```

Process Identification

Message Passing Basics



Each process has a distinct ID or **"rank"** associated to the communicator, in this sense a process may have different rank on different communicators.

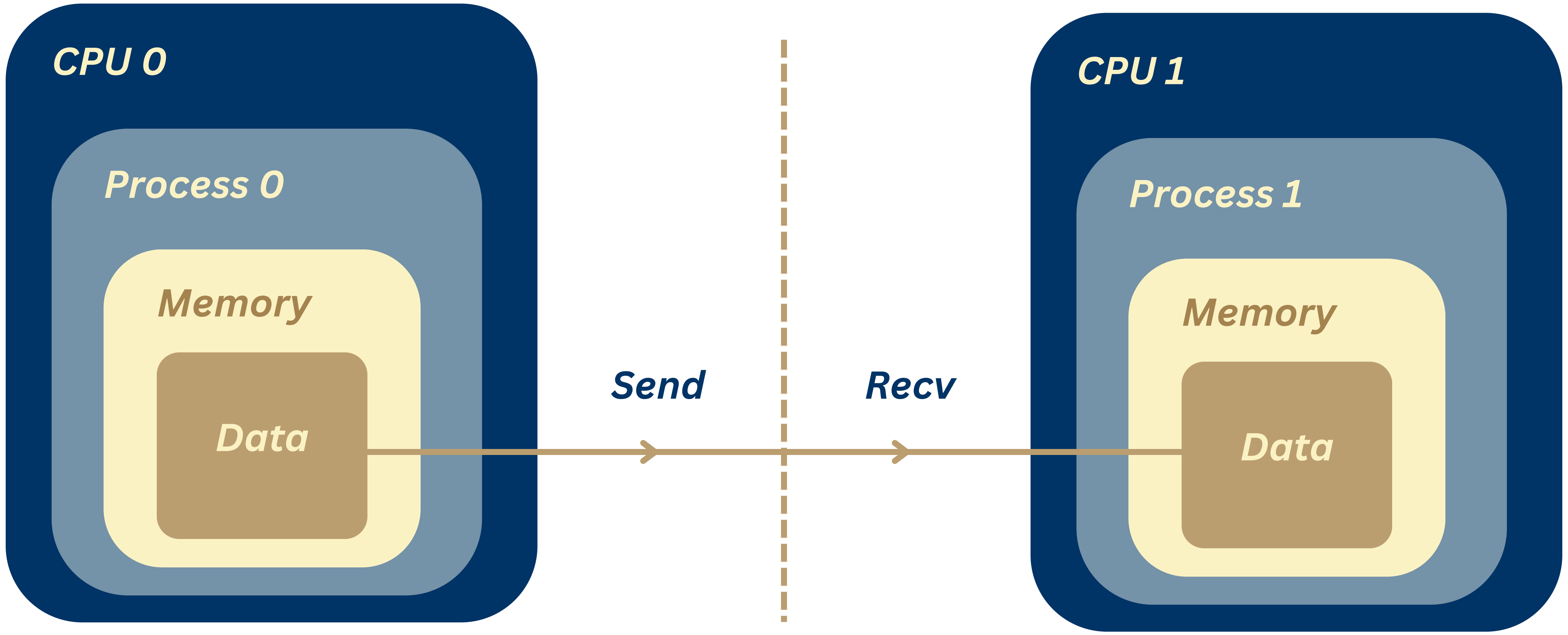
- To obtain the **rank** for each process one uses the **MPI_Comm_rank** function.

```
int MPI_Comm_rank(MPI_Comm comm, int *rank)
```

P2P comms

Message Passing Basics

***Point-to-point communication** refers to direct communication between two processes in a distributed memory system.*



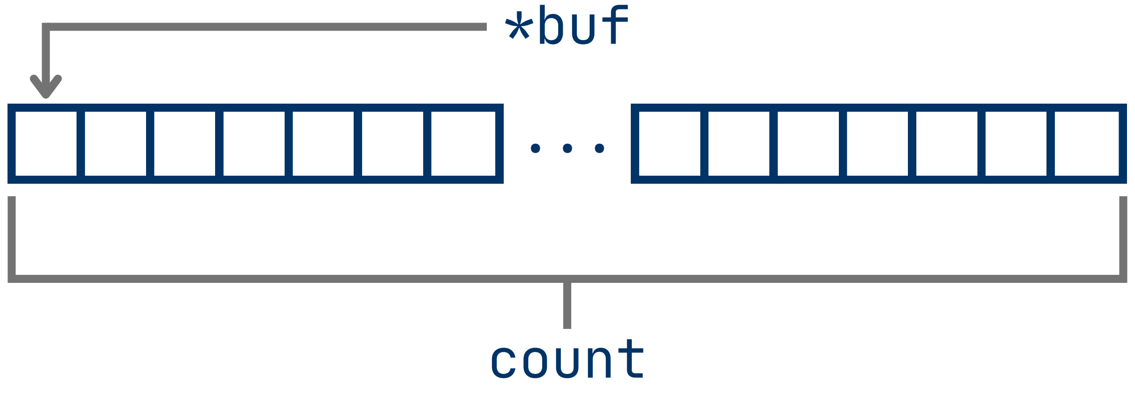
P2P comms

Message Passing Basics

Point-to-point communication *refers to direct communication between two processes in a distributed memory system.*

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

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



`MPI_Sendrecv()`

Message Passing Interface

Collectives (Blocking/synchronous)

Message Passing Interface

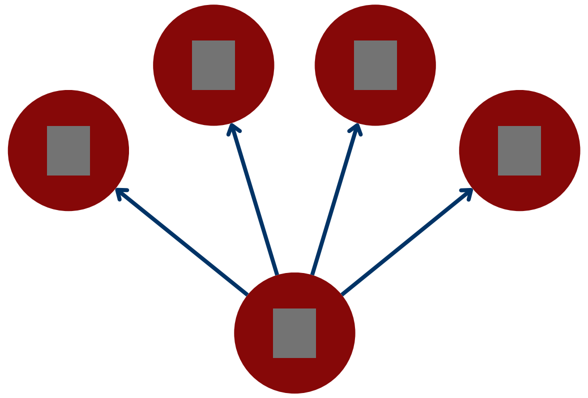
***Collective** communication refers to operations that involve all processes in a communicator working together to exchange data or perform computations.*

- **MPI_Bcast:** Sends data from one process to all other processes
- **MPI_Gather:** Collects data from all processes to a single root process
- **MPI_Scatter:** Distributes distinct chunks of data from the root process to all other processes
- **MPI_Allgather:** Every process gathers data from all other processes
- **MPI_Alltoall:** Every process sends distinct data to all other processes and receives distinct data from all others
- **MPI_Reduce:** Combines data from all processes using an operation (sum, mean, max)
- **MPI_Allreduce:** Combines data from all processes using an operation (sum, mean, max) and distributes the result to all processes
- **MPI_Scan:** Performs a prefix reduction where each process gets the reduction result up to its rank
- **MPI_Barrier:** Synchronizes all processes in the communicator. Each process waits until all have reached the barrier.

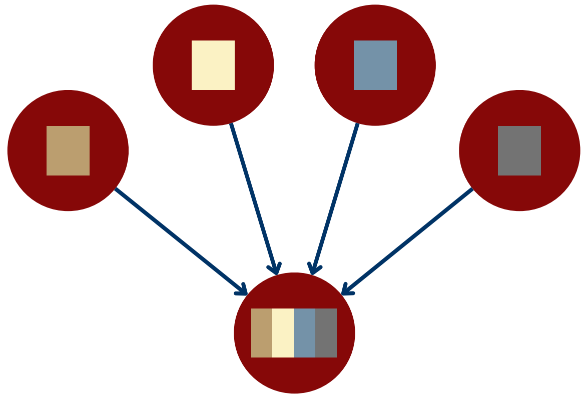
Collectives (Blocking/Synchronous)

Message Passing Interface

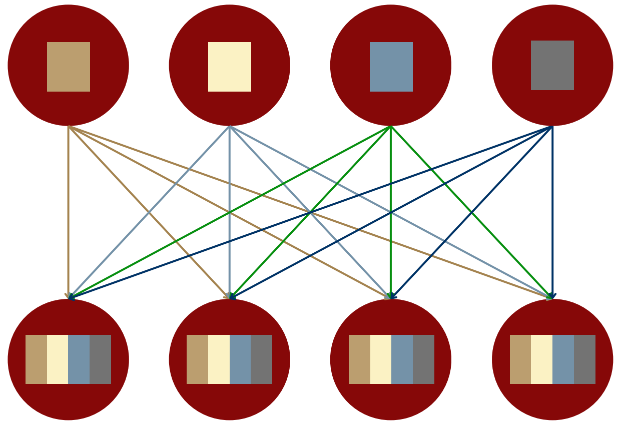
Collective *communication refers to operations that involve all processes in a communicator working together to exchange data or perform computations.*



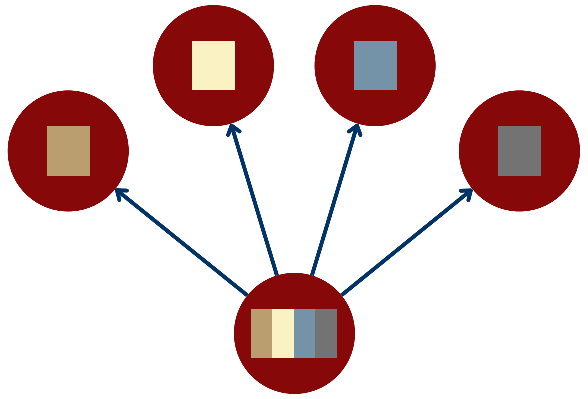
Broadcast



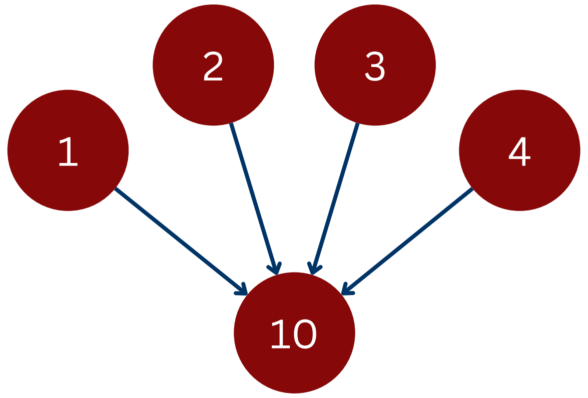
Gather



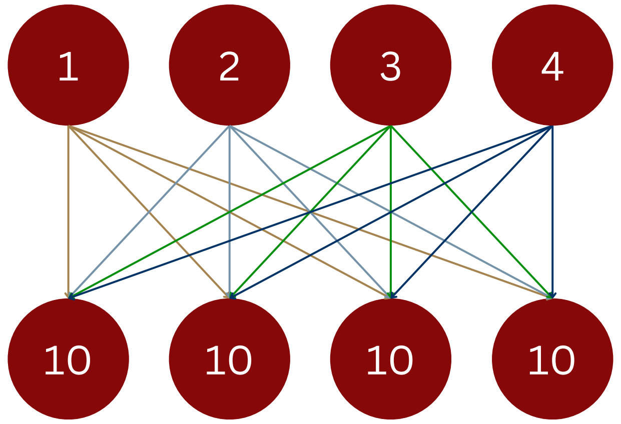
All-Gather



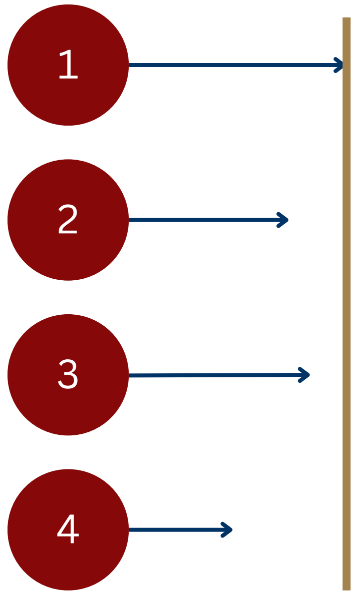
Scatter



Reduction



All-Reduce

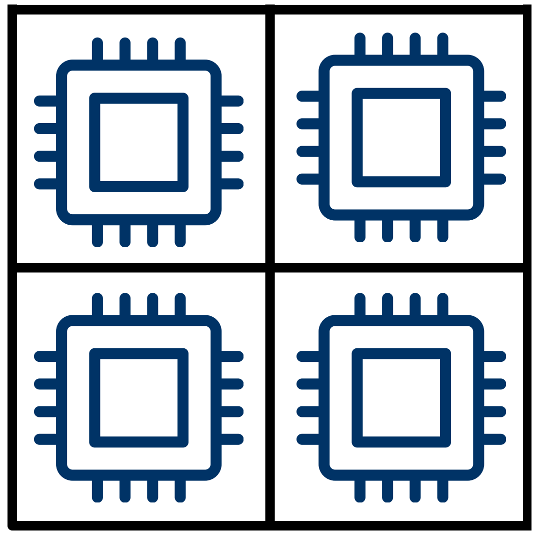


Barrier

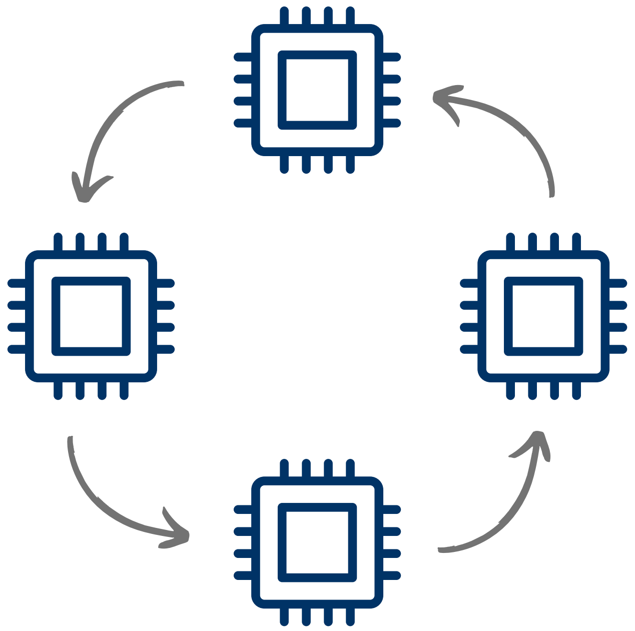
MPI Applications

Communicaiton Patterns

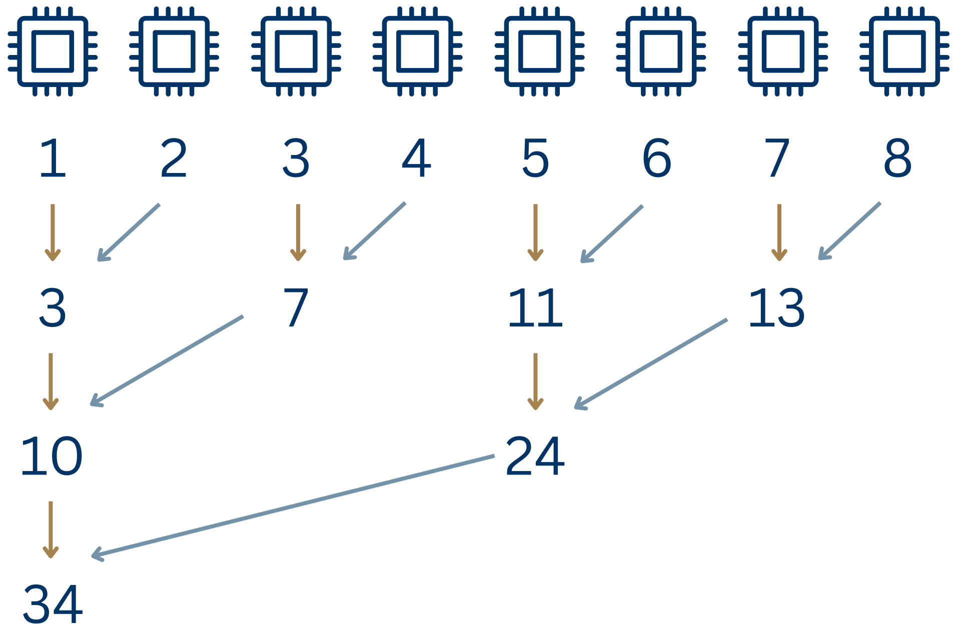
MPI Applications



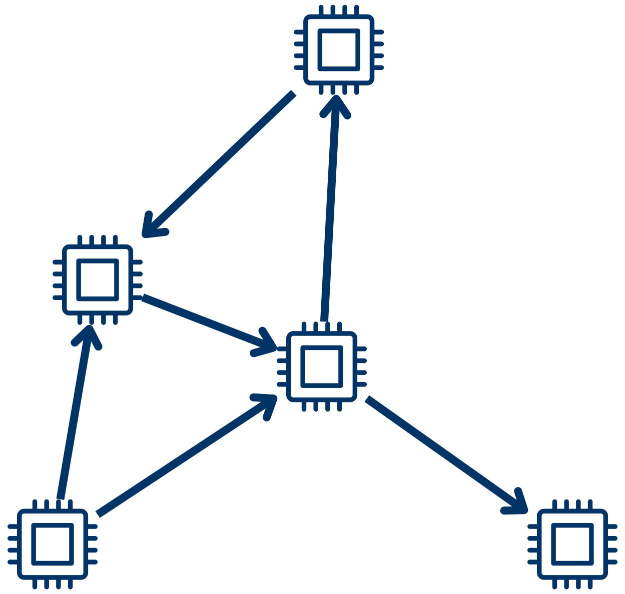
Cartesian Grid
(Halo Exchange)



Ring Exchange



Tree Communication



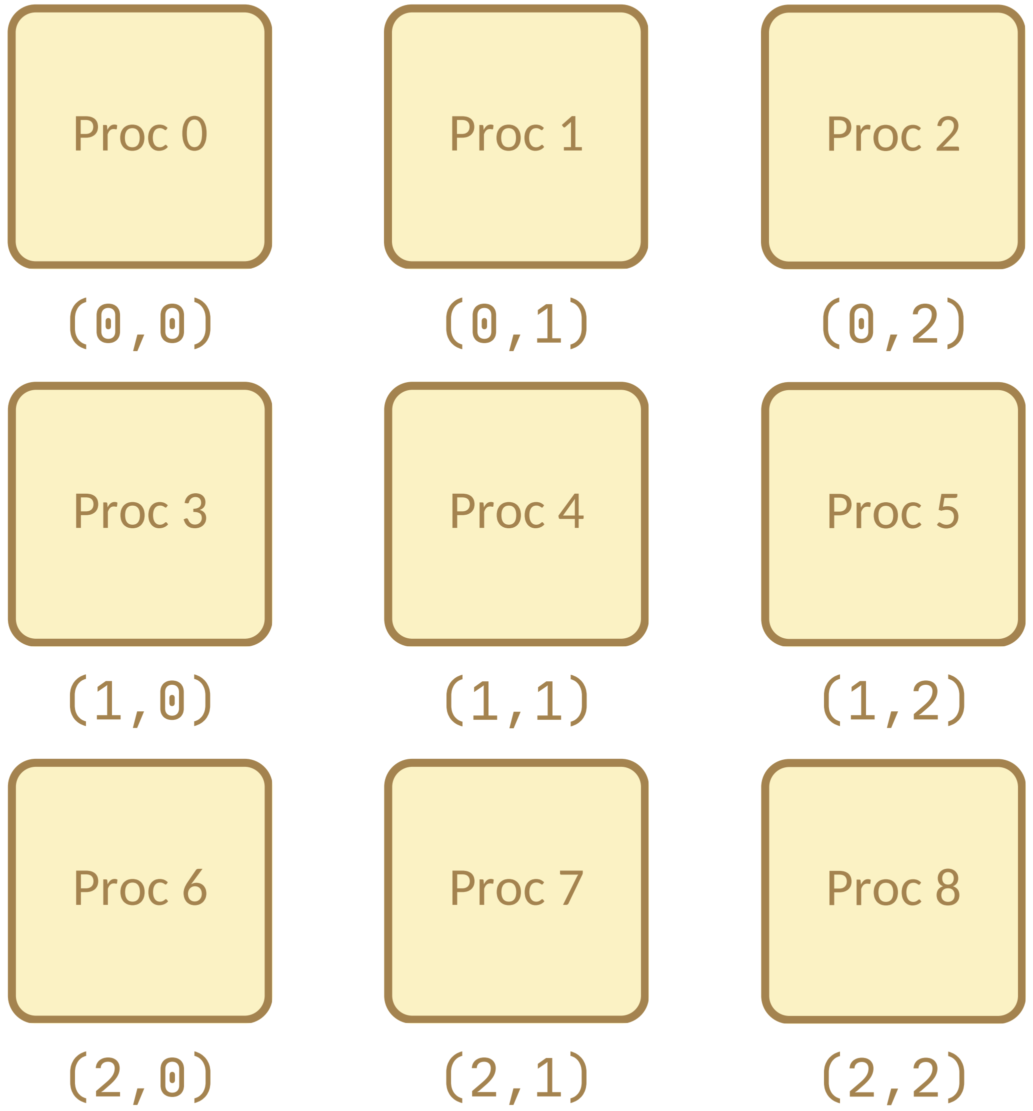
Graph Communication

Halo Exchange

MPI Applications

- MPI will create a communicator with a N-D mesh topology which can help communication by specyfing the coordinates of each rank.
- MPI will also be able to specify the neighboring processes for communication.
- Manually, you would have to perform this computation:

$$(p_i, p_j) = (\text{rank}/\text{size}_x, \text{rank}\%\text{size}_x)$$



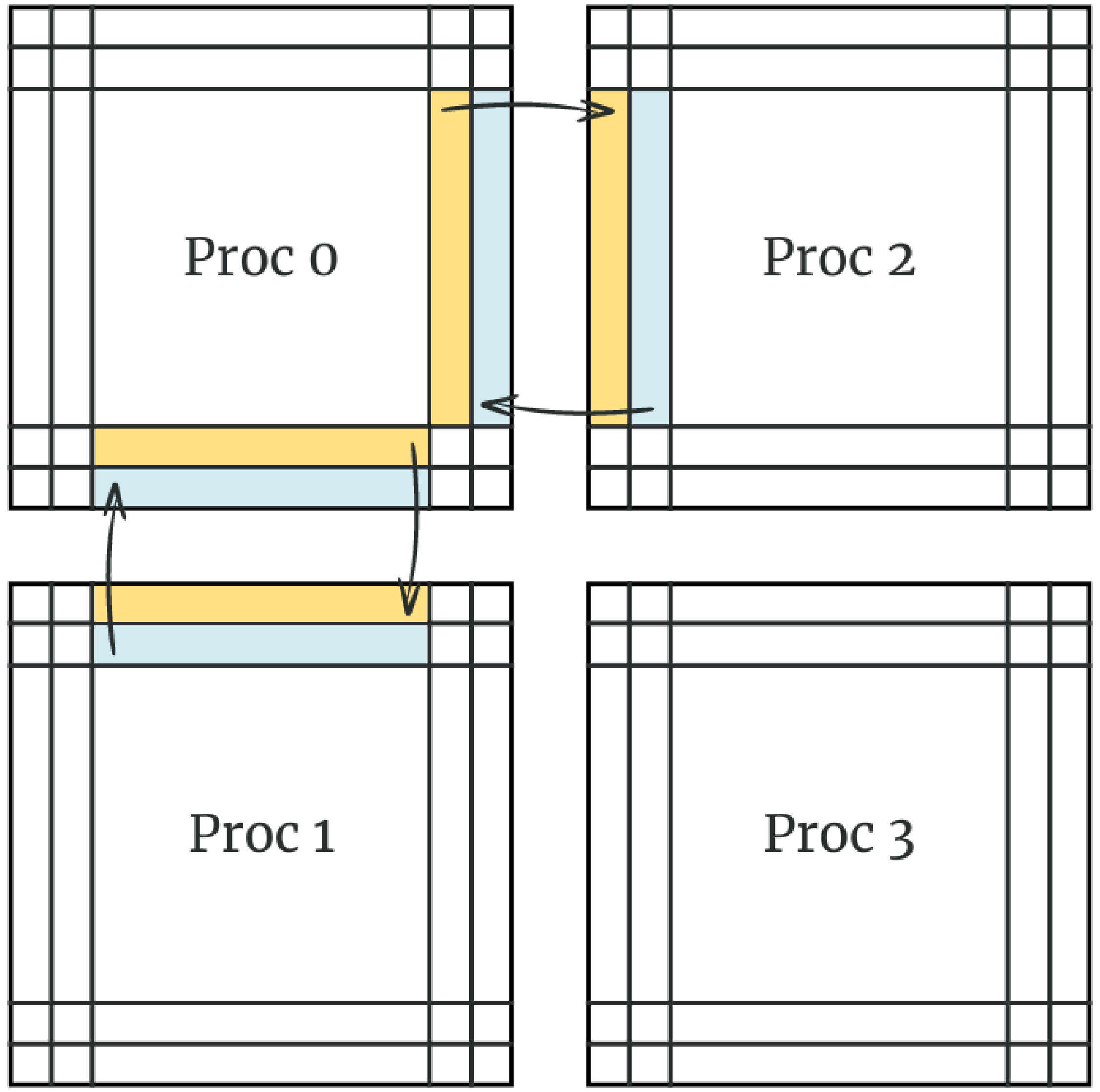
Proc 0 (0,0)	Proc 1 (0,1)	Proc 2 (0,2)
Proc 3 (1,0)	Proc 4 (1,1)	Proc 5 (1,2)
Proc 6 (2,0)	Proc 7 (2,1)	Proc 8 (2,2)

Halo Exchange

MPI Applications

```
int MPI_Cart_create(
  MPI_Comm comm_old, int ndims,
  const int dims[], const int periods[],
  int reorder, MPI_Comm *comm_cart
)
```

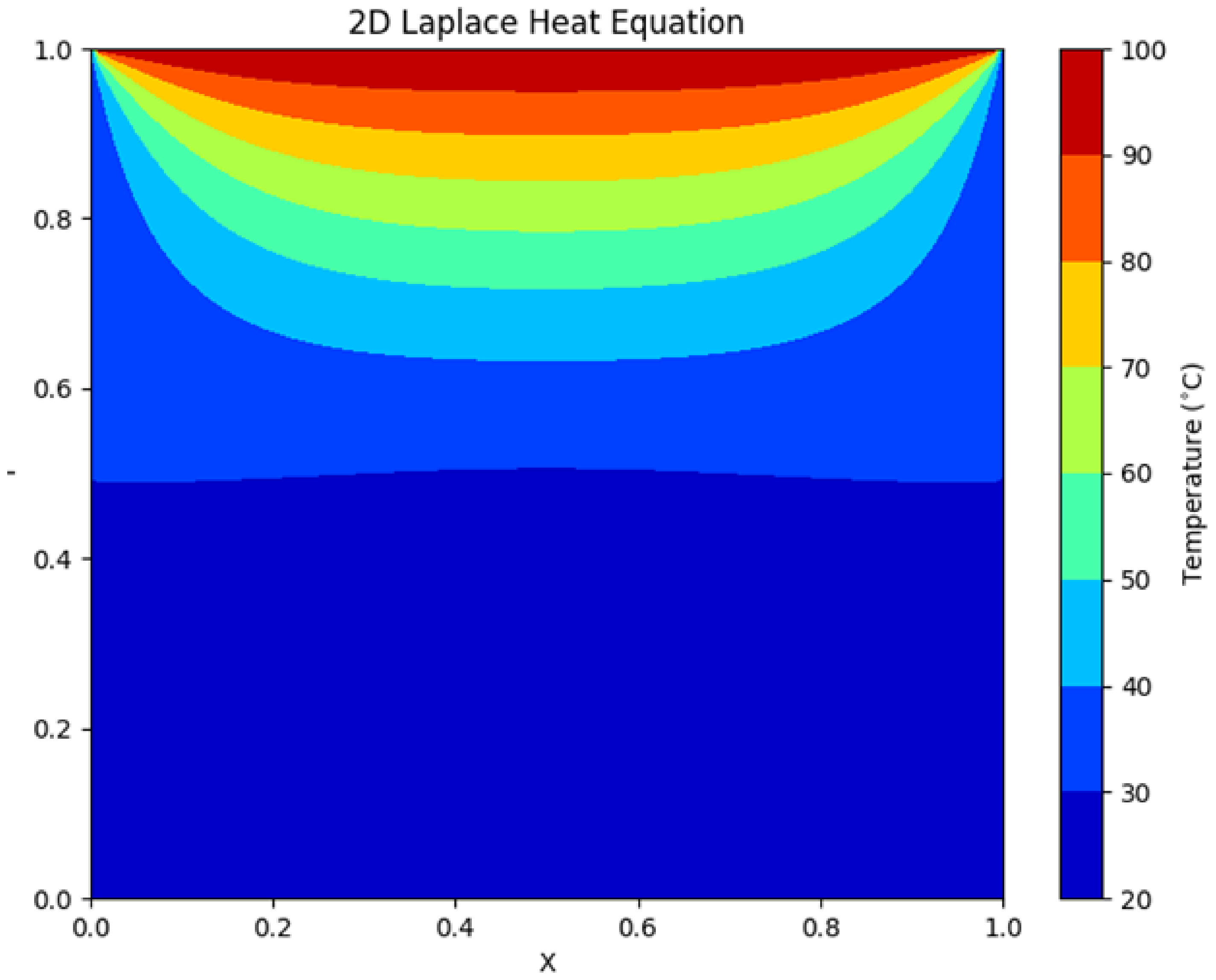
```
int MPI_Cart_shift(
  MPI_Comm comm, int direction,
  int disp, int *rank_source,
  int *rank_dest
)
```



Halo Exchange

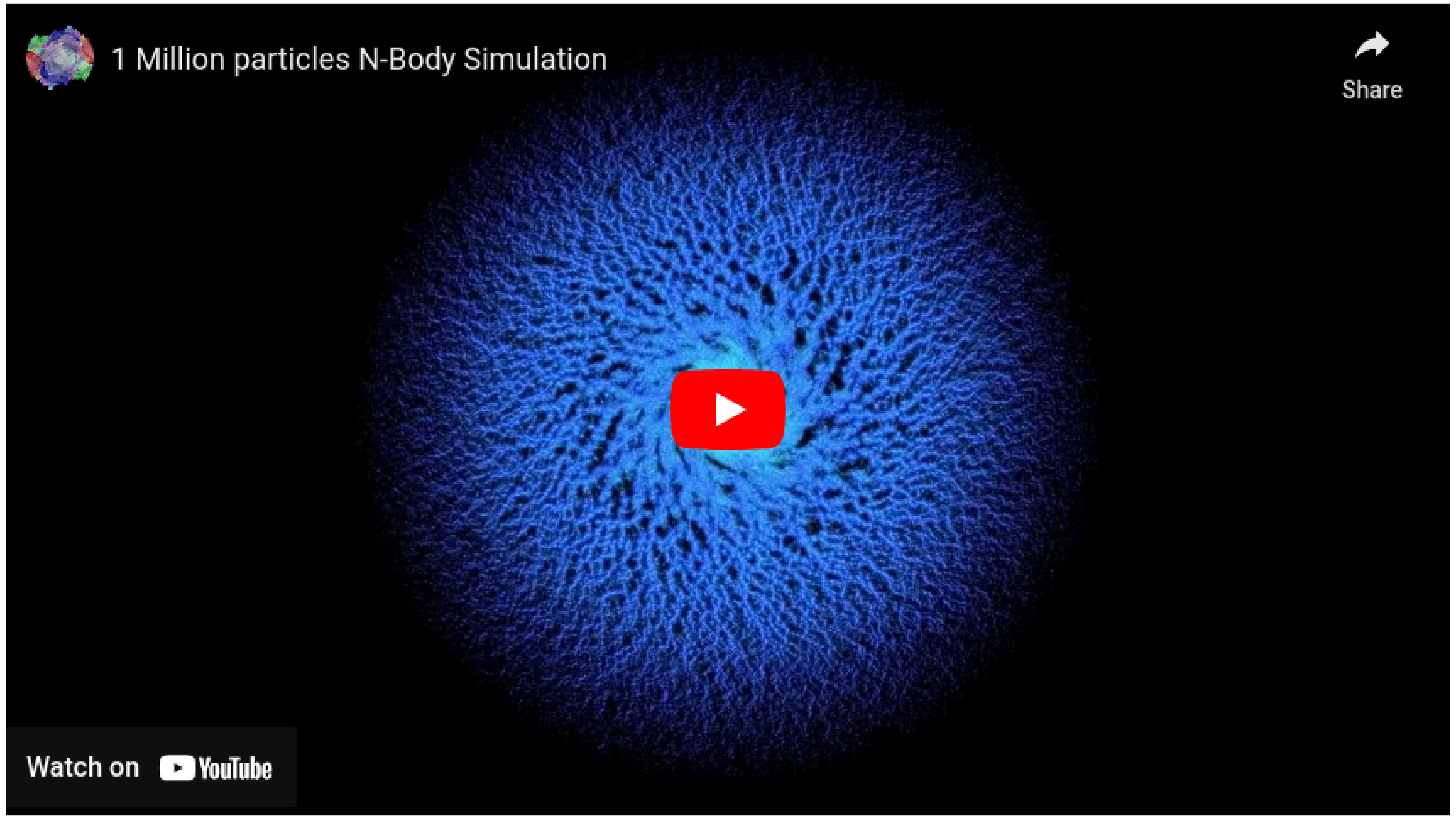
MPI Applications

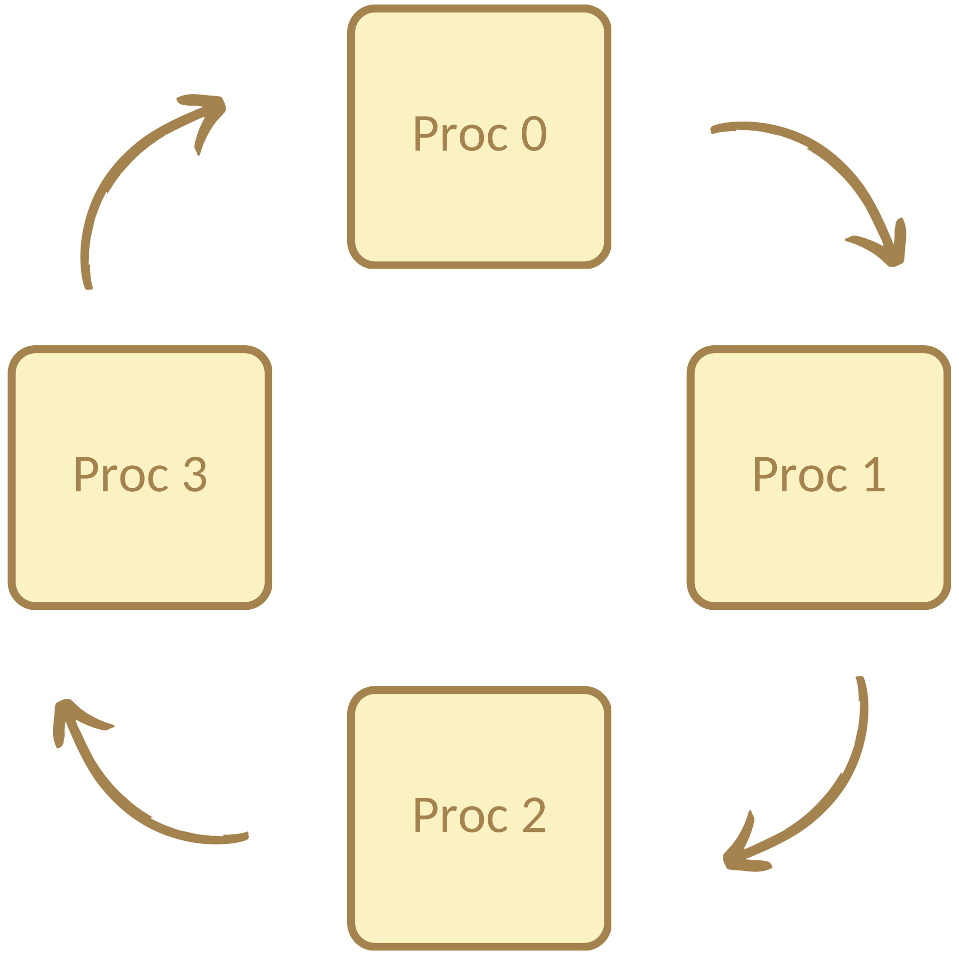
```
int MPI_Cart_create(
  MPI_Comm comm_old, int ndims,
  const int dims[], const int periods[],
  int reorder, MPI_Comm *comm_cart
)


int MPI_Cart_shift(
  MPI_Comm comm, int direction,
  int disp, int *rank_source,
  int *rank_dest
)
```



Ring Exchange

MPI Applications



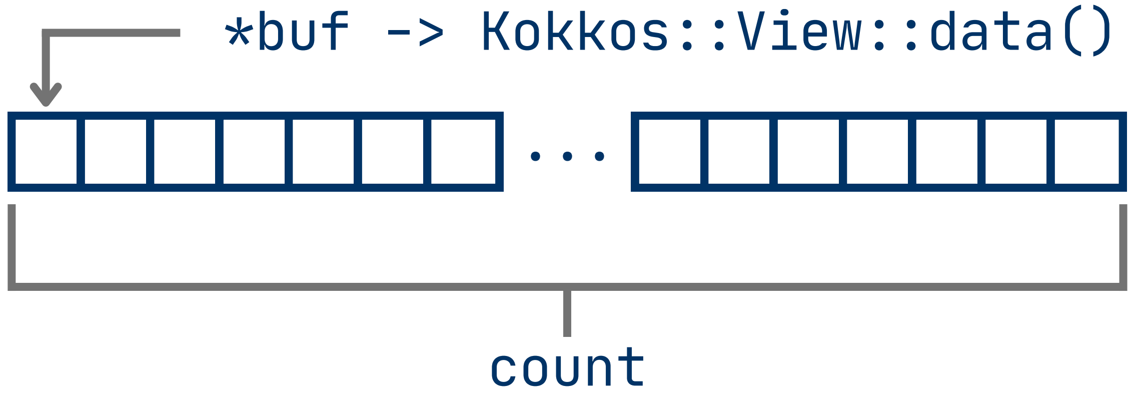


Using accelerators

MPI + Kokkos

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

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



Practice