Programming Massively Parallel Processors

Chapter20 Programming a heterogeneous computing cluster

242AIG18 JiYeong Yi

Chapter20

Programming a heterogeneous computing cluster

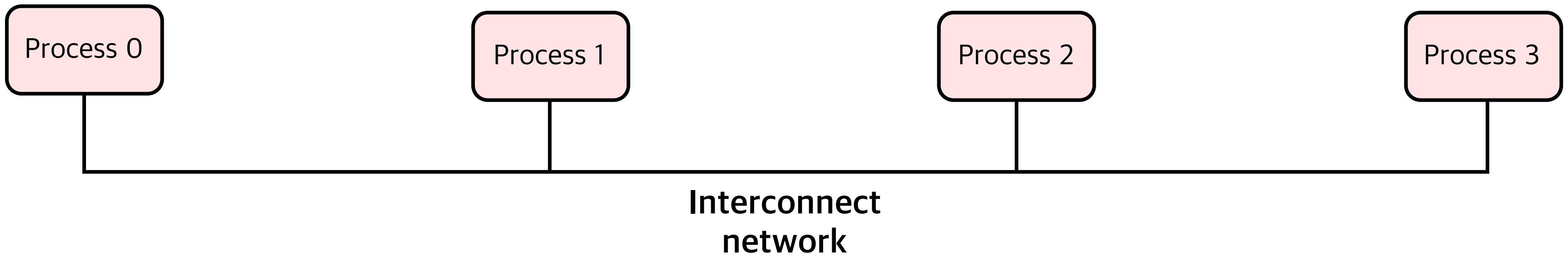
Table of contents

20.1 Background

- So far, we have focused on programming a heterogeneous computing system with one host and one device.
- In high-performance computing (HPC), applications require the aggregate computing power of a cluster of computing nodes. Many of the **HPC clusters** today have one or more hosts and one or more devices in each node.
- The dominating programming interface for computing clusters today is **MPI** (Message Passing Interface), which is a set of API functions for communication between processes running in a computing cluster.

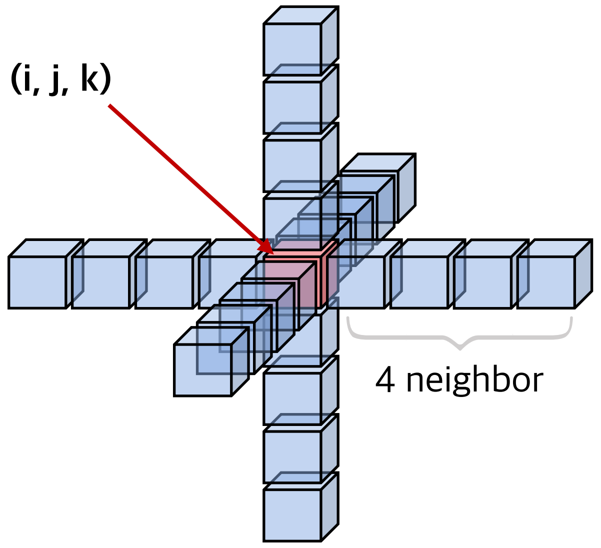
20.1 Background

- In a typical MPI application, data and work are partitioned among processes.
- As these processes progress, they may need data from each other. This need is satisfied by sending and receiving messages.
- In some cases, the processes also need to synchronize with each other and generate collective results when collaborating on a large task. This is done with collective communication API functions.



20.2 A running example

- As a running example three-dimensional (3D) stencil computation that was introduced in Chapter 8, Stencil will be used.
- In each iteration or time step, the value of the grid point is calculated as a weighted sum of neighbors (north, east, south, west, up, down) and its own value from the previous time step.
- A 25-point stencil computation example -> 4 neighbor point in each direction



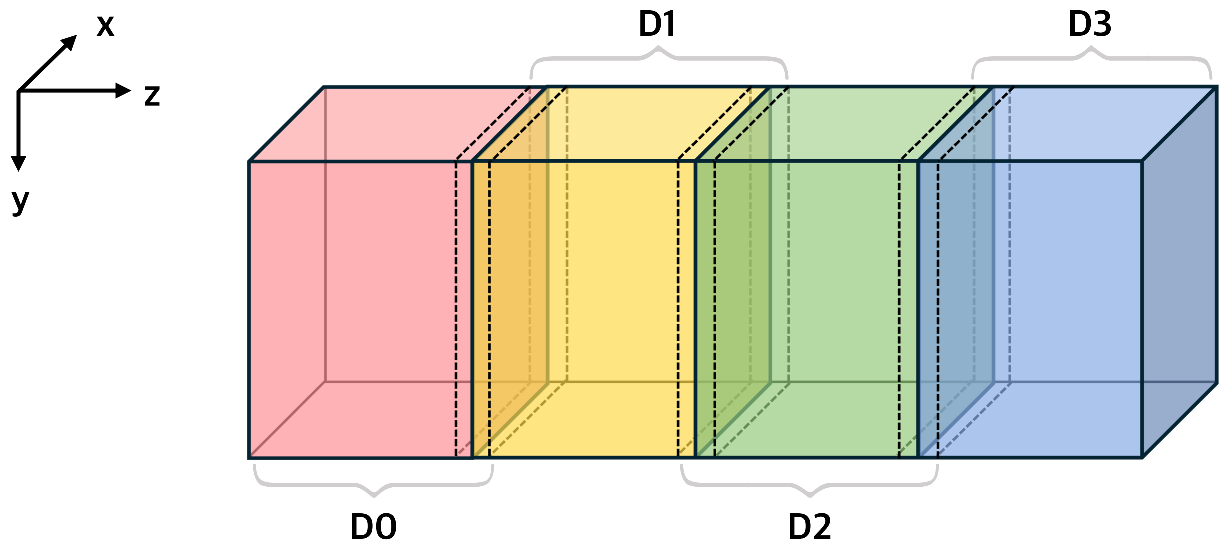
(i − 4, j, k), (i − 3, j, k), (i − 2, j, k), (i − 1, j, k),
(i + 1, j, k), (i + 2, j, k), (i + 3, j, k), (i + 4, j, k),

(i, j − 4, k), (i, j − 3, k), (i, j − 2, k), (i, j − 1, k),
(i, j + 1, k), (i, j + 2, k), (i, j + 3, k), (i, j + 4, k),

(i, j, k − 4), (i, j, k − 3), (i, j, k − 2), (i, j, k −1),
(i, j, k + 1), (i, j, k + 2), (i, j, k + 3),(i, j, k + 4)

20.2 A running example

- When one uses a computing cluster, it is common to divide the input data into several partitions, called domain partitions, and assign each partition to a node in the cluster.



- The 3D array is divided into four domain partitions: D0, D1, D2, and D3. Each of the partitions will be assigned to an MPI compute process.

20.3 Message passing interface basics

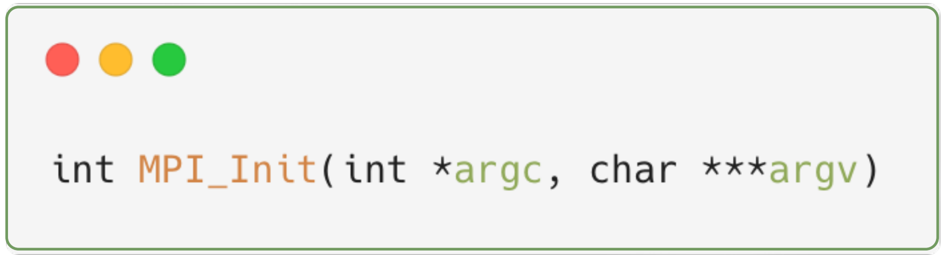
- All MPI processes execute the same program. The MPI system provides a set of API functions to establish communication systems that allow the processes to communicate with each other.

- Five essential MPI function that set up and tear down the communication system:

    1. MPI_Init(): Initialize MPI
    2. MPI_Comm_rank(): Rank of the calling process in group of comm
    3. MPI_Comm_size(): Number of processes in the group of comm
    4. MPI_Comm_abort(): Terminate MPI communication connection with an error flag
    5. MPI_Finalize(): Ending an MPI application, close all resources

20.3 Message passing interface basics

MPI_Init()



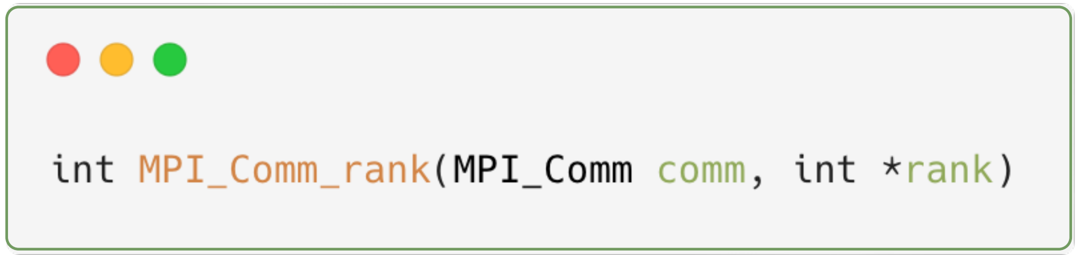
```
int MPI_Init(int *argc, char ***argv)
```

- argc is the pointer to the number of arguments.
- argv is the pointer to the vector of arguments.

- To launch an MPI application in a cluster, a user needs to supply the executable file of the program to the *mpirun* command or the *mpiexec* command in the login node of the cluster.
- Each process starts by initializing the MPI runtime with an MPI_Init() call.
  This initializes the communication system for all the processes that are running the application.

20.3 Message passing interface basics

MPI_Comm_rank()



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

- comm is the communicator.
- rank is a pointer to the identifier of the calling process within the group of the communicator.

- MPI_Comm_rank() returns a unique number to each calling process, which is called the **MPI rank** or **process id** for process. It uniquely identifies the process in a communication.
- The numbers that are received by the processes vary from 0 to the number of processes minus 1.

20.3 Message passing interface basics

Communicator specifies the scope of the collection of processes that form the group for the purpose of communication.

MPI_Comm_rank()

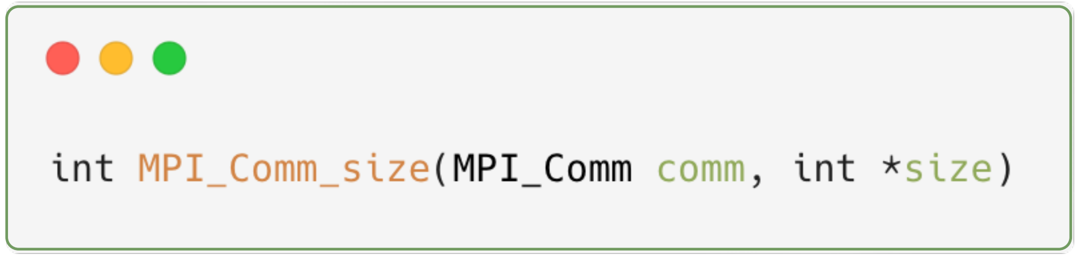
```
int MPI_Comm_rank(MPI_Comm comm, int *rank)
```

- comm is the **communicator**.
- rank is a pointer to the identifier of the calling process within the group of the communicator.

- MPI_Comm_rank() returns a unique number to each calling process, which is called the **MPI rank** or **process id** for process. It uniquely identifies the process in a communication.
- The numbers that are received by the processes vary from 0 to the number of processes minus 1.

20.3 Message passing interface basics

MPI_Comm_size()



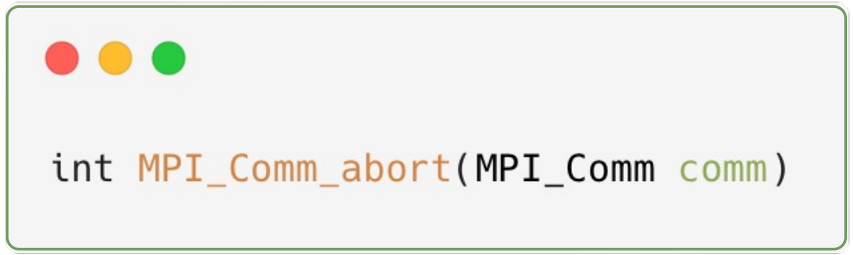
```
int MPI_Comm_size(MPI_Comm comm, int *size)
```

- comm is the communicator.
- size indicates the number of processes in the group for the communicator.

- MPI_Comm_size() returns the toal number of MPI processes running in the communicator.
- The system may or may not be able to create all the processes that the user requested. Therefore, it is a necessary for an MPI application program to check the actual number of processes that are running.

20.3 Message passing interface basics

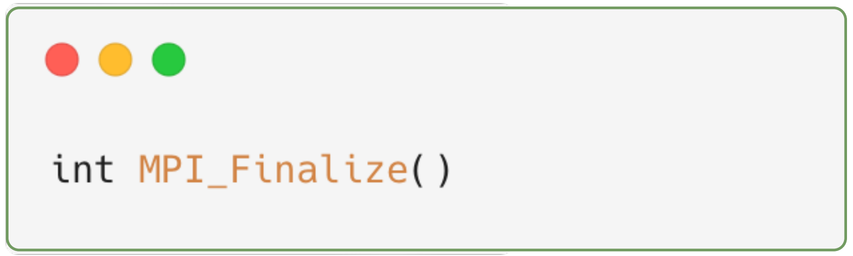
MPI_Comm_abort()



```
int MPI_Comm_abort(MPI_Comm comm)
```

- MPI_Comm_abort() terminates the communication connections and returns with an error flag value 1.

MPI_Finalize()



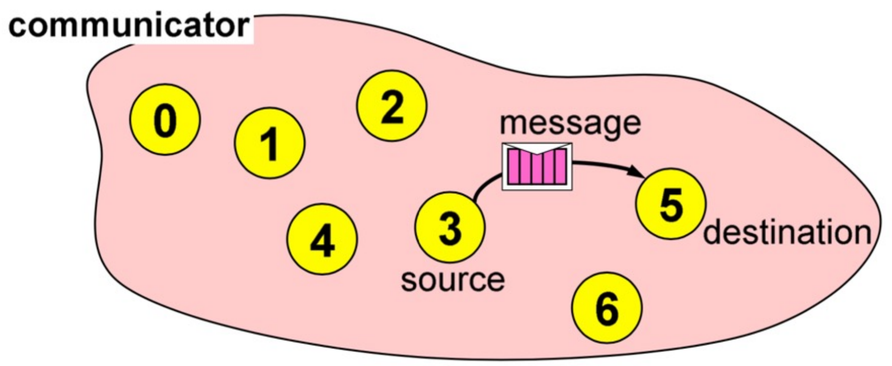
```
int MPI_Finalize()
```

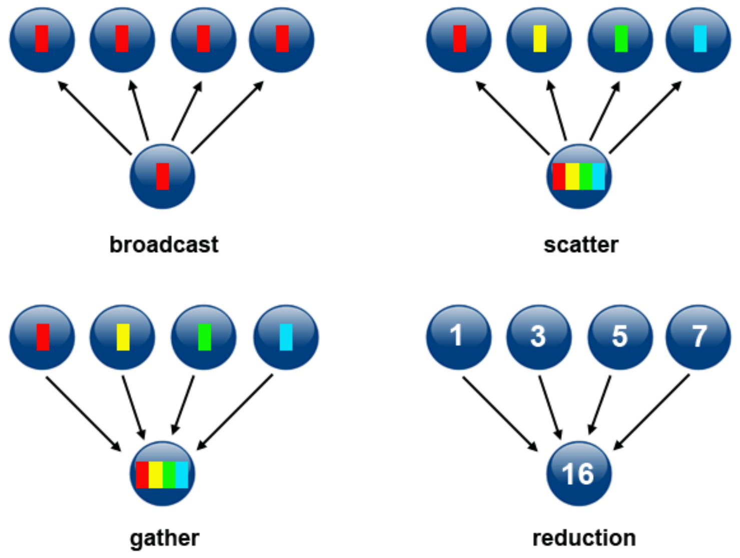
- MPI_Finalize() frees all MPI communication resources that are allocated to the application.

20.4 Message passing interface point-to-point communication

- MPI supports two major types of communication:

Point-to-Point communication	Point-to-Point communication is the simplest communication as it involves only two processes. One of the processes acts as a sender and the other one as the receiver.
Collective communication	Collective communication is a method of communication which involves participation of all processes in a communicator.





20.4 Message passing interface point-to-point communication

- In the point-to-point communication, the source process calls the **MPI_Send()** function, and the destination process calls the **MPI_Recv()** function.
- Syntax for using MPI_Send() and MPI_Recv() function:

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

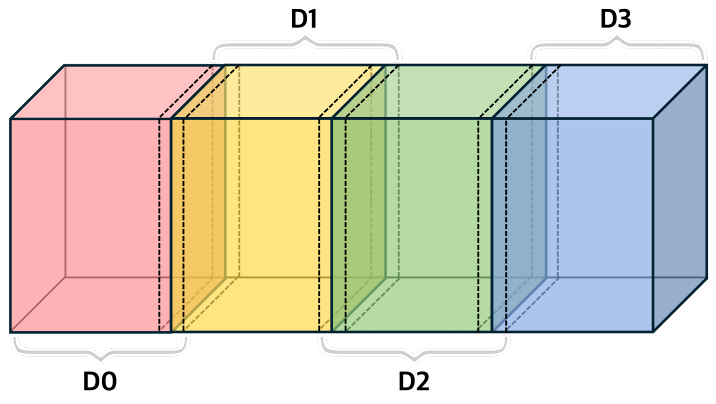
```
int MPI_Recv(void *buf, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm)
```

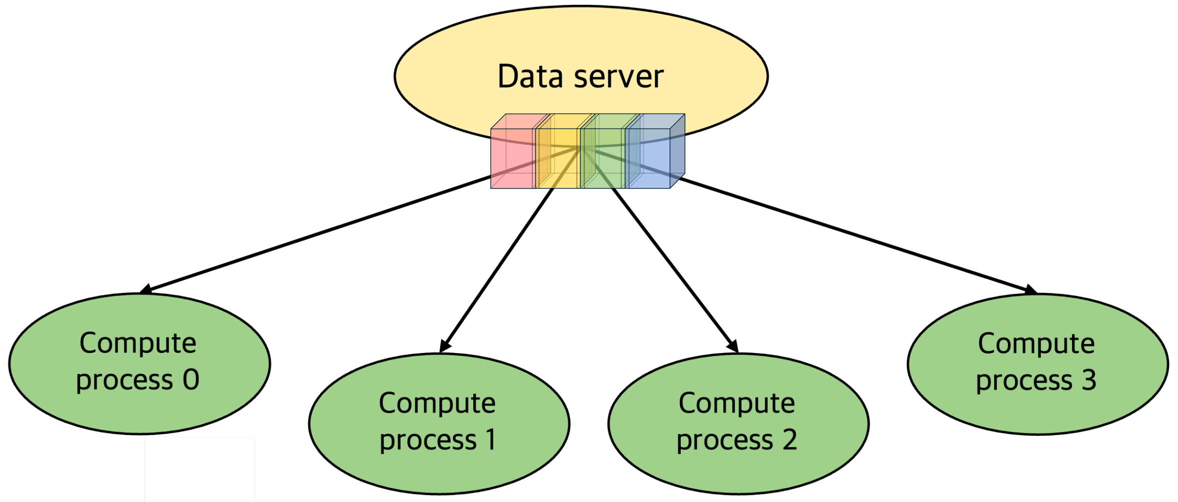
```
int MPI_Sendrecv(void *sendbuf, int sendcount, MPI_Datatype sendtype, int dest, int sendtag,
                 void *recvbuf, int recvcount, MPI_Datatype recvtype, int source, int recvtag,
                 MPI_Comm comm, MIP_Status *status)
```

20.4 Message passing interface point-to-point communication

* np: The number of processes in the communicator

Data server	Process id: (np – 1)	Data server process performs the I/O services.
Compute processes	Process id: 0 ~ (np - 2)	Compute processes perform the calculation.

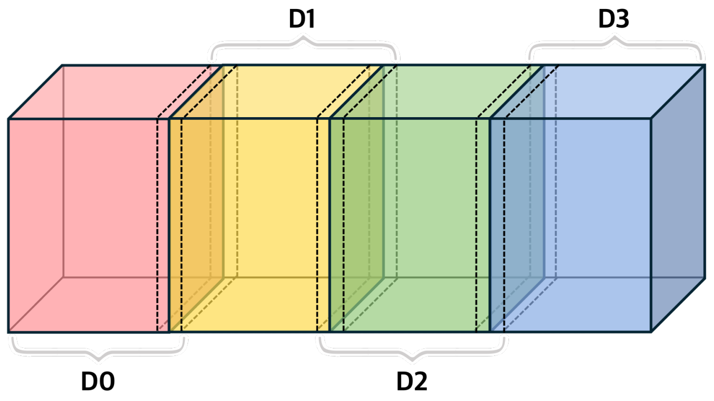


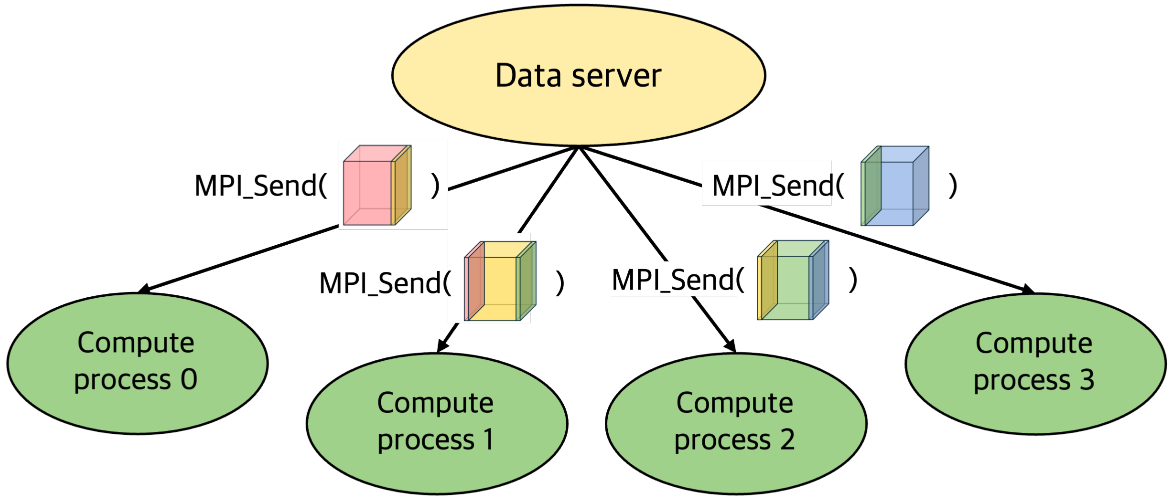


20.4 Message passing interface point-to-point communication

* np: The number of processes in the communicator

Data server	Process id: (np – 1)	Data server process performs the I/O services.
Compute processes	Process id: 0 ~ (np - 2)	Compute processes perform the calculation.

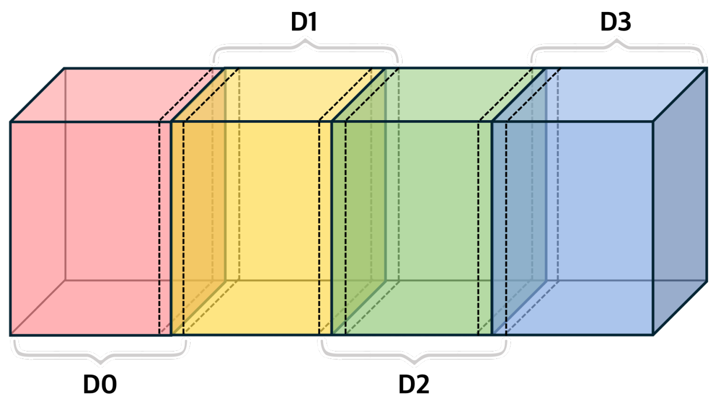


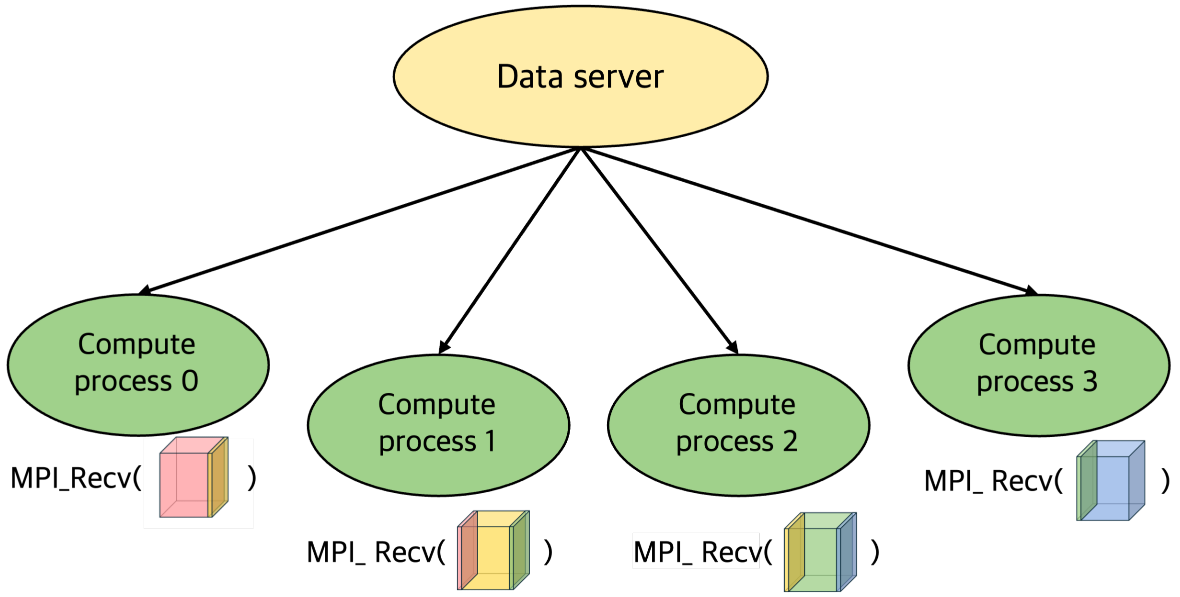


20.4 Message passing interface point-to-point communication

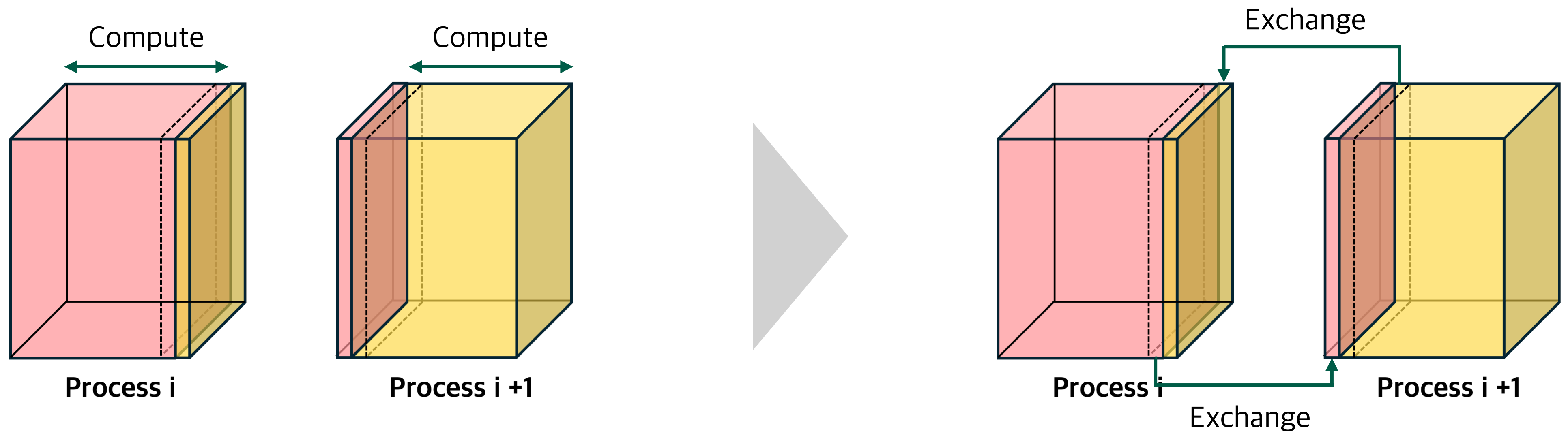
* np: The number of processes in the communicator

Data server	Process id: (np – 1)	Data server process performs the I/O services.
Compute processes	Process id: 0 ~ (np - 2)	Compute processes perform the calculation.



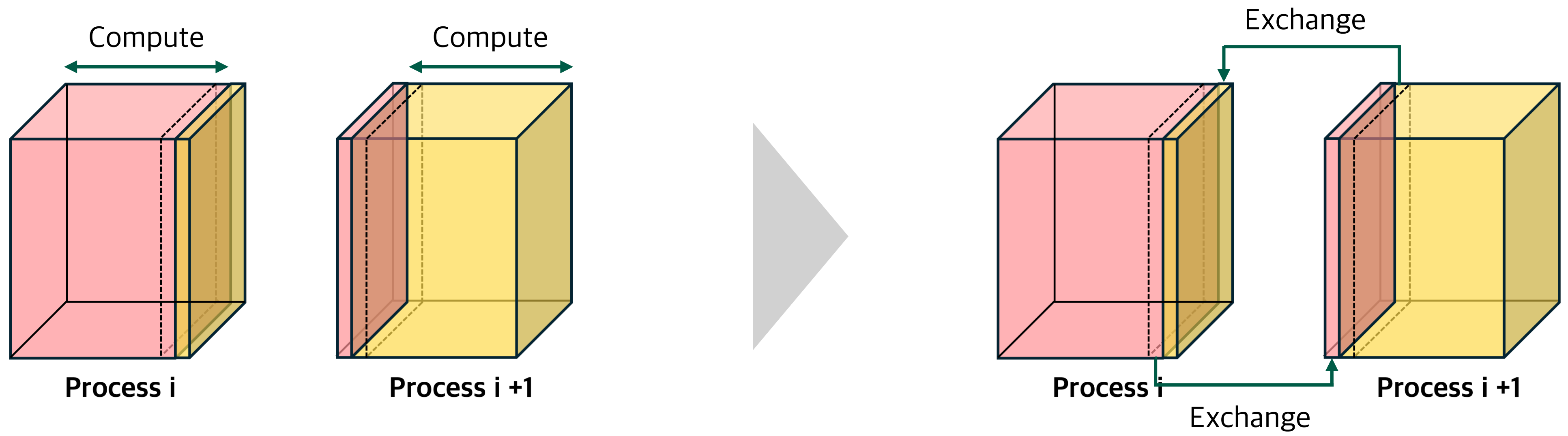


20.5 Overlapping computation and communication



- A simple way to perform the computation steps is for each compute process to perform a computation step on its entire partition, exchange halo data with the left and right neighbors, and repeat.
- While this is a very simple strategy, it is <u>not very effective</u>.
  The reason is that this strategy forces the system to be in one of the two modes.

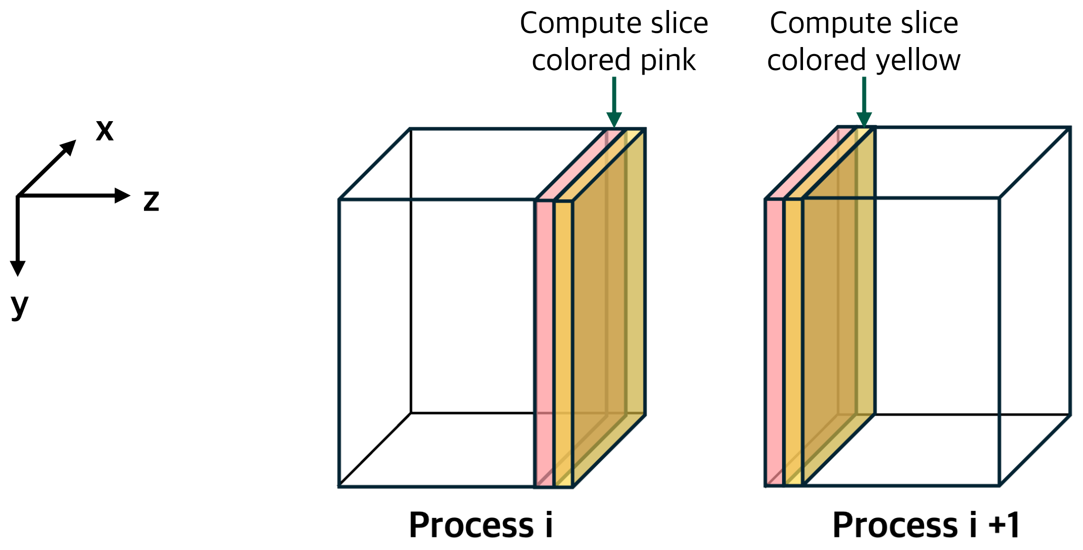
20.5 Overlapping computation and communication



- In the first mode, all compute processes are performing computation steps. During this time, the communication network is not used.
- In the second mode, all compute processes are exchanging halo data with their left and right neighbors. During this time, the computation hardware is not well utilized.

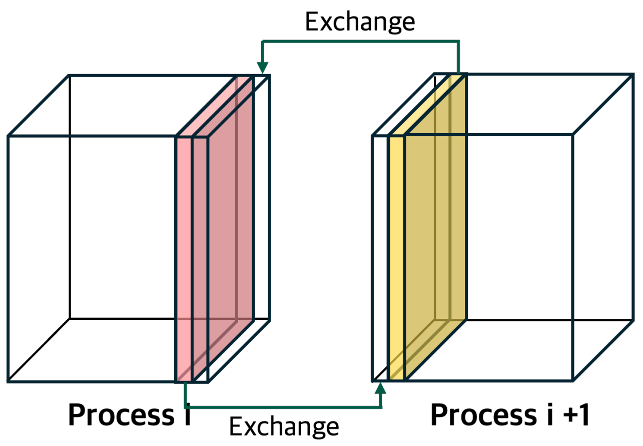
20.5 Overlapping computation and communication

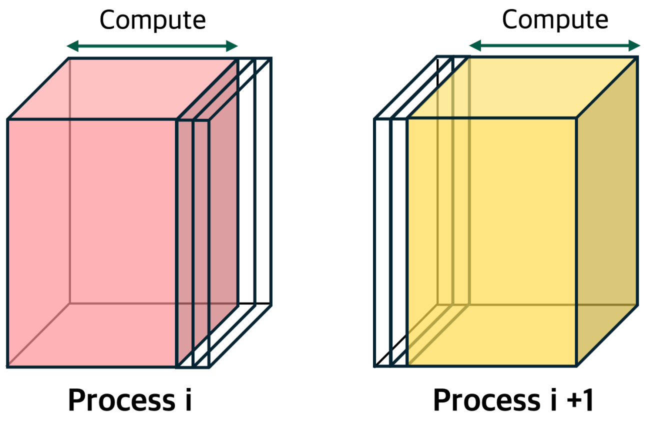
- Better performance by utilizing both the communication network and the computation hardware all the time can be achieved by dividing the computation tasks of each compute process into two stages.
- During the **first stage (stage 1)**, each compute process calculates its boundary slices that will be needed as halo cells by its neighbors in the next iteration.



20.5 Overlapping computation and communication

- During the second stage (stage 2), each compute process performs two activities in parallel.
- The first is to communicate its new boundary values to its neighbor processes.
  This is done by first copying the data from the device memory into the host memory, followed by sending MPI messages to the neighbors.
- The second activity is to calculate the rest of the data in the partition.

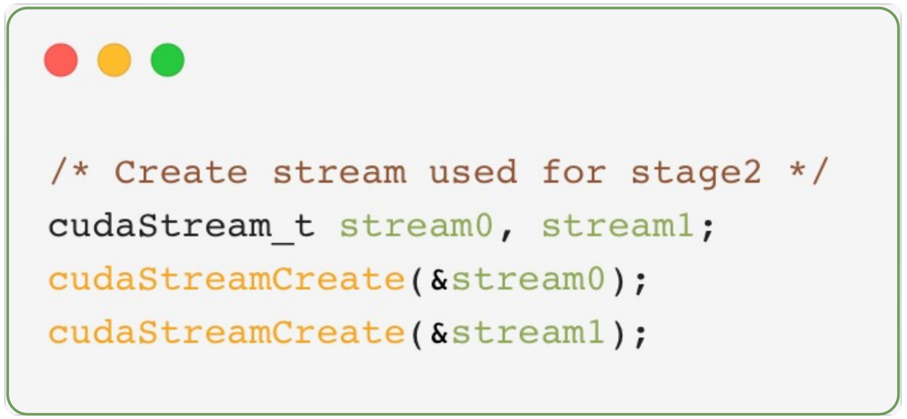




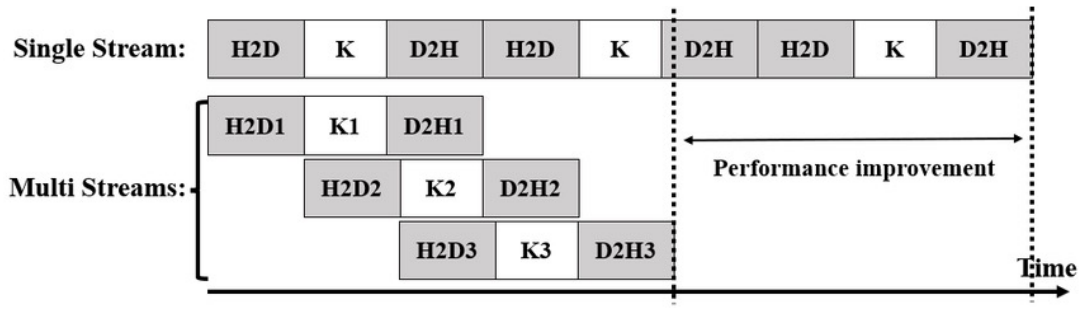
20.5 Overlapping computation and communication

- The advanced CUDA feature that is used for overlapping communication with computation is **streams**, a feature that supports managed concurrent execution of CUDA API functions.

- All operations in the **same stream** will be done sequentially according to the order in which they are placed into that stream. However, operations in **different streams** can be executed in parallel without any ordering constraint.



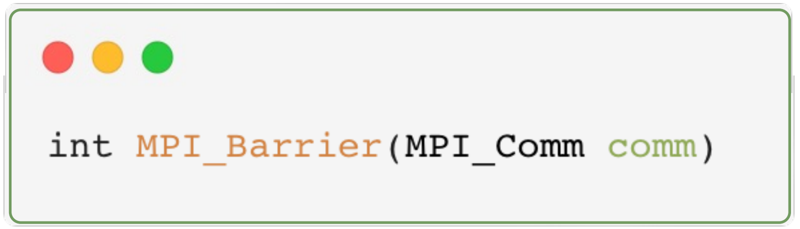
```
/* Create stream used for stage2 */
cudaStream_t stream0, stream1;
cudaStreamCreate(&stream0);
cudaStreamCreate(&stream1);
```



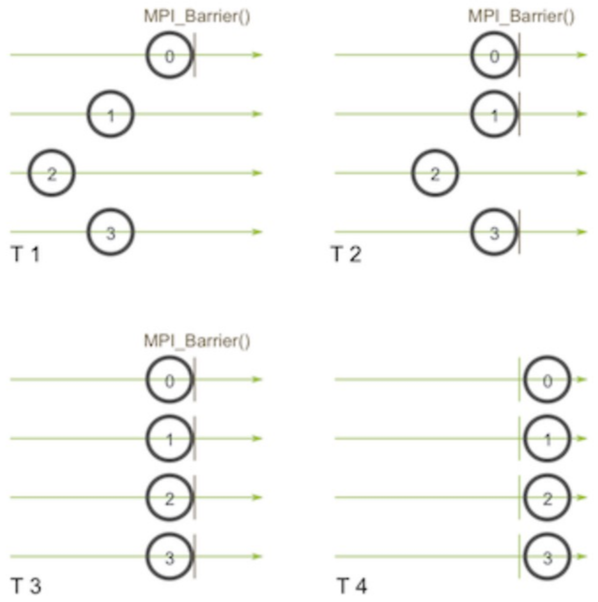
20.6 Message passing interface collective communication

- The other commonly used group collective communication types are barrier, broadcast, reduce, gather, and scatter. Barrier synchronization MPI_Barrier() is perhaps the most commonly used collective communication function.

- MPI barrier synchronization is similar to the CUDA __syncthreads() across threads in a block.
  None of the processes can continue their execution beyond this point until all have reached this point.

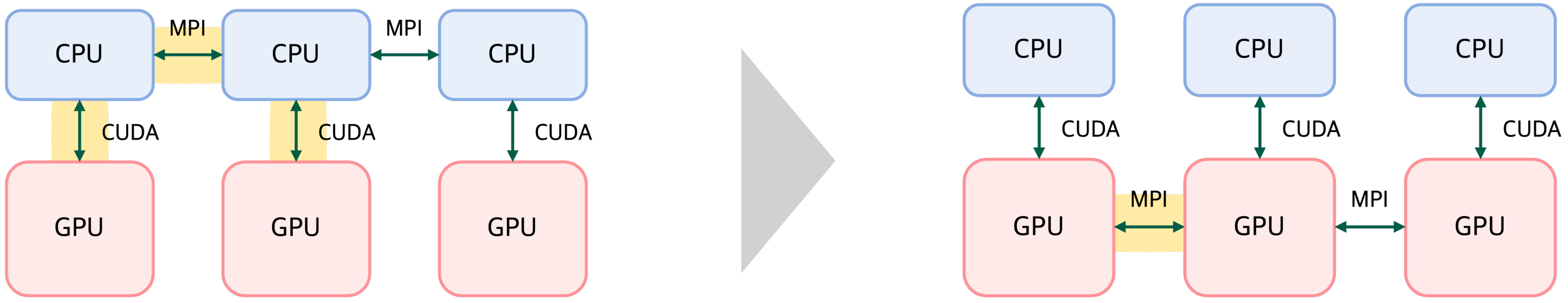


```
int MPI_Barrier(MPI_Comm comm)
```



20.7 CUDA aware message passing interface

- Modern MPI implementations are aware of the CUDA programming model and are designed to minimize the communication latency between GPUs.
- CUDA-aware MPI implementations are capable of sending messages from the GPU memory in one node to the GPU memory in a different node. This effectively removes the need for device-to-host data transfers before sending MPI messages and host-to-device data transfers after receiving an MPI message.


CPU
MPI
CPU
MPI
CPU
CUDA
CUDA
CUDA
GPU
GPU
GPU
CPU
CPU
CPU
CUDA
CUDA
CUDA
GPU
MPI
GPU
MPI
GPU