Programming Using the Message-Passing Paradigm

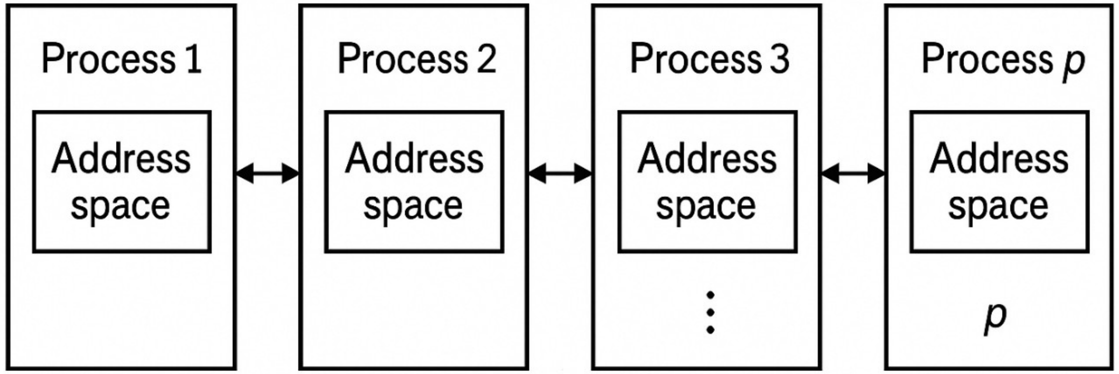
By Nilesh Ghavate

Outline

- **Principles of Message Passing Programming,**
- The Building Blocks: Send and Receive Operations
- MPI :Message Passing Interface
- Topology and Embedding
- Overlapping Communication with Computation
- Collective Communication and Computation Operations.
- Groups and Communicators

Principles of Message-Passing Programming

- Each data element must belong to one of the partitions of the space; hence, data must be explicitly partitioned and placed.
- All interactions (read-only or read/write) require cooperation of two processes - the process that has the data and the process that wants to access the data.



The logical view of a machine supporting the message-passing paradigm consists of p processes, each with its own exclusive address space.

Principles of Message-Passing Programming

- Message-passing programs are often written using the *asynchronous* or *loosely synchronous* paradigms.
- In the **asynchronous** paradigm, all concurrent tasks execute asynchronously.
- In the **loosely synchronous** model, tasks or subsets of tasks synchronize to perform interactions. Between these interactions, tasks execute completely asynchronously.
- Most message-passing programs are written using the *single program multiple data* (SPMD) model.

SPMD Approach

- In **SPMD (Single Program Multiple Data)**, all processes execute the same code, with differences only for specific processes (e.g., the "root" process).
- This approach is common in message-passing programs, as it simplifies the design and makes it more scalable.
- **SPMD** can be **loosely synchronous** or **completely asynchronous**, offering flexibility in how tasks are coordinated.

Outline

- Principles of Message Passing Programming,
- **The Building Blocks: Send and Receive Operations**
- MPI :Message Passing Interface
- Topology and Embedding
- Overlapping Communication with Computation
- Collective Communication and Computation Operations.
- Groups and Communicators

The Building Blocks: Send and Receive Operations

- The prototypes of these operations are as follows:

```
send(void *sendbuf, int nelems, int dest)
receive(void *recvbuf, int nelems, int source)
```

- Consider the following code segments:

```
        P0
P1

        a = 100;
receive(&a, 1, 0)
        send(&a, 1, 1);                                  printf("%d\n", a);
        a = 0;
```

- The semantics of the send operation require that the value received by process P1 must be 100 as opposed to 0.
- Most message passing platforms have **additional hardware support** for sending and receiving messages. They **may support DMA (direct memory access) and asynchronous message** transfer using network interface hardware.
- This motivates the design of the send and receive protocols.

The Building Blocks: Send and Receive Operations

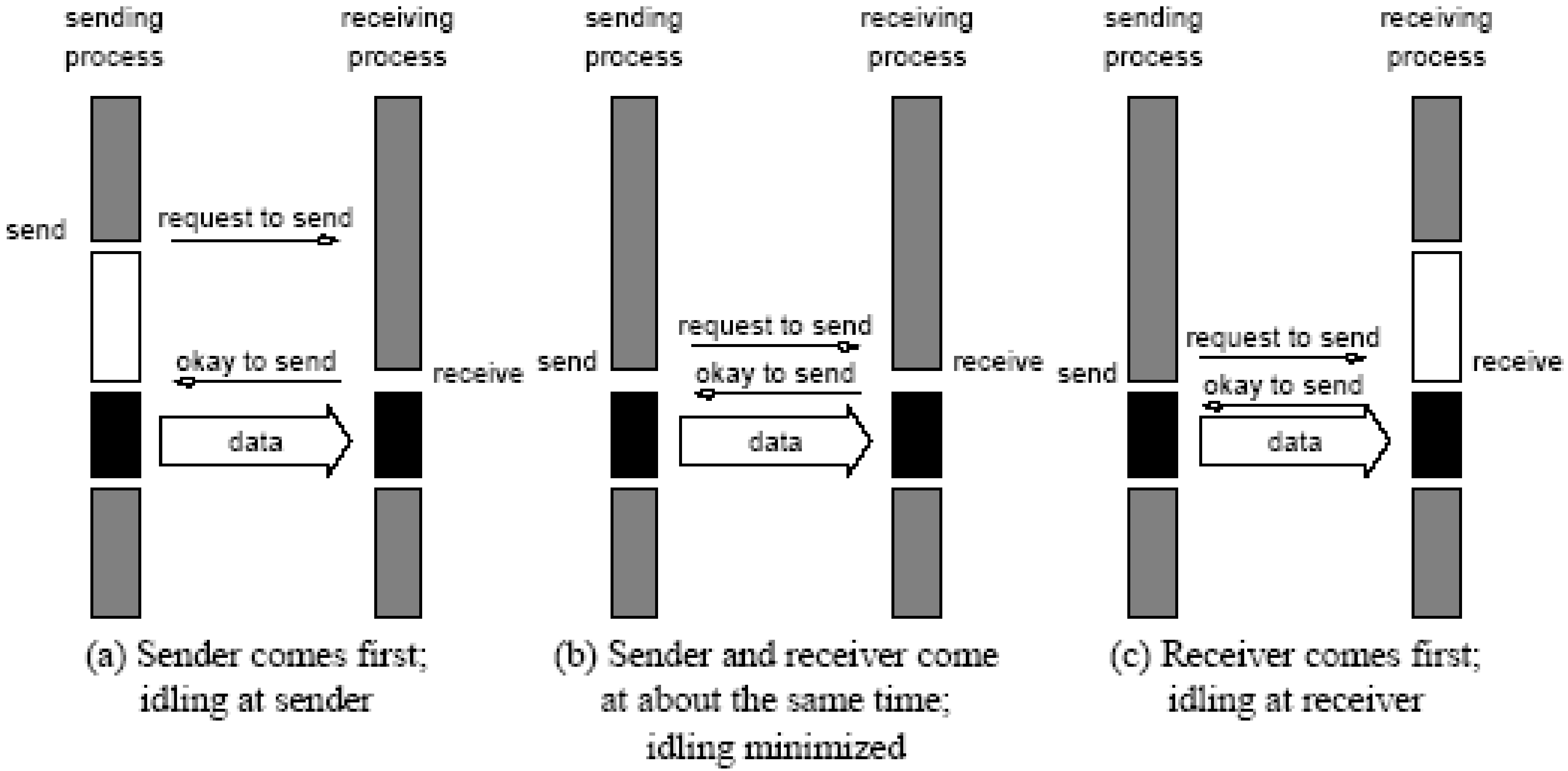
When using the **send operation**, you want to ensure that the sending process can safely continue its execution, without violating the program's logic or correctness, even if the receiver hasn't yet received the data.

- **Key Concepts:**
- **Semantically Safe**: In the context of parallel programming, this means that the send operation ensures the message can be passed without causing errors in the program's logic or flow, even if the receiver hasn't yet processed the message.
- **Send Operation**: When a process sends data to another, it typically involves some form of communication. However, in many systems, you might want the sending process to wait until it can guarantee that sending the data won’t cause logical problems later, even though the message may not yet have been received by the receiving process.

Blocking Message Passing Operations

- A simple method for forcing send/receive semantics is for the send operation to return only when it is safe to do so.
- Two Mechanisms to Achieve Semantic Safety:
  - **Synchronous Send (With Blocking or Handshake)**
  - **(Asynchronous) Buffered Send (With a Message Buffer)**:

Blocking Non-Buffered Message Passing Operations



Handshake for a blocking non-buffered send/receive operation.
It is easy to see that in cases where sender and receiver do not
reach communication point at similar times, there can be considerable idling overheads.

Blocking Non-Buffered Message Passing Operations

Deadlocks in Blocking Non-Buffered Operations

Consider the following simple exchange of messages that can lead to a deadlock:

```
P0

send(&a, 1, 1);
receive(&b, 1, 1);
```

```
P1

send(&a, 1, 0);
receive(&b, 1, 0);
```

Blocking Buffered Message Passing Operations

- In the non-buffered blocking send, the operation does not return until the matching receive has been encountered at the receiving process.
- Idling and deadlocks are major issues with non-buffered blocking sends.
- In buffered blocking sends, the sender simply copies the data into the designated buffer and returns after the copy operation has been completed. The **data is copied at a buffer at the receiving end** as well.
- Buffering alleviates idling at the expense of copying overheads.

Blocking Buffered Message Passing Operations

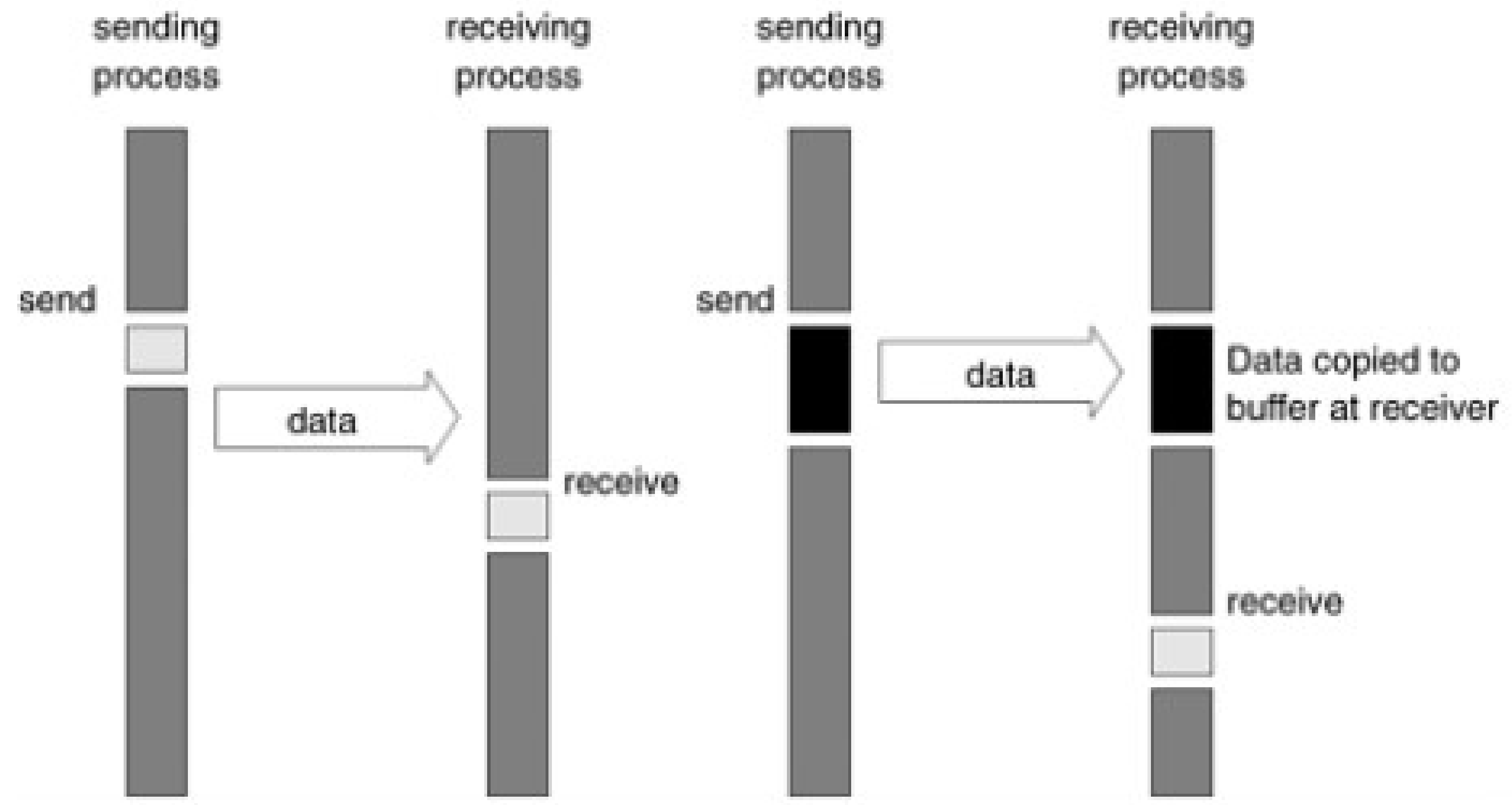


Figure 6.2. Blocking buffered transfer protocols: (a) in the presence of communication hardware with buffers at send and receive ends; and (b) in the absence of communication hardware, sender interrupts receiver and deposits data in buffer at receiver end.

Blocking Buffered Message Passing Operations

Deadlocks in Buffered Send and Receive Operations

```
P0                          P1

receive(&a, 1, 1);          receive(&a, 1, 0);
send(&b, 1, 1);             send(&b, 1, 0);
```

A simple code fragment such as the following deadlocks since both processes wait to receive data but nobody sends it.

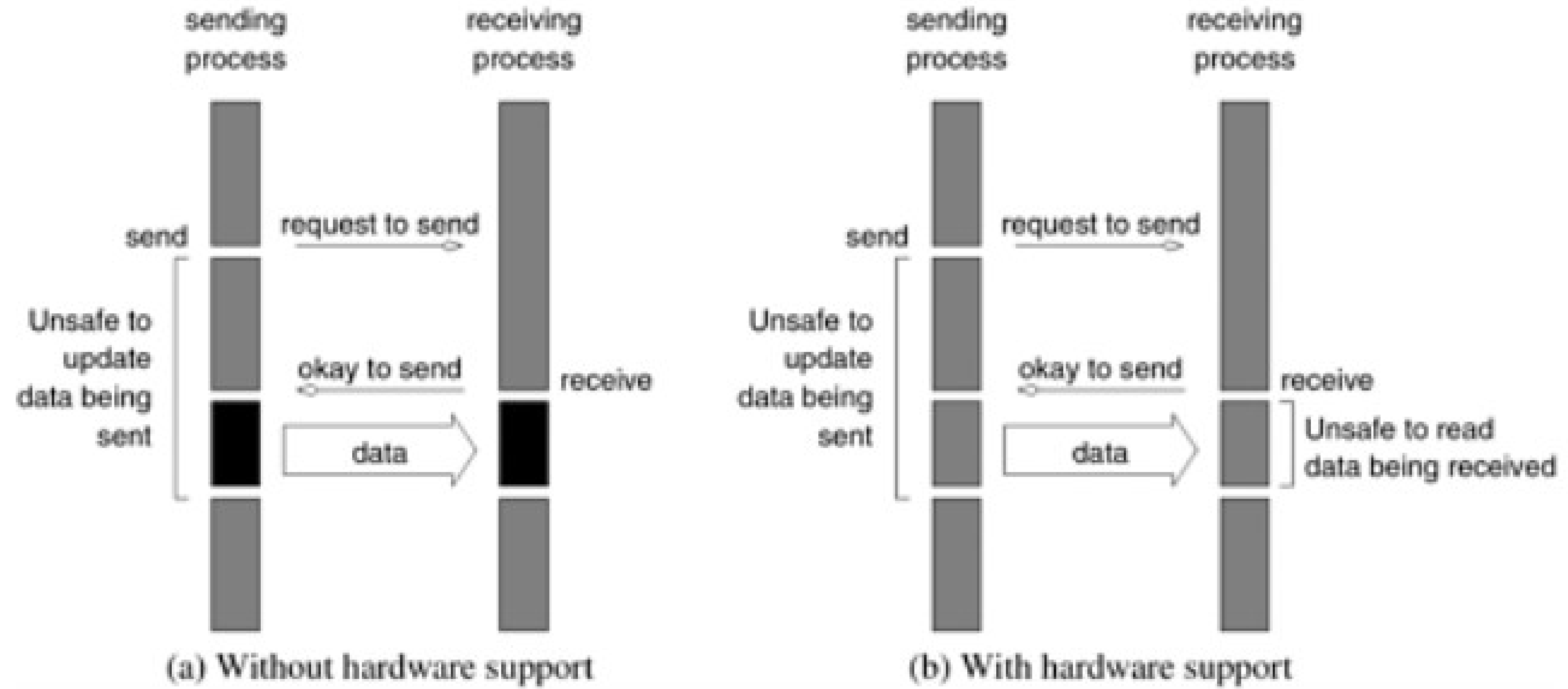
Blocking vs. Non-Blocking Communication

- **Blocking Communication:**
- In blocking protocols, a process waits (idles) until the data transfer is complete.
  - If no buffering is used, the sender and receiver must synchronize, meaning one might have to wait for the other.
  - If buffering is used, extra memory is needed to temporarily store data, requiring buffer management overhead.

Blocking vs. Non-Blocking Communication

- **Non-Blocking Communication:**
- In non-blocking protocols, the function **returns immediately**, without waiting for the operation to complete.
- The program can continue executing other tasks while the communication is still in progress.
- However, the **responsibility** of ensuring correctness is placed on the **programmer**.

Non-Blocking Non-Buffered Message Passing Operations



Non-blocking non-buffered send and receive operations (a) in absence of communication hardware;
(b) in presence of communication hardware.

Non-Blocking Buffered Message Passing Operations

- **Non-Blocking Send with Direct Memory Access (DMA):**
  - When a sender initiates a non-blocking send, instead of waiting for the receiver to be ready, it **hands over** the data to a **buffer** and starts a **DMA** **(Direct Memory Access) operation**.
  - **DMA** allows data transfer between memory and a device (like a network card) without involving the CPU, making it more efficient.
  - The function **returns immediately**, allowing the sender to proceed with other computations.
  - The receiver does not need to wait and can continue other tasks.

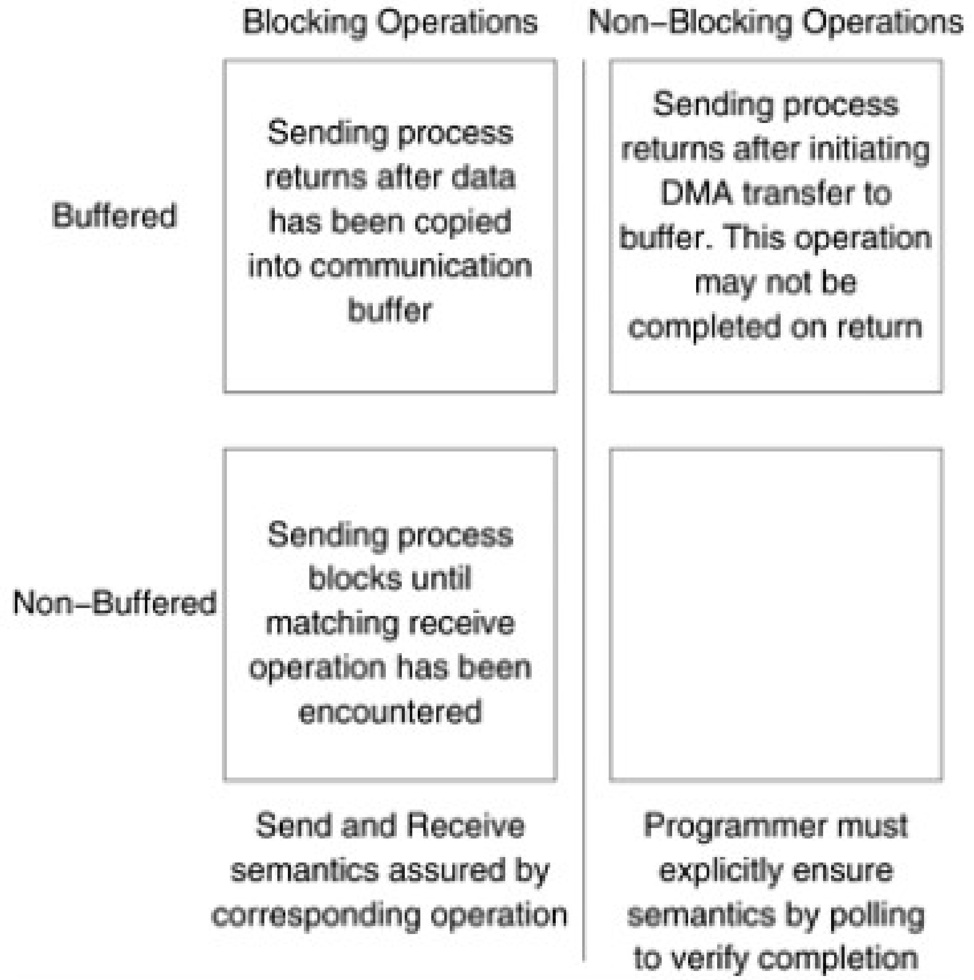
Non-Blocking Buffered Message Passing Operations

- **Non-Blocking Receive with Buffering:**
  - When the receiver initiates a non-blocking receive, it also uses a buffer.
  - The data transfer happens **in the background**, copying data from the sender’s buffer to the receiver’s memory.
  - The receiver does not need to wait and can continue other tasks

Challenges and Solutions in Non-Blocking Communication

- Since non-blocking operations return before communication is complete, modifying the involved data **too soon** can cause **errors** (e.g., sending incomplete or incorrect data).
- To prevent such issues, **check-status operations** are used to verify whether the data transfer has finished.
- If the check-status indicates the operation is incomplete, the program must **wait** before using or modifying the data.

Message Passing Operations



Space of possible protocols for send and receive operations.

Outline

- Principles of Message Passing Programming,
- The Building Blocks: Send and Receive Operations
- **MPI :Message Passing Interface**
- Topology and Embedding
- Overlapping Communication with Computation
- Collective Communication and Computation Operations.
- Groups and Communicators

MPI: the Message Passing Interface

- MPI defines a standard library for message-passing that can be used to develop portable message-passing programs using either C or Fortran.
- The MPI standard defines both the syntax as well as the semantics of a core set of library routines.
- Vendor implementations of MPI are available on almost all commercial parallel computers.
- It is possible to write fully-functional message-passing programs by using only the six routines.

MPI: the Message Passing Interface

The minimal set of MPI routines.

`MPI_Init`	Initializes MPI.
`MPI_Finalize`	Terminates MPI.
`MPI_Comm_size`	Determines the number of processes.
`MPI_Comm_rank`	Determines the label of calling process.
`MPI_Send`	Sends a message.
`MPI_Recv`	Receives a message.

Starting and Terminating the MPI Library

- MPI_Init is called prior to any calls to other MPI routines. Its purpose is to initialize the MPI environment.
- MPI_Finalize is called at the end of the computation, and it performs various clean-up tasks to terminate the MPI environment.
- The prototypes of these two functions are:

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

- MPI_Init also strips off any MPI related command-line arguments.
- All MPI routines, data-types, and constants are prefixed by "MPI_". The return code for successful completion is MPI_SUCCESS.

Communicators

- A communicator defines a *communication domain* - a set of processes that are allowed to communicate with each other.
- Information about communication domains is stored in variables of type `MPI_Comm`.
- Communicators are used as arguments to all message transfer MPI routines.
- A process can belong to many different (possibly overlapping) communication domains.
- MPI defines a default communicator called `MPI_COMM_WORLD` which includes all the processes.

Querying Information

- The `MPI_Comm_size` and `MPI_Comm_rank` functions are used to determine the number of processes and the label of the calling process, respectively.

- The calling sequences of these routines are as follows:

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

- The rank of a process is an integer that ranges from zero up to the size of the communicator minus one.

Our First MPI Program

```
#include <mpi.h>

main(int argc, char *argv[])
{
        int npes, myrank;
        MPI_Init(&argc, &argv);
        MPI_Comm_size(MPI_COMM_WORLD, &npes);
        MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
        printf("From process %d out of %d, Hello World!\n",
                myrank, npes);
        MPI_Finalize();
}
```

Sending and Receiving Messages

- The basic functions for sending and receiving messages in MPI are the `MPI_Send` and `MPI_Recv`, respectively.
- The calling sequences of these routines are as follows:

```
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, MPI_Status *status)
```

- MPI provides equivalent datatypes for all C datatypes. This is done for portability reasons.
- The datatype `MPI_BYTE` corresponds to a byte (8 bits) and `MPI_PACKED` corresponds to a collection of data items that has been created by packing non-contiguous data.
- The message-tag can take values ranging from zero up to the MPI defined constant `MPI_TAG_UB`. (it is at least 32,767)

MPI Datatypes

MPI Datatype	C Datatype
MPI_CHAR	signed char
MPI_SHORT	signed short int
MPI_INT	signed int
MPI_LONG	signed long int
MPI_UNSIGNED_CHAR	unsigned char
MPI_UNSIGNED_SHORT	unsigned short int
MPI_UNSIGNED	unsigned int
MPI_UNSIGNED_LONG	unsigned long int
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_LONG_DOUBLE	long double
MPI_BYTE	
MPI_PACKED	

Sending and Receiving Messages

- MPI allows specification of wildcard arguments for both source and tag.
- If source is set to `MPI_ANY_SOURCE`, then any process of the communication domain can be the source of the message.
- If tag is set to `MPI_ANY_TAG`, then messages with any tag are accepted.
- On the receive side, the message must be of length equal to or less than the length field specified.

Sending and Receiving Messages

- On the receiving end, the status variable can be used to get information about the MPI_Recv operation.
- The corresponding data structure contains:

```
typedef struct MPI_Status {
        int MPI_SOURCE;
        int MPI_TAG;
        int MPI_ERROR; };
```

- The MPI_Get_count function returns the precise count of data items received.

```
int MPI_Get_count (MPI_Status *status, MPI_Datatype datatype, int *count)
```

Avoiding Deadlocks

Consider:

```
int a[10], b[10], myrank;
MPI_Status status;
...
MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
if (myrank == 0) {
    MPI_Send(a, 10, MPI_INT, 1, 1, MPI_COMM_WORLD);
    MPI_Send(b, 10, MPI_INT, 1, 2, MPI_COMM_WORLD);
}
else if (myrank == 1) {
    MPI_Recv(b, 10, MPI_INT, 0, 2, MPI_COMM_WORLD);
    MPI_Recv(a, 10, MPI_INT, 0, 1, MPI_COMM_WORLD);
}
...
```

If MPI_Send is blocking, there is a deadlock.

If MPI_Send is implemented using buffering, then this code will run correctly provided that sufficient buffer space is available.

Avoiding Deadlocks

Consider the following piece of code, in which process i sends a message to process i + 1 (modulo the number of processes) and receives a message from process i - 1 (module the number of processes).

```
int a[10], b[10], npes, myrank;
MPI_Status status;
...
MPI_Comm_size(MPI_COMM_WORLD, &npes);
MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
MPI_Send(a, 10, MPI_INT, (myrank+1)%npes, 1, MPI_COMM_WORLD);
MPI_Recv(b, 10, MPI_INT, (myrank-1+npes)%npes, 1, MPI_COMM_WORLD);
...
```

Once again, we have a deadlock if MPI_Send is blocking.

Avoiding Deadlocks

We can break the circular wait to avoid deadlocks as follows:

```
int a[10], b[10], npes, myrank;
MPI_Status status;
...
MPI_Comm_size(MPI_COMM_WORLD, &npes);
MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
if (myrank%2 == 1) {
        MPI_Send(a, 10, MPI_INT, (myrank+1)%npes, 1, MPI_COMM_WORLD);
        MPI_Recv(b, 10, MPI_INT, (myrank-1+npes)%npes, 1, MPI_COMM_WORLD);
}
else {
        MPI_Recv(b, 10, MPI_INT, (myrank-1+npes)%npes, 1, MPI_COMM_WORLD);
        MPI_Send(a, 10, MPI_INT, (myrank+1)%npes, 1, MPI_COMM_WORLD);
}
...
```

Sending and Receiving Messages Simultaneously

To exchange messages, MPI provides the following function:

```
int MPI_Sendrecv (void *sendbuf, int sendcount, MPI_Datatype senddatatype, int dest, int
sendtag, void *recvbuf, int recvcount, MPI_Datatype recvdatatype, int source, int
recvtag, MPI_Comm comm, MPI_Status *status)
```

The arguments include arguments to the send and receive functions. If we wish to use the same buffer for both send and receive, we can use:

```
int MPI_Sendrecv_replace (void *buf, int count, MPI_Datatype datatype, int dest, int
sendtag, int source, int recvtag, MPI_Comm comm, MPI_Status *status)
```

Topic Overview

- Principles of Message-Passing Programming
- The Building Blocks: Send and Receive Operations
- MPI: the Message Passing Interface
- Topologies and Embedding
- Overlapping Communication with Computation
- Collective Communication and Computation Operations
- Groups and Communicators

Topologies and Embeddings

- MPI (Message Passing Interface) normally treats processes as a **1D list**, each with a unique **rank** (like an ID number).
- But in many scientific or parallel programs (like simulations), it’s more natural to think of processes as being arranged in a **grid or 3D cube**—like rows and columns in a matrix.
- This is where the idea of **topologies** comes in.

Topologies and Embeddings

- Imagine you're simulating heat spreading across a metal plate.
- It makes more sense to assign each process to a **cell in a 2D grid**, where each process communicates only with its **neighboring cells** (up, down, left, right).
- So instead of thinking in terms of simple ranks (0, 1, 2, ...), it's more intuitive to use **coordinates** like (row, col) or (x, y, z).

How Do We Map MPI Ranks to a Grid?

- Let’s say you have 8 MPI processes and want to arrange them in a **4×2 grid** (4 columns, 2 rows), or a 4×4 grid as in your example.
- MPI still gives you ranks like th

```
Ranks: 0, 1, 2, 3, 4, 5, 6, 7
```

- To place these in a 2D grid (let’s say 4 columns), you can convert a rank to (row, col) using

```
row = rank / 4    (integer division)
col = rank % 4    (modulus operator)
```

Example:

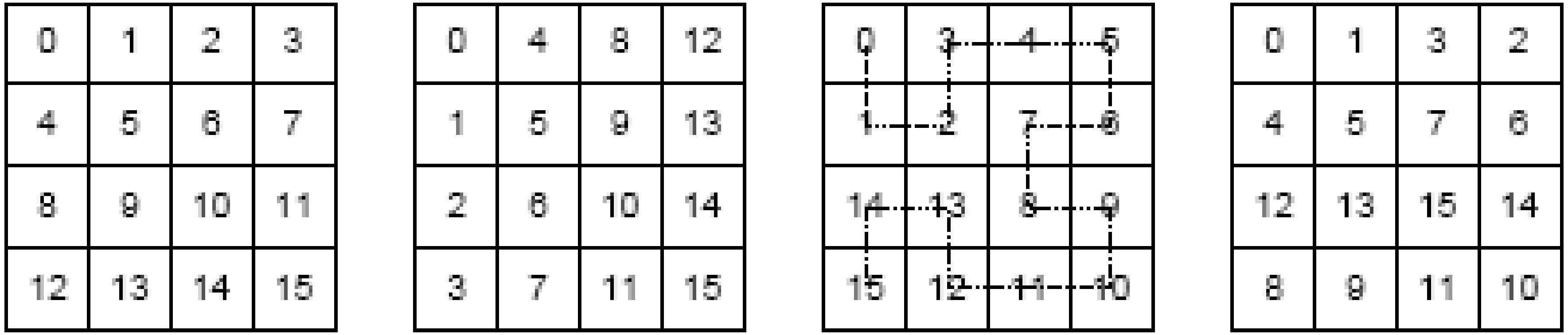
Rank 7 →

```
row = 7 / 4 = 1
```

```
col = 7 % 4 = 3
```

So rank 7 maps to **(1, 3)** in the grid.

Topologies and Embeddings



(a) Row−major mapping (b) Column−major mapping (c) Space−filling curve mapping (d) Hypercube mapping

Different ways to map a set of processes to a two-dimensional grid.

(a) and (b) show a row- and column-wise mapping of these processes,

(c) shows a mapping that follows a space-lling curve (dotted line), and

(d) shows a mapping in which neighboring processes are directly connected in a hypercube.

MPI lets you choose how you want to map ranks to grid coordinates depending on the communication pattern in your program.

Creating and Using Cartesian Topologies

- In MPI, **topologies** define how processes are connected i.e., who talks to whom.
- Instead of treating all processes as just part of one big group (MPI_COMM_WORLD), we can **organize them based on communication patterns** — like in a **line**, **grid**, or **custom graph**.
- This helps make communication **more structured**, **efficient**, and **closer to how your algorithm is designed**.

Creating and Using Cartesian Topologies

- **1. Graph Topology (Flexible but Complex)**
  - Think of each process as a **node in a graph**.
  - You can connect nodes arbitrarily — meaning **any process can be connected to any other**.
  - Useful for custom or irregular communication patterns.
  - For example: a tree, ring, or a network with random connections.
  - In MPI, you can define this using routines like MPI_Graph_create.

Creating and Using Cartesian Topologies

- **2. Cartesian Topology (Structured and Common)**
  - Most parallel algorithms use structured communication patterns like:
    - **1D line** (e.g., chain of processes)
    - **2D grid** (e.g., image processing, matrix multiplication)
    - **3D cube** (e.g., fluid simulations)
  - These are called **Cartesian topologies** (like a Cartesian coordinate system: x, y, z).
  - MPI makes it easier to work with such **regular, grid-like** layouts using specialized routines.

Cartesian Topologies

- We can create cartesian topologies using the function:

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

- This function takes the processes in the old communicator and creates a new communicator with dims dimensions.
- Each processor can now be identified in this new cartesian topology by a vector of dimension dims.

Explanation of Parameters

Parameter	Description
`comm_old`	The old communicator (e.g., `MPI_COMM_WORLD`)
`ndims`	Number of dimensions (e.g., 2 for a 2D grid, 3 for 3D)
`dims[]`	An array specifying the size of the grid in each dimension. Ex: `{3, 4}` for a 3×4 grid
`periods[]`	An array indicating whether the grid **wraps around** (like a torus). `1 = yes` , `0 = no`
`reorder`	If true (`1`), MPI may **reorder process ranks** for better performance
`comm_cart`	Output communicator with the new grid structure

Creating and Using Cartesian Topologies

- Since sending and receiving messages still require (one-dimensional) ranks, MPI provides routines to convert ranks to cartesian coordinates and vice-versa.

```
int MPI_Cart_coord (MPI_Comm comm_cart, int rank, int maxdims, int *coords)
```

  - It takes a **rank** and returns its **coordinates** in the grid.

```
int MPI_Cart_rank (MPI_Comm comm_cart, int *coords, int *rank)
```

  - This function converts (x, y, z) coordinates into a rank.

- In many parallel programs, processes **send data to their neighbors** (like to the left or right, up or down). MPI_Cart_shift helps figure out **which ranks are neighbors** in a given direction.

```
int MPI_Cart_shift (MPI_Comm comm_cart, int dir, int s_step, int *rank_source, int
  *rank_dest)
```

Topic Overview

- Principles of Message-Passing Programming
- The Building Blocks: Send and Receive Operations
- MPI: the Message Passing Interface
- Topologies and Embedding
- Overlapping Communication with Computation
- Collective Communication and Computation Operations
-

Cannon's Matrix-Matrix Multiplication Algorithm

- Cannon’s algorithm is a **parallel algorithm** to multiply two square matrices **efficiently on a 2D grid of processors**.
- It reduces **communication overhead** and improves **data reuse**, making it suitable for distributed-memory systems (like MPI clusters).
- Let’s break it down into **4 main steps**:

Cannon's Matrix-Matrix Multiplication Algorithm

- **1. Initial Alignment (Preprocessing)**
  - Before computation begins:
  - **Matrix A** blocks are **shifted left** by their row index:
    - Process at (i, j) shifts its block of A **left by i** positions.
  - **Matrix B** blocks are **shifted up** by their column index:
    - Process at (i, j) shifts its block of B **up by j** positions.
  - This ensures that all processes start with the **correct blocks** of A and B to begin computing C.

Cannon's Matrix-Matrix Multiplication Algorithm

- **2. Computation Loop**
- The main loop runs for **sqrt(p)** steps (number of grid rows or columns).
- In each step:
- Each process:
  - Multiplies its local A and B blocks
  - Adds the result to its local C block
- Then:
  - A is **shifted left** by 1 step (in the row)
  - B is **shifted up** by 1 step (in the column)
- This step rotates the blocks so that each process receives new A and B blocks to continue partial calculations.

Cannon's Matrix-Matrix Multiplication Algorithm

- **3. Final Result**
- After sqrt(p) steps:
- Each process will have computed its part of the result matrix C.

- **4. Communication Pattern**
- Cannon’s algorithm is **communication-efficient**:
- Only sqrt(p) communication steps per process
- Uses **wraparound (cyclic) shifts** — ideal for MPI topologies with MPI_Cart_create and MPI_Cart_shift

Cannon's Matrix-Matrix Multiplication Algorithm

Step	What Happens
Initial Alignment	A left-shift by row, B up-shift by column
Loop (k = 1 to √p)	Multiply A and B blocks, update C, shift A left & B up
Final	All processes hold parts of matrix C

Overlapping Communication with Computation

- `MPI_Send` and `MPI_Recv` wait (or block) until the data transfer is completed.
- Inside the main loop:
  - First, it **computes** the product of current sub-blocks of `a` and `b`.
  - Then, it **shifts** the sub-blocks (sends them to neighbors using `MPI_Sendrecv_replace`) for the next round of computation.
- `MPI_Sendrecv_replace` is **blocking**, so a process must **wait until the data is sent and received** before it can proceed.
- This causes **idle CPU time**, which is wasteful — especially since **the data being sent isn't going to change**.

Overlapping Communication with Computation

- The Optimization Idea : Why not overlap communication and computation?
- Many modern HPC systems support **asynchronous (non-blocking)** communication:
  - Communication happens in the background using **dedicated hardware** (e.g., DMA engines).
  - The CPU **doesn’t have to wait** — it can **start computing while data is in transit**.

Overlapping Communication with Computation

- So instead of:
  - MPI_Sendrecv_replace(...);   // block until done
  - compute();   // only after communication
- We can do:
  - MPI_Isend(...);   // non-blocking send
  - MPI_Irecv(...);   // non-blocking receive
  - compute();   // start computing immediately
  - MPI_Wait(...);   // wait only if needed

What Are Non-blocking Operations?

- MPI provides:
  - `MPI_Isend`: Initiates a send operation, but **returns immediately**.
  - `MPI_Irecv`: Initiates a receive operation, but **returns immediately**.

This means:

- Your program **does not wait** for the data to be fully sent or received.
- Instead, you can start other computations **while MPI handles the communication in the background** (using hardware like DMA).

What Are Non-blocking Operations?

- Why Is This Useful?
  - Because in many cases:
    - You're **sending data that won't change immediately**, or
    - You're **receiving data that you won’t use right away**.
- So, why sit idle? Let MPI handle the data transfer **while you compute** other things.

Typical flow using non-blocking MPI:

- MPI_Request request;
- MPI_Isend(data, count, datatype, dest, tag, comm, &request);
- // Do some useful computation here while data is being sent...
- MPI_Wait(&request, MPI_STATUS_IGNORE); // Ensure the send has finished

Typical flow using non-blocking MPI:

- Or for receiving:
- MPI_Request request;
- MPI_Irecv(buffer, count, datatype, source, tag, comm, &request);

- // Compute something else while data is being received...

- MPI_Wait(&request, MPI_STATUS_IGNORE); // Now it’s safe to use 'buffer'

Typical flow using non-blocking MPI:

- `MPI_Test`: Just checks **whether the communication is done**. It **doesn’t block**.
  - Returns immediately, lets you decide what to do if the operation isn’t complete yet.
- `MPI_Wait`: **Blocks** until the communication is finished.
  - You use this when you’re **ready to use the data** or **modify the buffer**.

Non-blocking MPI:

Function	Purpose	Blocking?
`MPI_Isend`	Start sending, return early	No
`MPI_Irecv`	Start receiving, return early	No
`MPI_Wait`	Wait until send/recv finishes	Yes
`MPI_Test`	Check if send/recv is done	No

Overlapping Communication with Computation

- Syntax:

```
int MPI_Isend(void *buf, int count, MPI_Datatype datatype,
int dest, int tag, MPI_Comm comm, MPI_Request *request)


int MPI_Irecv(void *buf, int count, MPI_Datatype datatype,
int source, int tag, MPI_Comm comm, MPI_Request *request)
```

```
Both return MPI_Request object

int MPI_Test(MPI_Request *request, int *flag, MPI_Status *status)
```

```
int MPI_Wait(MPI_Request *request, MPI_Status *status)
```

Overlapping Communication with Computation

- Syntax:

```
int MPI_Isend(void *buf, int count, MPI_Datatype datatype,
int dest, int tag, MPI_Comm comm, MPI_Request *request)


int MPI_Irecv(void *buf, int count, MPI_Datatype datatype,
int source, int tag, MPI_Comm comm, MPI_Request *request)


Both return MPI_Request object

int MPI_Test(MPI_Request *request, int *flag, MPI_Status *status)


int MPI_Wait(MPI_Request *request, MPI_Status *status)
```

Topic Overview

- Principles of Message-Passing Programming
- The Building Blocks: Send and Receive Operations
- MPI: the Message Passing Interface
- Topologies and Embedding
- Overlapping Communication with Computation
- Collective Communication and Computation Operations

Collective Communication and Computation Operations

- MPI provides an extensive set of functions for performing common collective communication operations.
- Each of these operations is defined over a group corresponding to the communicator.
- All processors in a communicator must call these operations.

Collective Communication Operations

- The barrier synchronization operation is performed in MPI using:

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

- The one-to-all broadcast operation is:

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

- The all-to-one reduction operation is:

```
int MPI_Reduce(void *sendbuf, void *recvbuf, int count,
MPI_Datatype datatype, MPI_Op op, int target, MPI_Comm comm)
```

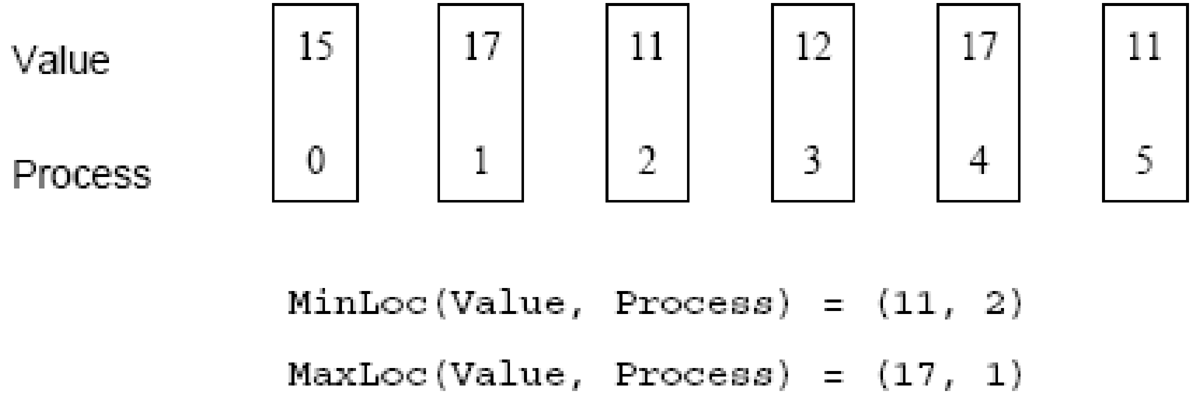
Predefined Reduction Operations

Operation	Meaning	Datatypes
`MPI_MAX`	Maximum	C integers and floating point
`MPI_MIN`	Minimum	C integers and floating point
`MPI_SUM`	Sum	C integers and floating point
`MPI_PROD`	Product	C integers and floating point
`MPI_LAND`	Logical AND	C integers
`MPI_BAND`	Bit-wise AND	C integers and byte
`MPI_LOR`	Logical OR	C integers
`MPI_BOR`	Bit-wise OR	C integers and byte
`MPI_LXOR`	Logical XOR	C integers
`MPI_BXOR`	Bit-wise XOR	C integers and byte
`MPI_MAXLOC`	max-min value-location	Data-pairs
`MPI_MINLOC`	min-min value-location	Data-pairs

Collective Communication Operations

- The operation `MPI_MAXLOC` combines pairs of values (v_i, l_i) and returns the pair (v, l) such that v is the maximum among all v_i 's and l is the corresponding l_i (if there are more than one, it is the smallest among all these l_i 's).

- `MPI_MINLOC` does the same, except for minimum value of v_i.



An example use of the `MPI_MINLOC` and `MPI_MAXLOC` operators.

Collective Communication Operations

MPI datatypes for data-pairs used with the `MPI_MAXLOC` and `MPI_MINLOC` reduction operations.

MPI Datatype	C Datatype
`MPI_2INT`	pair of ints
`MPI_SHORT_INT`	short and int
`MPI_LONG_INT`	long and int
`MPI_LONG_DOUBLE_INT`	long double and int
`MPI_FLOAT_INT`	float and int
`MPI_DOUBLE_INT`	double and int

Collective Communication Operations

- If the result of the reduction operation is needed by all processes, MPI provides:

```
int MPI_Allreduce(void *sendbuf, void *recvbuf, int count, MPI_Datatype
datatype, MPI_Op op, MPI_Comm comm)
```

- To compute prefix-sums, MPI provides:

```
int MPI_Scan(void *sendbuf, void *recvbuf, int count, MPI_Datatype
datatype, MPI_Op op, MPI_Comm comm)
```

Collective Communication Operations

- The gather operation is performed in MPI using:

```
int MPI_Gather(void *sendbuf, int sendcount,
               MPI_Datatype senddatatype, void *recvbuf,
               int recvcount, MPI_Datatype recvdatatype,
               int target, MPI_Comm comm)
```

- MPI also provides the MPI_Allgather function in which the data are gathered at all the processes.

```
int MPI_Allgather(void *sendbuf, int sendcount, MPI_Datatype senddatatype, void
*recvbuf, int recvcount, MPI_Datatype recvdatatype, MPI_Comm comm)
```

- The corresponding scatter operation is:

```
int MPI_Scatter(void *sendbuf, int sendcount,
               MPI_Datatype senddatatype, void *recvbuf,
               int recvcount, MPI_Datatype recvdatatype,
               int source, MPI_Comm comm)
```

Collective Communication Operations

- The all-to-all personalized communication operation is performed by:

```
int MPI_Alltoall(void *sendbuf, int sendcount,
MPI_Datatype senddatatype, void *recvbuf,
            int recvcount, MPI_Datatype recvdatatype,
MPI_Comm comm)
```

- Using this core set of collective operations, a number of programs can be greatly simplified.

Topic Overview

- Principles of Message-Passing Programming
- The Building Blocks: Send and Receive Operations
- MPI: the Message Passing Interface
- Topologies and Embedding
- Overlapping Communication with Computation
- Collective Communication and Computation Operations
- Groups and Communicators

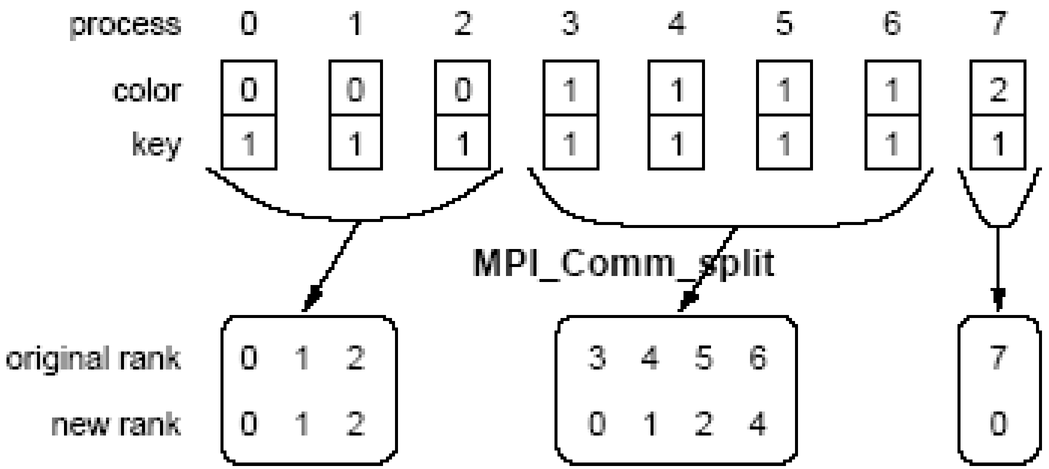
Groups and Communicators

- In many parallel algorithms, communication operations need to be restricted to certain subsets of processes.
- MPI provides mechanisms for partitioning the group of processes that belong to a communicator into subgroups each corresponding to a different communicator.
- The simplest such mechanism is:

```
int MPI_Comm_split(MPI_Comm comm, int color, int key,
MPI_Comm *newcomm)
```

- This operation groups processors by color and sorts resulting groups on the key.

Groups and Communicators



Using `MPI_Comm_split` to split a group of processes in a communicator into subgroups.

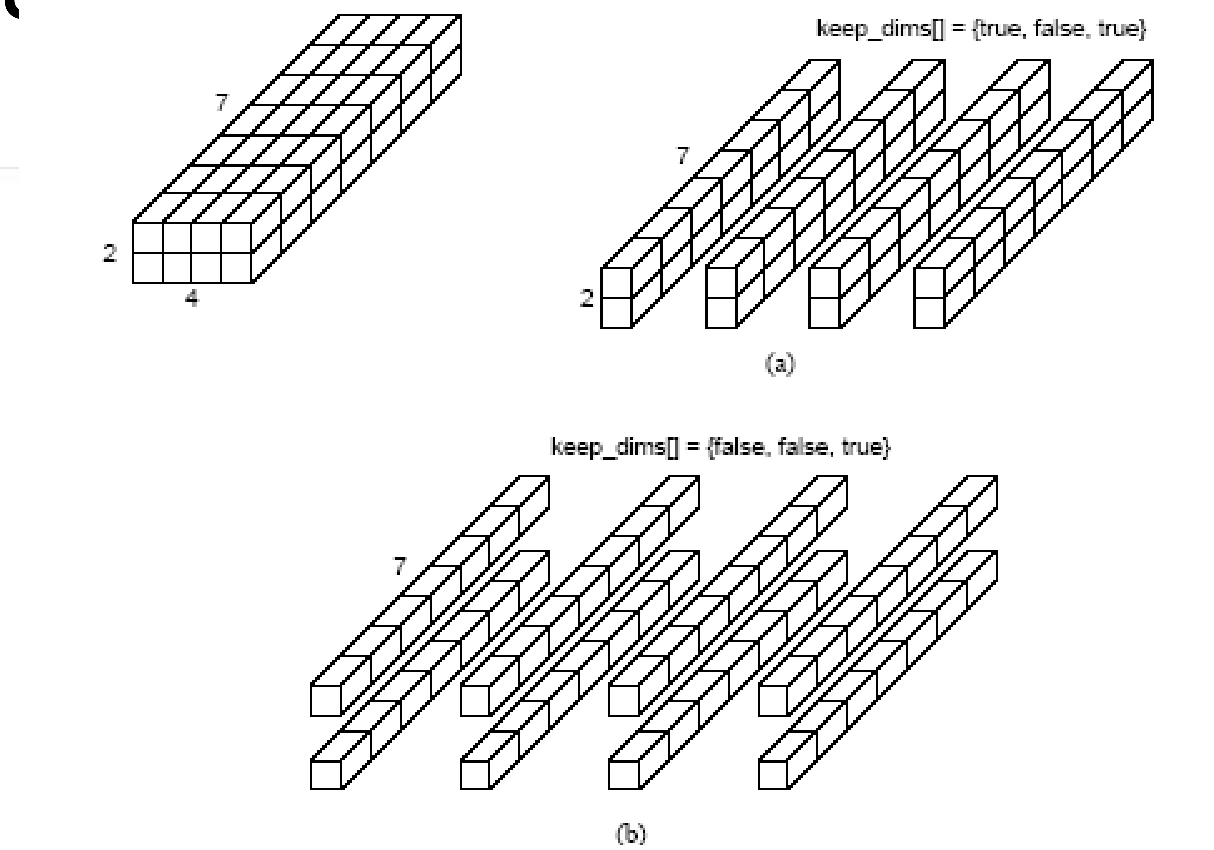
Groups and Communicators

- In many parallel algorithms, processes are arranged in a virtual grid, and in different steps of the algorithm, communication needs to be restricted to a different subset of the grid.
- MPI provides a convenient way to partition a Cartesian topology to form lower-dimensional grids:

```
int MPI_Cart_sub(MPI_Comm comm_cart, int *keep_dims,
MPI_Comm *comm_subcart)
```

- If `keep_dims[i]` is true (non-zero value in C) then the `i`th dimension is retained in the new sub-topology.
- The coordinate of a process in a sub-topology created by `MPI_Cart_sub` can be obtained from its coordinate in the original topology by disregarding the coordinates that correspond to the dimensions that were not retained.

Groups and Communicators



Splitting a Cartesian topology of size 2 x 4 x 7 into (a) four subgroups of size 2 x 1 x 7, and (b) eight subgroups of size 1 x 1 x 7.