

```
sudo apt install mpich
```

```
mpicc mpi_hello_world.c -o hello-world  
mpirun -np 5 ./hello-world
```

Programming using the Message-Passing Paradigm

Dr. Bheemappa H

Distributed memory systems

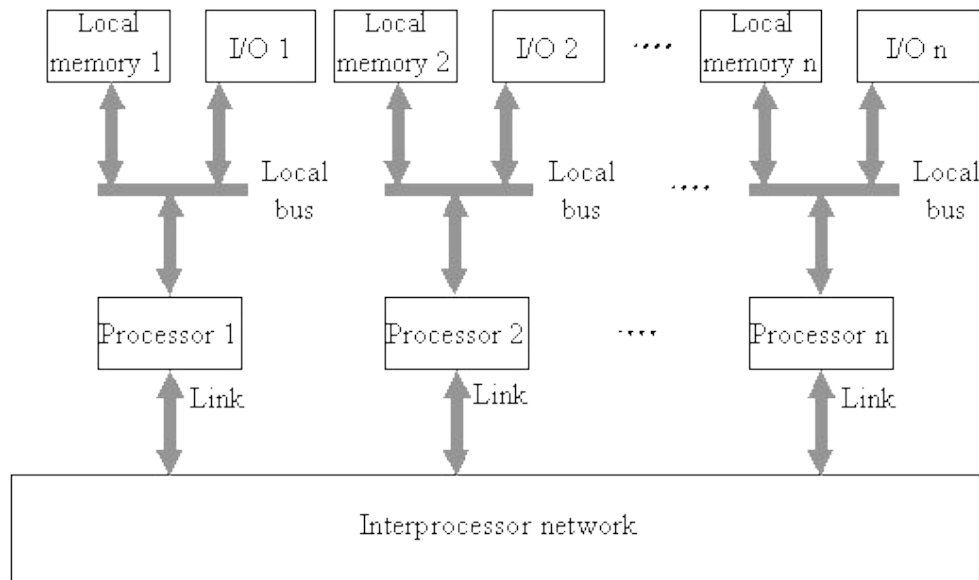


Fig: A multiprocessor system with a distributed memory (loosely coupled system)¹

Distributed memory

Each processor has its own private memory.

Computational tasks can only operate on local data,

if remote data is required, the computational task must communicate with one or more remote processors. Communication through the **message passing**.

¹<https://edux.pjwstk.edu.pl/mat/264/lec/index119.html>

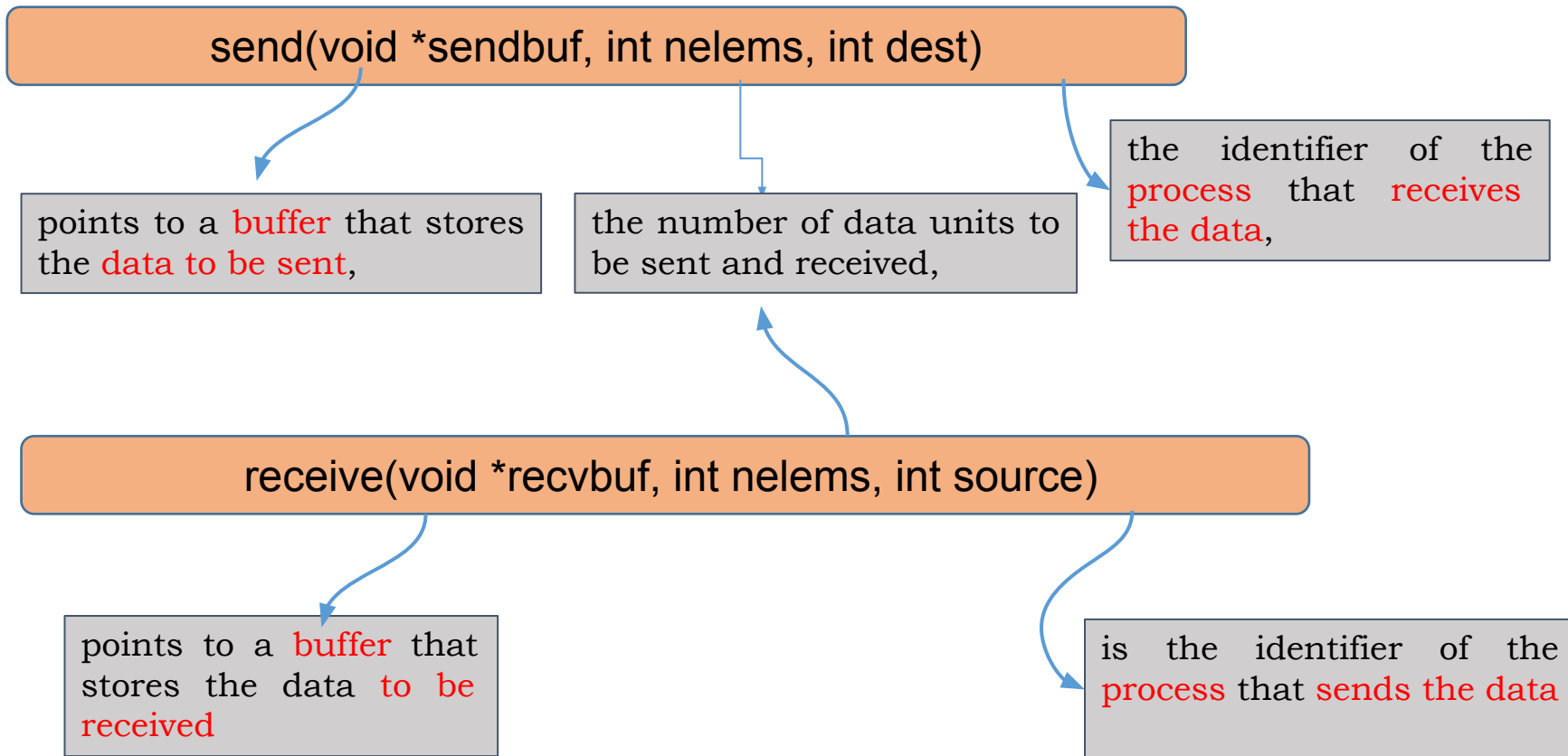
Programming using the Message-Passing Paradigm

□ Principles of Message-Passing Programming

- Message-passing paradigm consists of p processes, each with its own exclusive address space
- All interactions (read-only or read/write) require cooperation of two processes.
- The programmer is fully aware of all the costs of **non-local interactions** by Two-way interactions.
- Message-passing programs are often **written using the asynchronous or loosely synchronous paradigms**.
 - In the asynchronous paradigm, **all concurrent tasks execute asynchronously**.
 - loosely synchronous :
 - **Tasks or subsets of tasks synchronize to perform interactions.**
 - Between these **interactions**, tasks execute **completely asynchronously**.

Programming using the Message-Passing Paradigm ➤ Building Blocks

□ Send and Receive Operations



Programming using the Message-Passing Paradigm > Building Blocks

Send and Receive Operations

P0

```
a = 100;  
send(&a, 1, 1);  
a = 0;
```

P1

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

Blocking Message Passing Operations

- Blocking Non-Buffered Send/Receive
- Blocking Buffered Send/Receive

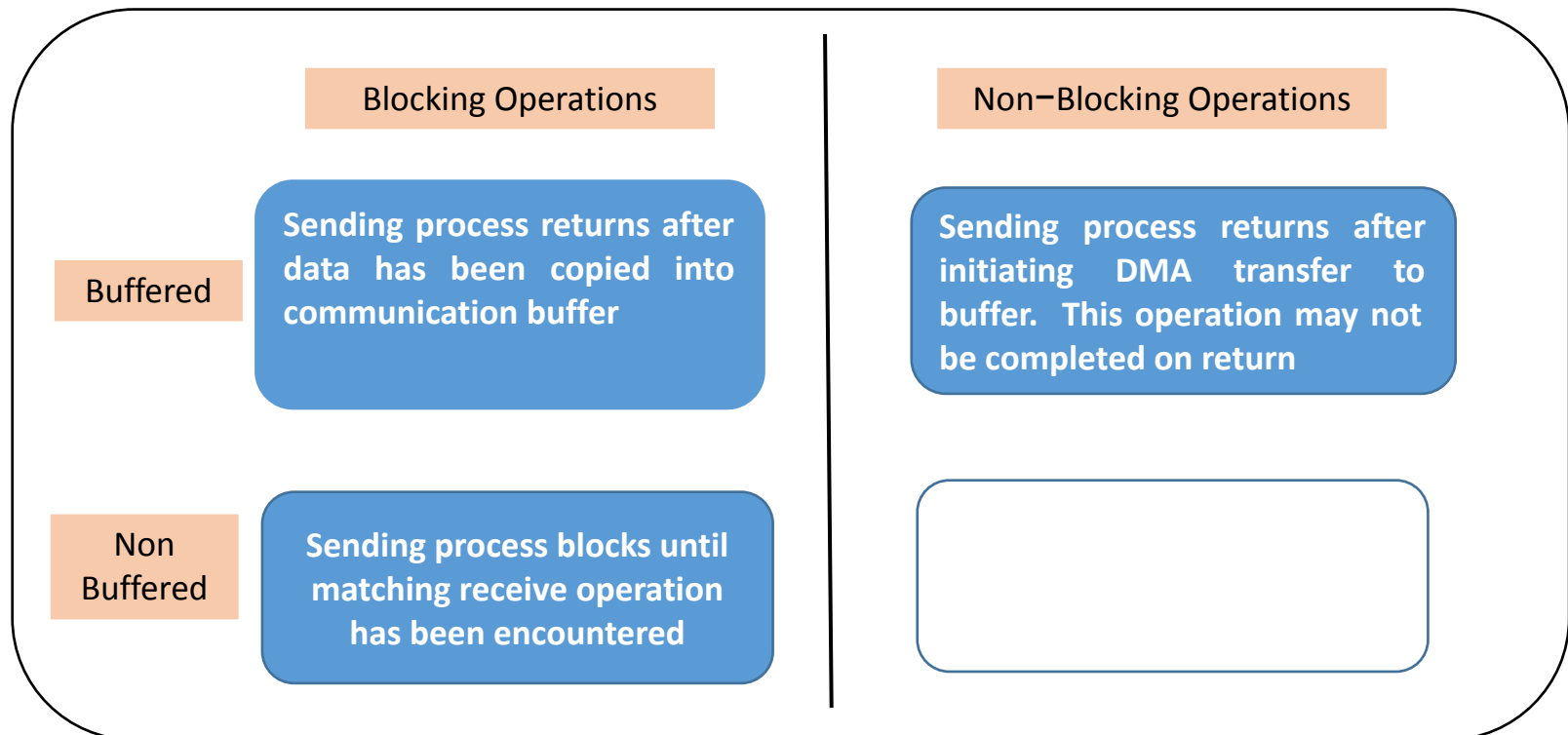
Non-Blocking Message Passing Operations

Blocking Non-Buffered Send/Receive

- Send **operation does not return until(Block)** the matching receive has been encountered at the receiving process.
- Process involves a handshake between the sending and receiving processes
 - The sending process sends a request to communicate to the receiving process.
 - When the receiving process encounters the target receive, it responds to the request.
 - The sending process upon receiving this response initiates a transfer operation

Blocking Send and Receive Protocols

Possible protocols for send and receive operations.



- 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

Programming using the Message-Passing Paradigm > MPI

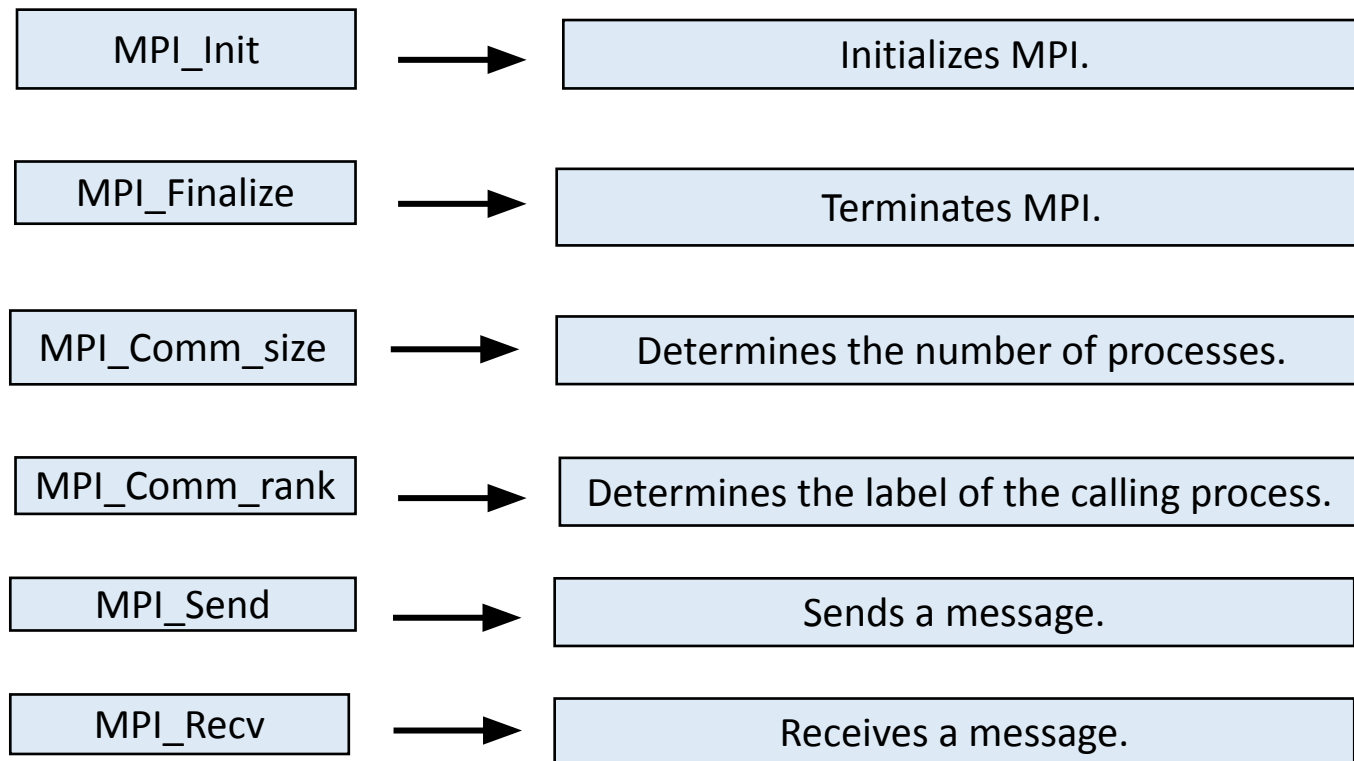
MPI: the Message Passing Interface

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

- MPI defines a standard library for message-passing.
 - Using either C or Fortran
- The MPI standard defines
 - Syntax
 - Semantics of a core set of library routines
- The **minimal set of MPI routines** can be used write to fully-functional message-passing programs
- MPI Features
 - Communicator information (com. domain).
 - Point to point communication.
 - Collective communication, Topology support, „Error handling.

Programming using the Message-Passing Paradigm > MPI

The minimal set of MPI routines



Programming using the Message-Passing Paradigm > MPI

MPI Functions: Initialization

- **MPI_Init** initializes the MPI environment
 - Must be called once by all processes. MPI_SUCCESS (if successful).
- **MPI_Finalize** performs clean-up tasks, no MPI calls after that (not even MPI_Init)
- **MPI_Init, MPI_Finalize** must be called by all processes.

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

```
MPI_Finalize()
```

Programming using the Message-Passing Paradigm > MPI

MPI Functions: Communicator

- Concept of communication domain-
 - Set of processes allowed to communicate with each other.
 - Processes may belong to **different communicators**
- **MPI_COMM_WORLD** default for all processes involved.
 - **Rank** is an `int[0..comm_size-1]`
- Processes calling `MPI_Comm_size`, `MPI_Comm_rank` functions must belong to the appropriate communicator otherwise error

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

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

Programming using the Message-Passing Paradigm > MPI

Hello World!

```
#include <mpi.h>
int 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();
    return 0;
}
```

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)

Programming using the Message-Passing Paradigm > MPI

Topologies and Embedding

- MPI views the processes as being arranged in a one-dimensional topology and uses a linear ordering to number the processes.
- Both the computation and the set of interacting processes are naturally identified by their coordinates in that topology.
- MPI does not provide the programmer any control over these mappings.
- An MPI process with rank *rank* corresponds to process (*row*, *col*) in the grid such that
 - ***row = rank/4 and col = rank%4***
- As an illustration, the process with **rank 7** is mapped to process (1, 3) in the grid.

Programming using the Message-Passing Paradigm > MPI

0	1	2	3
4	5	6	7
8	9	10	11
12	13	14	15

(a) Row-major mapping

0	4	8	12
1	5	9	13
2	6	10	14
3	7	11	15

(b) Column-major mapping

0	3	4	5
1	2	7	6
14	13	8	9
15	12	11	10

(c) Space-filling curve mapping

0	1	3	2
4	5	7	6
12	13	15	14
8	9	11	10

(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-filling curve (dotted line), and (d) shows a mapping in which neighboring processes are directly connected in a hypercube.

- MPI topologies are virtual – no relation to the physical structure of the computer
- Data mapping “more natural” only to the programmer.
- Usually no performance benefits – But code becomes more readable

Programming using the Message-Passing Paradigm > MPI

Topologies and Embedding

- MPI provides a set of routines that allows the programmer to arrange the processes in different topologies.
 - Both the computation and the set of interacting processes are naturally identified by their coordinates in that topology.
 - It is up to the MPI library to find the most appropriate mapping that reduces the cost of sending and receiving messages.
- Each node in the graph corresponds to a **process** and two nodes are connected if they **communicate with each other**. Graphs of processes can be used to specify any desired topology.
- Most commonly used topologies in message-passing programs are one-, two-, or higher-dimensional grid and its referred as “**Cartesian topologies**”.

Topologies and Embedding

- Creating a topology produces a new communicator.
- Topology Types
 - Cartesian Topologies –
 - Connected to its neighbor in a virtual grid.
 - Graph Topologies -general graphs,
- Creating a Cartesian Virtual Topology
 - New communicator with processes ordered in a Cartesian grid

Programming using the Message-Passing Paradigm > MPI

Cartesian topologies

- MPI's function for describing Cartesian topologies is called **MPI_Cart_create**.
- A group of processes that belong to the communicator **comm_old** and creates a virtual process topology.

The shape and properties of the topology are specified by the arguments **ndims**, **dims**, and **periods**.

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

Topology information is attached to a new communicator

Programming using the Message-Passing Paradigm > MPI

Cartesian topologies

- MPI's function for describing Cartesian topologies is called **MPI_Cart_create**.
- A group of processes that belong to the communicator **comm_old** and creates a virtual process topology.

It Takes the coordinates of the process as argument in the coords array and returns its rank in rank.

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

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

The MPI_Cart_coords takes the rank of the process rank and returns its Cartesian coordinates in the array coords

1. Process coordinates in a Cartesian structure begin their numbering at 0.
Row-major numbering is always used for the processes in a cartesian structure.

This means that, for example, the relation between group rank and coordinates for **four processes** in a (2×2) **grid is as follows.**

coord (0,0):	rank 0
coord (0,1):	rank 1
coord (1,0):	rank 2
coord (1,1):	rank 3

Overlapping Communication with Computation

- In order to overlap communication with computation, MPI provides a pair of functions for performing non-blocking send and receive operations.

```
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)
```

- These operations return before the operations have been completed. Function `MPI_Test` tests **whether or not the non-blocking send or receive operation identified by its request has finished**.

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

- `MPI_Wait` waits for the operation to complete.

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

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)
```

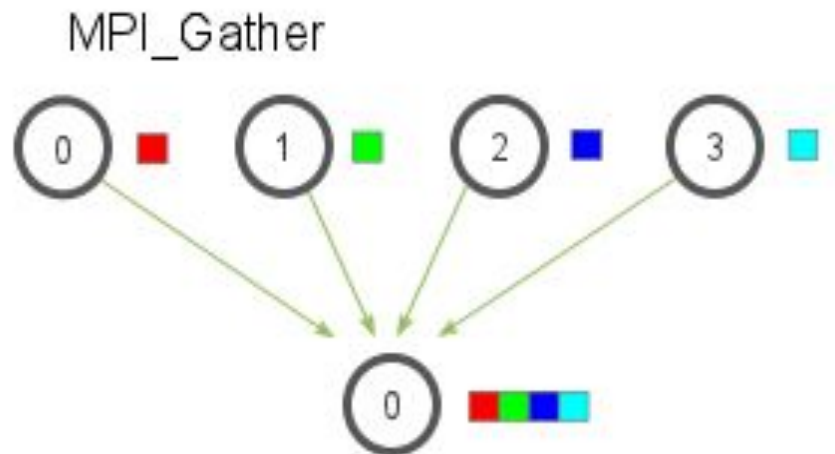
Predefined Reduction Operations

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 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)
```

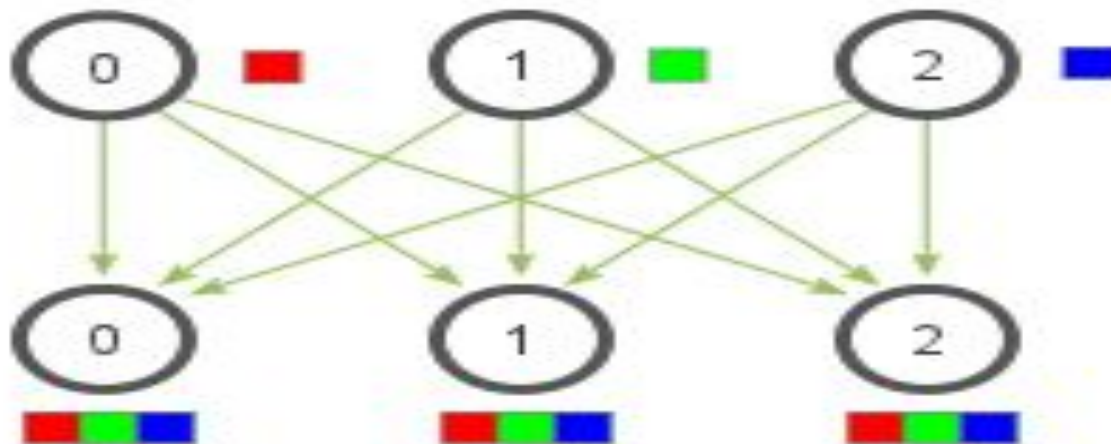
The meanings of the various parameters are similar to those for MPI_Gather ;

however, each process must now supply a recvbuf array that will store the gathered data.

In addition to the above versions of the gather operation, in which the sizes of the arrays sent by each process are the same, MPI also provides versions in which the size of the arrays can be different.

MPI refers to these operations as the vector variants. The vector variants of the MPI_Gather and MPI_Allgather operations are provided by the functions MPI_Gatherv and MPI_Allgatherv , respectively.

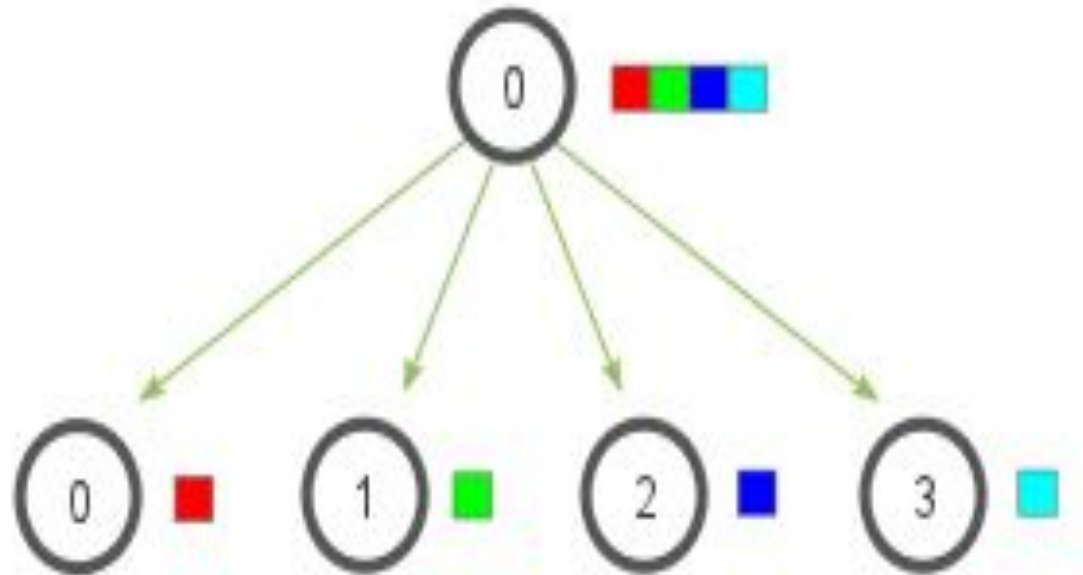
MPI_Allgather



- 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)
```

MPI_Scatter



```
int MPI_Scatterv(void *sendbuf, int *sendcounts, int *displs, MPI_Datatype  
senddatatype, void *recvbuf, int recvcount, MPI_Datatype recvdatatype, int source,  
MPI_Comm comm)
```

MPI_Scatterv , that allows different amounts of data to be sent to different processes.

As we can see, the parameter sendcount has been replaced by the array sendcounts that determines the number of elements to be sent to each process.

In particular, the target process sends sendcounts[i] elements to process i .
Also, the array displs is used to determine where in sendbuf these elements will be sent from.

In particular, if sendbuf is of the same type is senddatatype , the data sent to process i start at location displs[i] of array sendbuf .

Both the sendcounts and displs arrays are of size equal to the number of processes in the communicator.

All-to-All: Each process sends a different portion of the sendbuf array to each other process, including itself.

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

Program 6.4 Row-wise Matrix-Vector Multiplication

```
1  RowMatrixVectorMultiply(int n, double *a, double *b, double *x,
2                          MPI_Comm comm)
3  {
4      int i, j;
5      int nlocal;      /* Number of locally stored rows of A */
6      double *fb;      /* Will point to a buffer that stores the entire vector b
7                          locally */
8      int npes, myrank;
9      MPI_Status status;
10
11     /* Get information about the communicator */
12     MPI_Comm_size(comm, &npes);
13     MPI_Comm_rank(comm, &myrank);
```

```

14  /* Allocate the memory that will store the entire vector b */
15  fb = (double *)malloc(n*sizeof(double));
16
17  nlocal = n/npes;
18
19  /* Gather the entire vector b on each processor using MPI's ALLGATHER operat
20  MPI_Allgather(b, nlocal, MPI_DOUBLE, fb, nlocal, MPI_DOUBLE,
21  comm);
22
23  /* Perform the matrix-vector multiplication involving the locally stored sub
24  for (i=0; i<nlocal; i++) {
25      x[i] = 0.0;
26      for (j=0; j<n; j++)
27          x[i] += a[i*n+j]*fb[j];
28  }
29
30  free(fb);
31  }

```


Program 6.5 Column-wise Matrix-Vector Multiplication

```
1  ColMatrixVectorMultiply(int n, double *a, double *b, double *x,  
2                          MPI_Comm comm)  
3  {  
4      int i, j;  
5      int nlocal;  
6      double *px;  
7      double *fx;  
8      int npes, myrank;  
9      MPI_Status status;  
10  
11     /* Get identity and size information from the communicator */  
12     MPI_Comm_size(comm, &npes);  
13     MPI_Comm_rank(comm, &myrank);
```

```
4
5     nlocal = n/npes;
6
7     /* Allocate memory for arrays storing intermediate results. */
8     px = (double *)malloc(n*sizeof(double));
9     fx = (double *)malloc(n*sizeof(double));
10
11    /* Compute the partial-dot products that correspond to the local columns of
12    for (i=0; i<n; i++) {
13        px[i] = 0.0;
14        for (j=0; j<nlocal; j++)
15            px[i] += a[i*nlocal+j]*b[j];
16    }
17
18    /* Sum-up the results by performing an element-wise reduction operation */
19    MPI_Reduce(px, fx, n, MPI_DOUBLE, MPI_SUM, 0, comm);
20
21    /* Redistribute fx in a fashion similar to that of vector b */
22    MPI_Scatter(fx, nlocal, MPI_DOUBLE, x, nlocal, MPI_DOUBLE, 0,
23        comm);
24
25    free(px); free(fx);
26 }
```

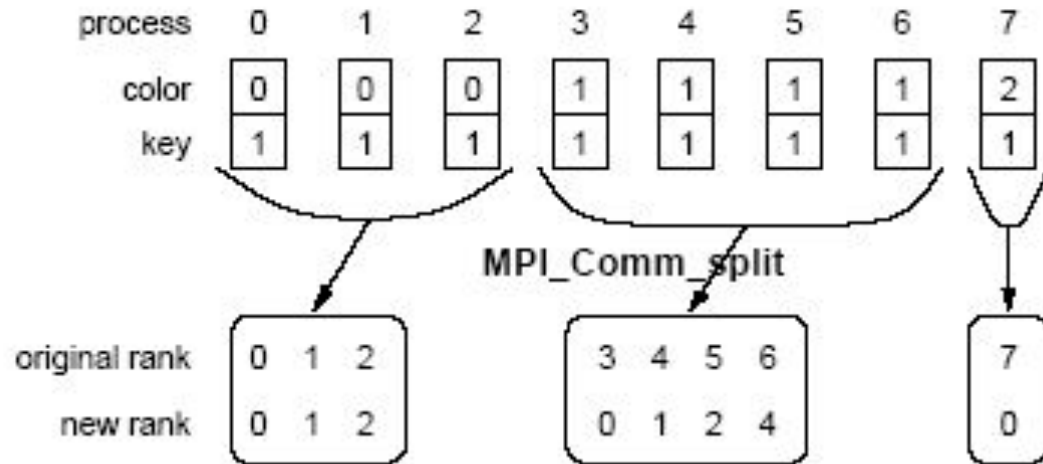
Groups and Communicators

- 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.

Introduction: GPUs as Parallel Computers

Dr. Bheemappa H

Introduction to GPU



GPUs as Parallel Computers,

- Microprocessors based on a single central processing unit (CPU)
 - Giga (billion)floating-point operations per second (GFLOPS)
- Most software developers have relied on the advances in hardware to increase the speed of their applications.
- Due to **energy consumption and heat-dissipation** issues that have **limited the increase of the clock frequency** and the level of productive activities.
- **Processor cores.**
 - Vast majority of software applications are written as sequential programs, as described by von Neumann.
 - A sequential program will only run on one of the processor cores, which will not become significantly faster than those in use today.
 - concurrency revolution
 - multiple threads of execution cooperate to complete the work faster

Introduction to GPU



GPUs as Parallel Computers,

- few elite applications can justify the use of these expensive computers, thus limiting the practice of parallel programming to a small number of application developers.
- Many-core trajectory focuses more on the execution throughput of parallel applications.
- The many-cores began as a large number of much smaller cores, and, once again, the number of cores doubles with each generation.
 - Example : NVIDIA GeForce GTX 280 graphics processing unit (GPU) with 240 cores.
 - Each of which is a heavily multithreaded, in-order, single-instruction issue processor that shares its control and instruction cache with seven other cores

Introduction to GPU



GPUs as Parallel Computers,

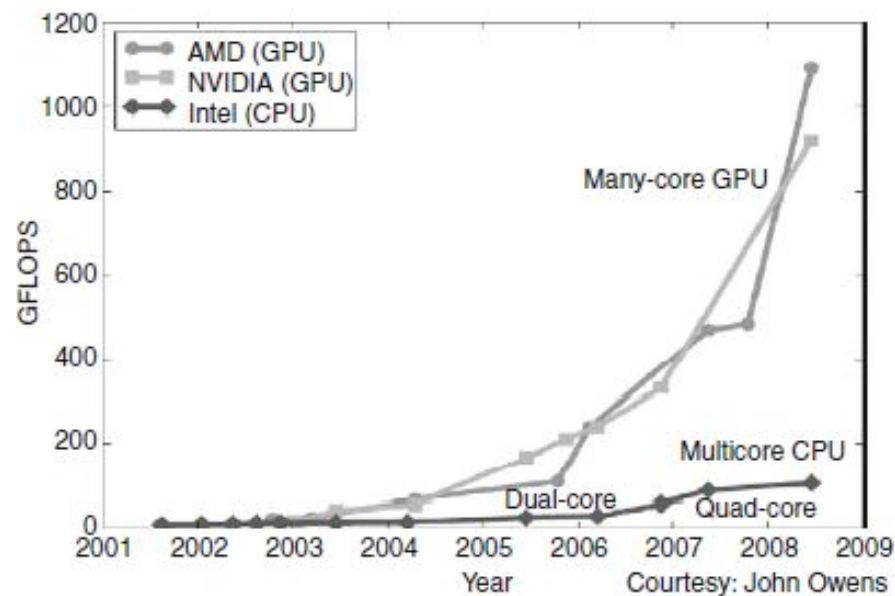


FIGURE 1.1

Enlarging performance gap between GPUs and CPUs.



Why there is such a large performance gap between many-core GPUs and general-purpose multicore CPUs?

- Differences in the fundamental design philosophies between the two types of processors
- The design of a CPU is optimized for sequential code performance.
- Control logic to allow instructions from a single thread of execution to execute in parallel while maintaining the appearance of sequential execution
- Control logic nor cache memories contribute to the peak calculation speed.
- Deliver strong sequential code performance on multicore.
- Memory bandwidth is another important issue.

GPUs as Parallel Computers,



- Graphics chips have been operating at approximately 10 times the bandwidth of contemporaneously available CPU chips

GeForce 8800 GTX

- 85 GB/s : in and out

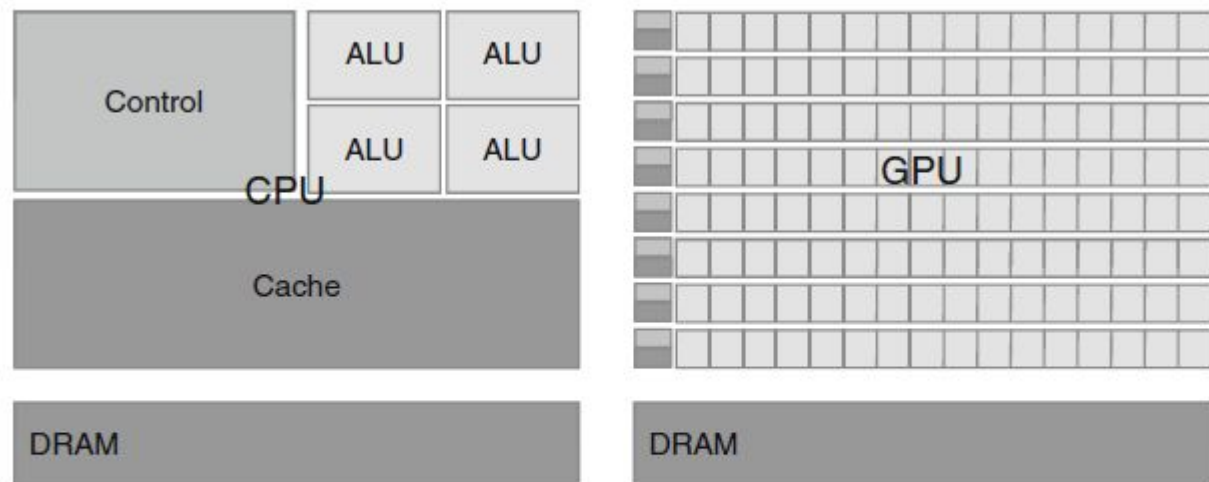


FIGURE 1.2

CPU and GPU have fundamentally different design philosophies.