

3: Fundamental Design Issues in HPC

Kusum Lata Dhiman

Assistant Professor

Computer Science & Engineering

Syllabus

- Programming Using the Message-Passing Paradigm: Principles of Message-Passing Programming, The Building Blocks: Send and Receive Operations,
- MPI: the Message Passing Interface, Topology and Embedding, Overlapping Communication with Computation, Collective Communication and Computation Operations,
- One-Dimensional Matrix-Vector Multiplication, Single-Source Shortest-Path, Sample Sort, Groups and Communicators, Two-Dimensional Matrix-Vector Multiplication

- The logical view of a machine supporting the message-passing paradigm consists of p processes, each with its own exclusive address space.
- There are two immediate implementations of a partitioned address space.
- First, Each data element must belong to one of the partitions of the space; hence, data must be explicitly partitioned and placed.
- Second, 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.
- These two constraints, while onerous, make underlying costs very explicit to the programmer.

- 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.

- 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

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

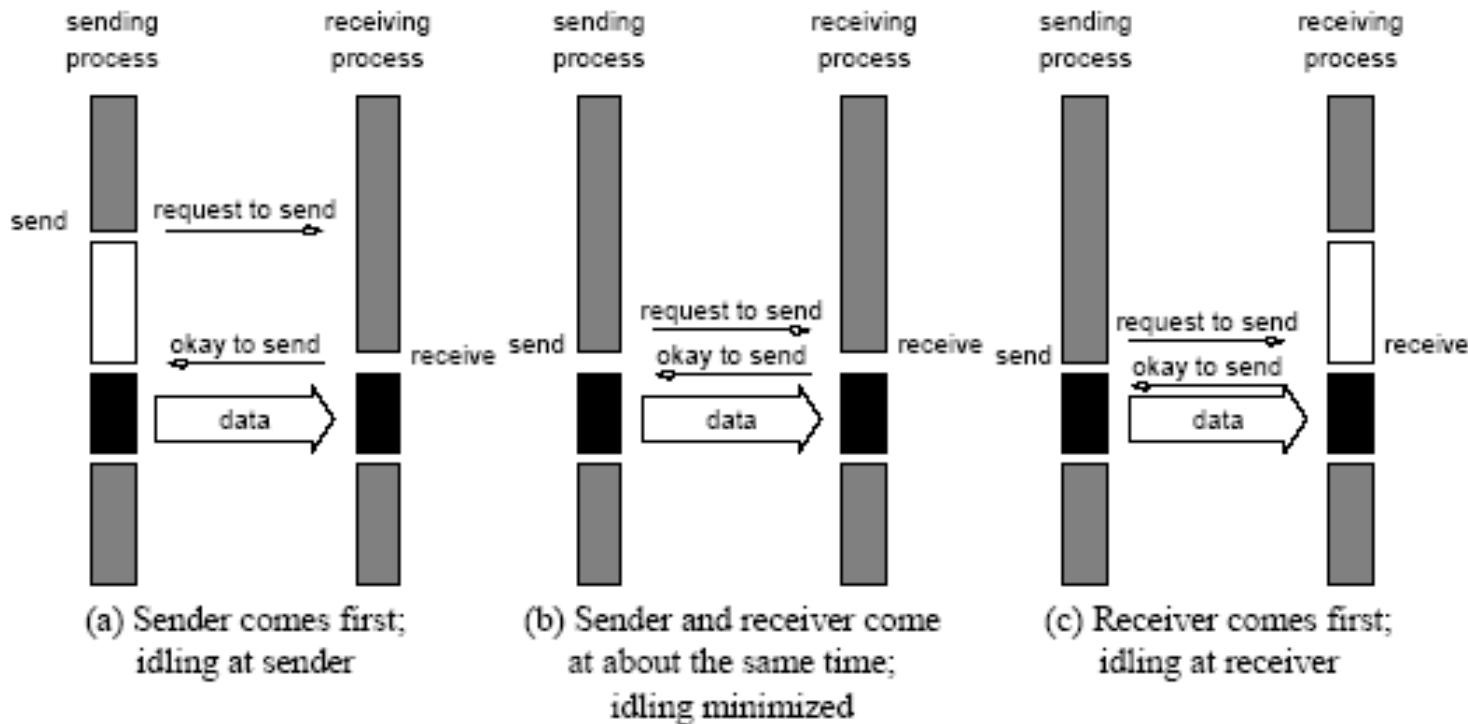
P1

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

- The semantics of the send operation require that the value received by process P1 must be 100 as opposed to 0.
- This motivates the design of the send and receive protocols.

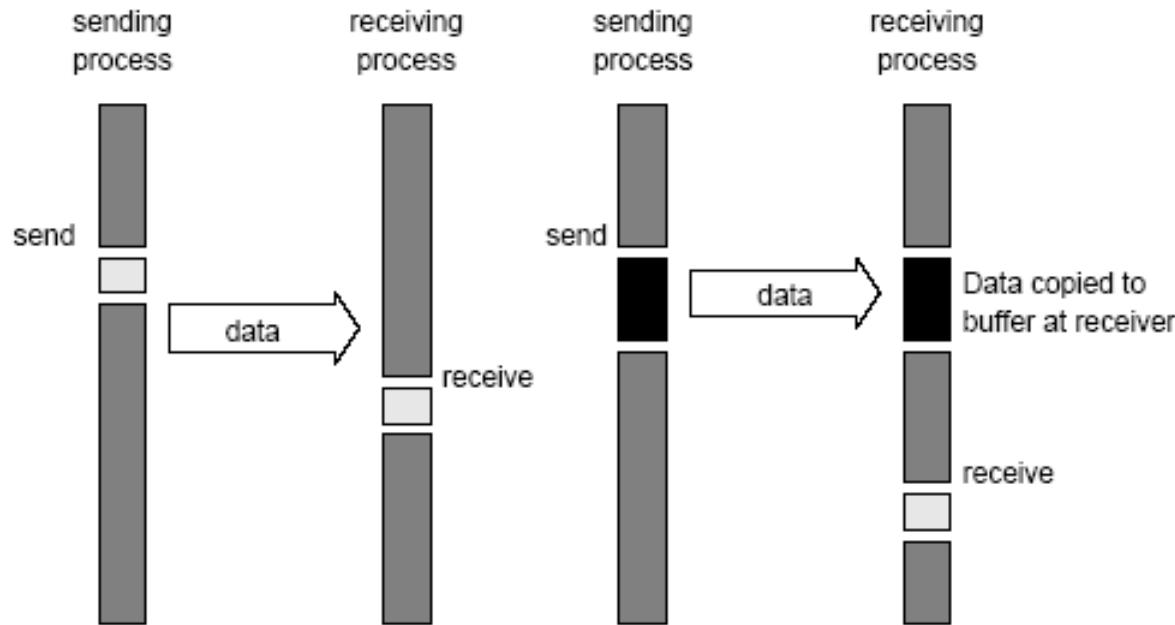
Message Passing Operations

- A simple method for forcing send/receive semantics is for the send operation to return only when it is safe to do so.
- 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.



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.

- A simple solution to the idling and deadlocking problem outlined above is to rely on buffers at the sending and receiving ends.
- The sender simply copies the data into the designated buffer and returns after the copy operation has been completed.
- The data must be buffered at the receiving end as well.
- Buffering trades off idling overhead for buffer copying overhead.



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.

Bounded buffer sizes can have significant impact on performance.

```
P0                                P1
for (i = 0; i < 1000; i++) {    for (i = 0; i < 1000; i++) {
    produce_data(&a);          receive(&a, 1, 0);
    send(&a, 1, 1);           consume_data(&a);
}                                }
```

What if consumer was much slower than producer?

Deadlocks are still possible with buffering since receive operations block.

P0

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

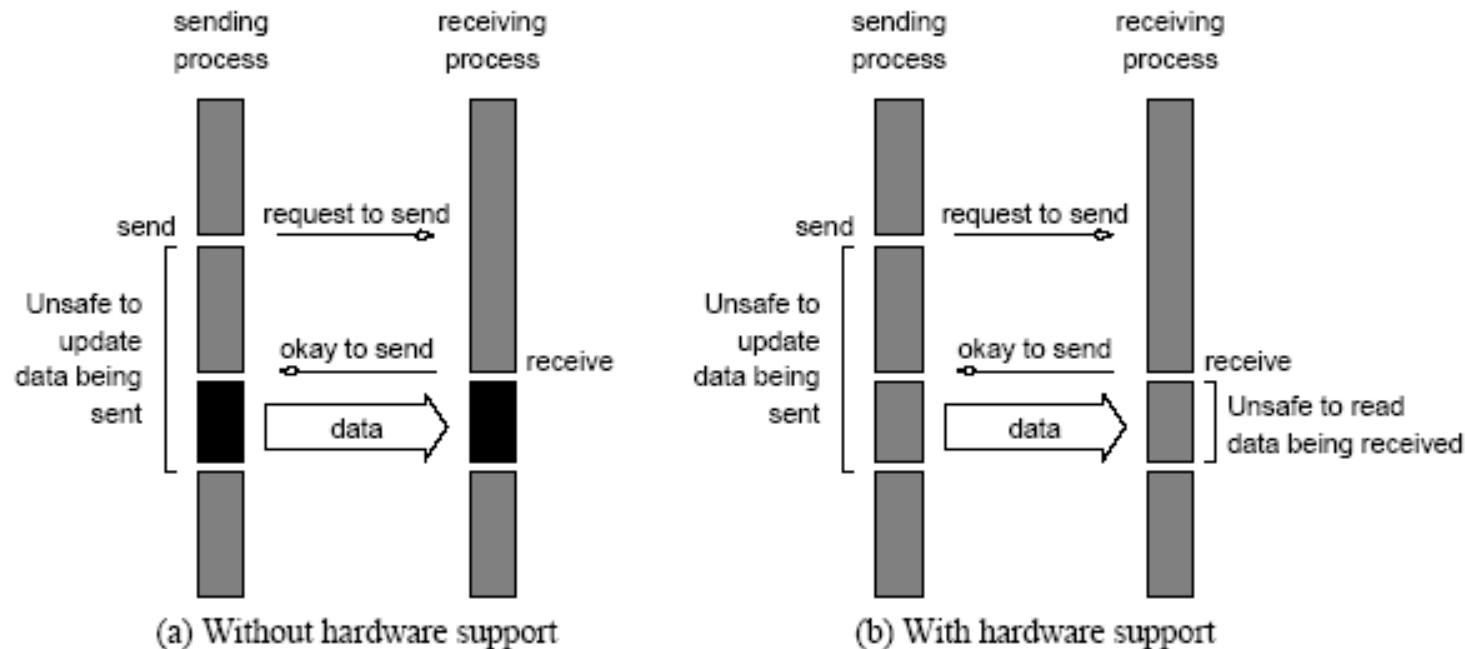
P1

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

- The programmer must ensure semantics of the send and receive.
- This class of non-blocking protocols returns from the send or receive operation before it is semantically safe to

~~check-status operation.~~

- When used correctly, these primitives are capable of overlapping communication overheads with useful computations.
- Message passing libraries typically provide both blocking and non-blocking primitives.



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

	Blocking Operations	Non-Blocking Operations
Buffered	Sending process returns after data has been copied into communication buffer	Sending process returns after initiating DMA transfer to buffer. This operation may not be completed on return
Non-Buffered	Sending process blocks until matching receive operation has been encountered	

Space of possible protocols for send and receive operations.

- 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.

The minimal set of MPI routines.

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

- `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`.

- 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.

- 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.

```
#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();
}
```

- 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`.

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	

- 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.

- 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)
```

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.

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 $i - 1$ (modulo 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.

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);  
}  
...  
...
```

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)
```

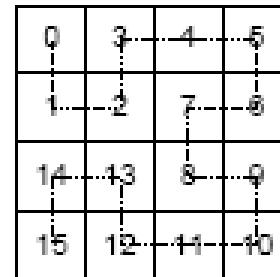
- MPI allows a programmer to organize processors into logical k -d meshes.
- The processor ids in `MPI_COMM_WORLD` can be mapped to other communicators (corresponding to higher-dimensional meshes) in many ways.
- The goodness of any such mapping is determined by the interaction pattern of the underlying program and the topology of the machine.
- MPI does not provide the programmer any control over these mappings.

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



(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.

- 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.

- 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)
```

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

- The most common operation on cartesian topologies is a shift. To determine the rank of source and destination of such shifts, MPI provides the following function:

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

- 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)
```

Using non-blocking operations remove most 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, &status, MPI_COMM_WORLD);  
    MPI_Recv(a, 10, MPI_INT, 0, 1, &status, MPI_COMM_WORLD);  
}  
...  
...
```

Replacing either the send or the receive operations with non-blocking counterparts fixes this deadlock.

- 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.

- 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)
```

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

- 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 .

Value	15	17	11	12	17	11
Process	0	1	2	3	4	5

`MinLoc(Value, Process) = (11, 2)`

`MaxLoc(Value, Process) = (17, 1)`

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

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

- 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)
```

- 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)
```

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

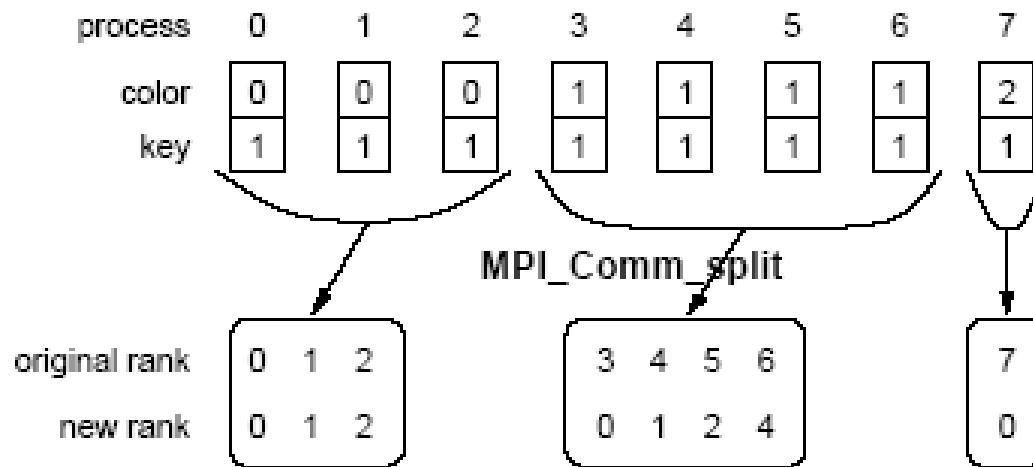
```
int MPI_Alltoall(void *sendbuf, int sendcount,  
                  MPI_Datatype senddatatype, void *recvbuf,  
                  int recvcount, MPI_Datatype recvdatatype,  
                  MPI_Comm comm)
```

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

- 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.

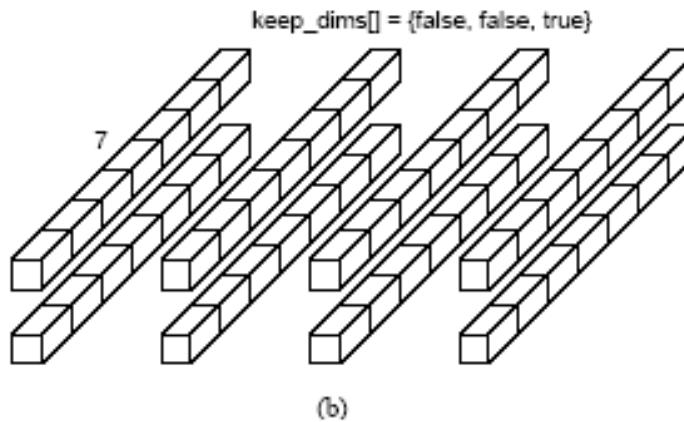
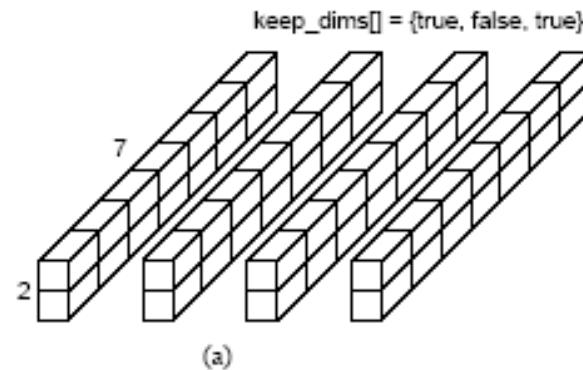
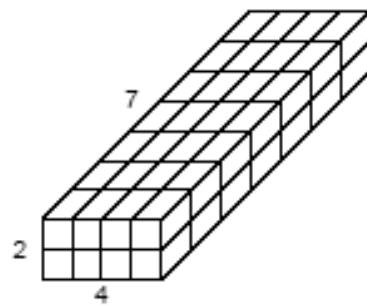


Using `MPI_Comm_split` to split a group of processes in a communicator into subgroups.

- 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.



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

- Message-passing program using collective communications.
- To multiply a dense $n \times n$ matrix A with a vector b , i.e., $x = Ab$
- One way of performing this multiplication in parallel is to have each process compute different portions of the product-vector x .
- In particular, each one of the p processes is responsible for computing n/p consecutive elements of x . This algorithm can be implemented in MPI by distributing the matrix A in a row-wise fashion, such that each process receives the n /p rows that correspond to the portion of the product-vector x it computes. Vector b is distributed in a fashion similar to x .

- MPI program that uses a row-wise distribution of matrix A . The dimension of the matrices is supplied in the parameter n , the parameters a and b point to the locally stored portions of matrix A and vector b , respectively, and the parameter x points to the local portion of the output matrix-vector product. This program assumes that n is a multiple of the number of processors.

- Message-passing program that uses collective communication operations computes the shortest paths from a source-vertex s to all the other vertices in a graph using Dijkstra's single-source shortest-path algorithm.
- The parameter n stores the total number of vertices in the graph, and the parameter $source$ stores the vertex from which we want to compute the single-source shortest path. The parameter wgt points to the locally stored portion of the weighted adjacency matrix of the graph. The parameter $lengths$ points to a vector that will store the length of the shortest paths from source to the locally stored vertices. Finally, the parameter $comm$ is the communicator to be used by the MPI routines. Note that this routine assumes that the number of vertices is a multiple of the number of processors.

- The SampleSort function takes as input the sequence of elements stored at each process and returns a pointer to an array that stores the sorted sequence as well as the number of elements in this sequence. The elements of this SampleSort function are integers and they are sorted in increasing order. The total number of elements to be sorted is specified by the parameter n and a pointer to the array that stores the local portion of these elements is specified by elmnts . On return, the parameter nsorted will store the number of elements in the returned sorted array. This routine assumes that n is a multiple of the number of processes.

- The dimension of the matrix is supplied in the parameter n , the parameters a and b point to the locally stored portions of matrix A and vector b , respectively, and the parameter x points to the local portion of the output matrix-vector product. Note that only the processes along the first column of the process grid will store b initially, and that upon return, the same set of processes will store the result x . For simplicity, the program assumes that the number of processes p is a perfect square and that n is a multiple of \sqrt{p} .



<https://paruluniversity.ac.in/>

