Design of Parallel Algorithms

Introduction to the Message Passing Interface

MPI

+

Principles of Message-Passing Programming

- The logical view of a machine supporting the message-passing paradigm consists of p processes, each with its own exclusive address space.
- 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. (Two Sided Communication Methods)
- These two constraints, while onerous, make underlying costs very explicit to the programmer.

+

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.

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:

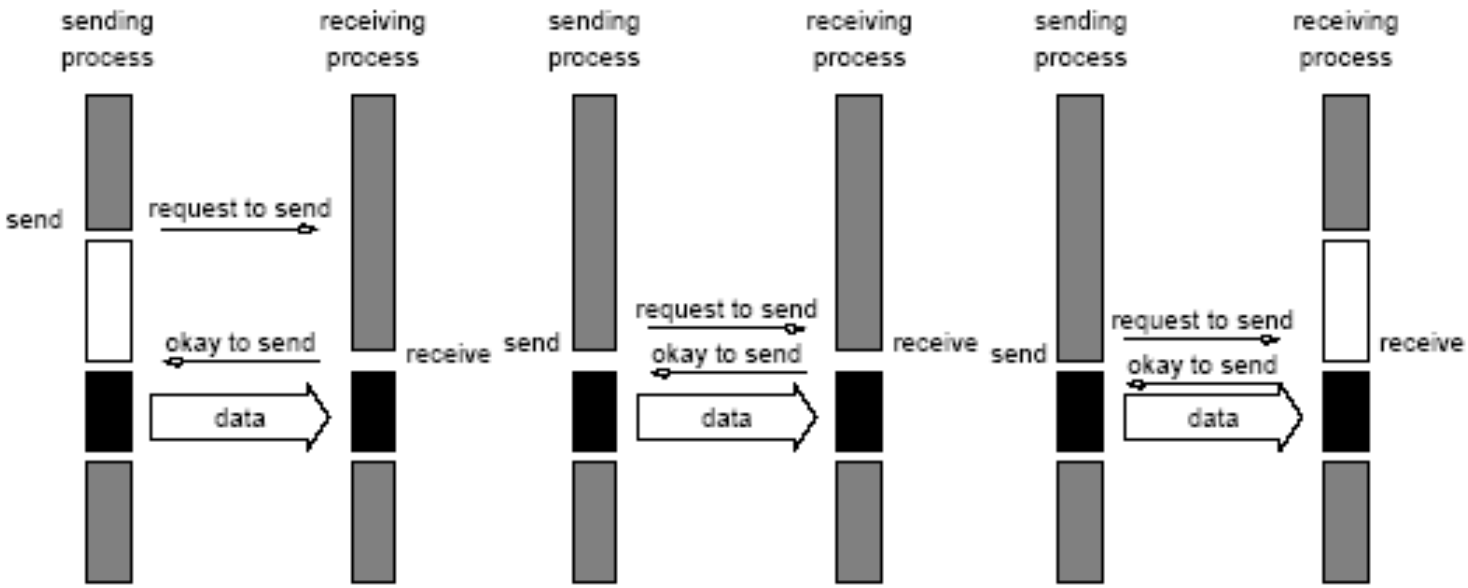
```
P0                          P1

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, not 0.
- This motivates the design of the send and receive protocols.

Non-Buffered 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.
- 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.

Non-Buffered Blocking Message Passing Operations



(a) Sender comes first; idling at sender

(b) Sender and receiver come at about the same time; idling minimized

(c) Receiver comes first; idling at receiver

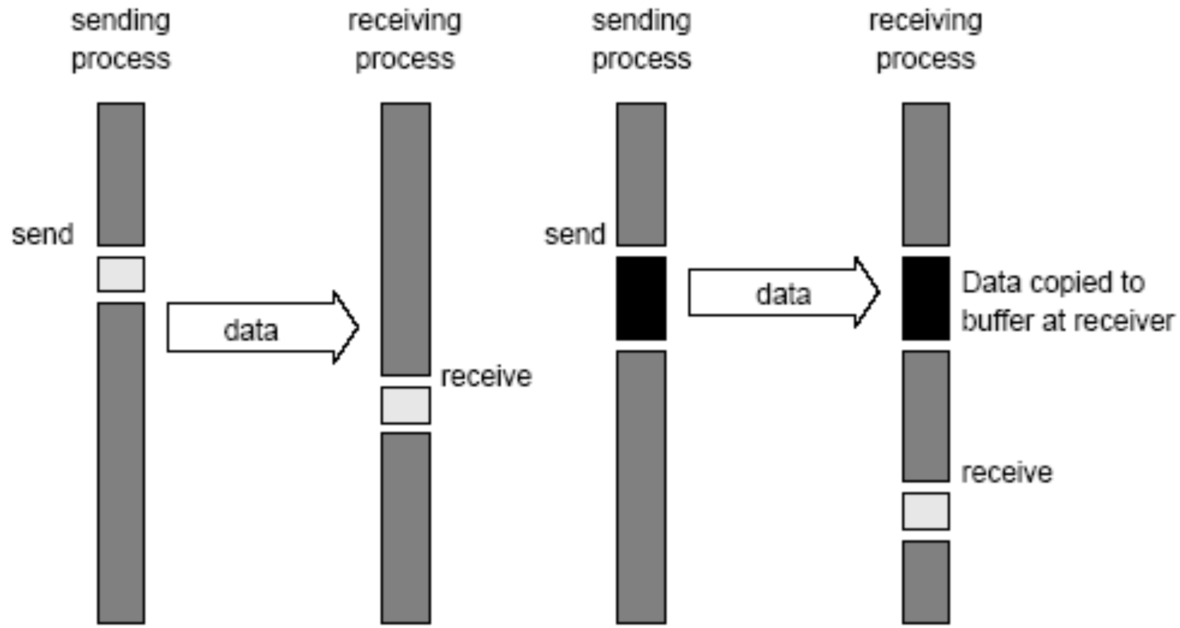
Handshake for a blocking non-buffered send/receive operation. It is easy to see that in cases where sender and receiver do not reach communication point at similar times, there can be considerable idling overheads.

+

Buffered Blocking Message Passing Operations

- 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.

Buffered Blocking Message Passing Operations



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.

Buffered Blocking Message Passing Operations

Bounded buffer sizes can have significant impact on performance.

```
P0

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

```
P1

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

What if consumer was much slower than producer?

Buffered Blocking Message Passing Operations

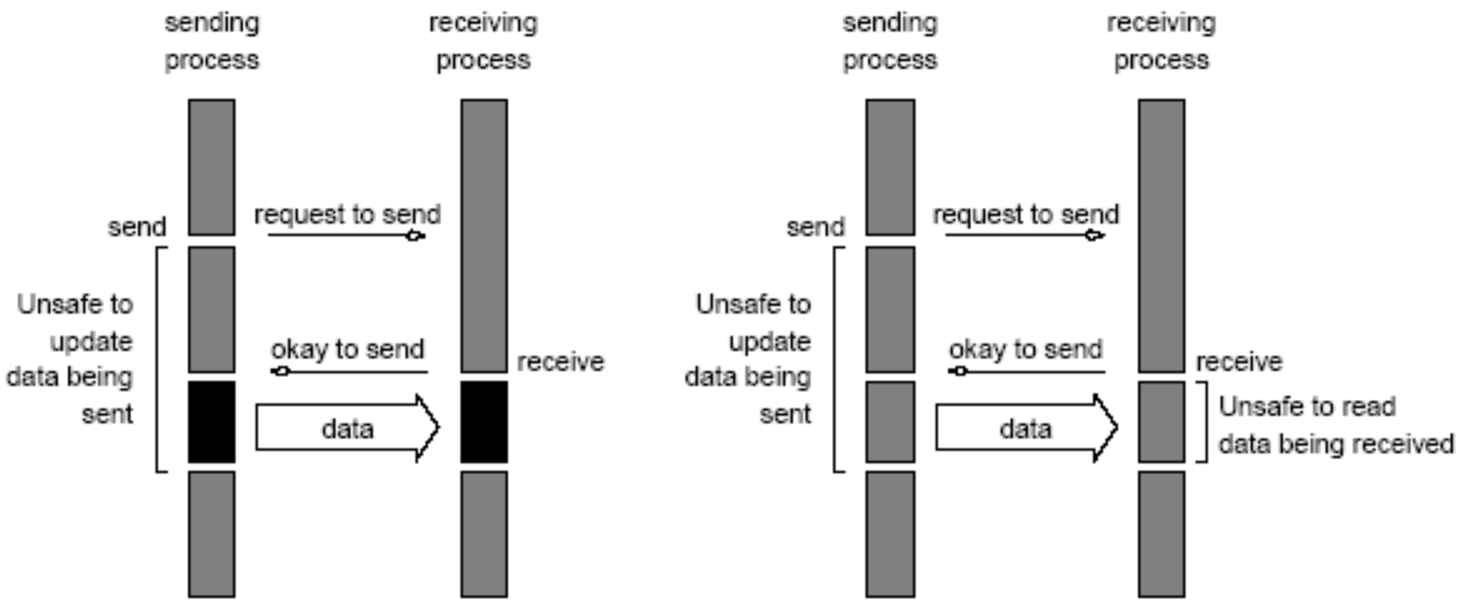
Deadlocks are still possible with buffering since receive operations block.

P0	P1
`receive(&a, 1, 1);`	`receive(&a, 1, 0);`
`send(&b, 1, 1);`	`send(&b, 1, 0);`

Non-Blocking Message Passing Operations

- 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 do so.
- Non-blocking operations are generally accompanied by a 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.

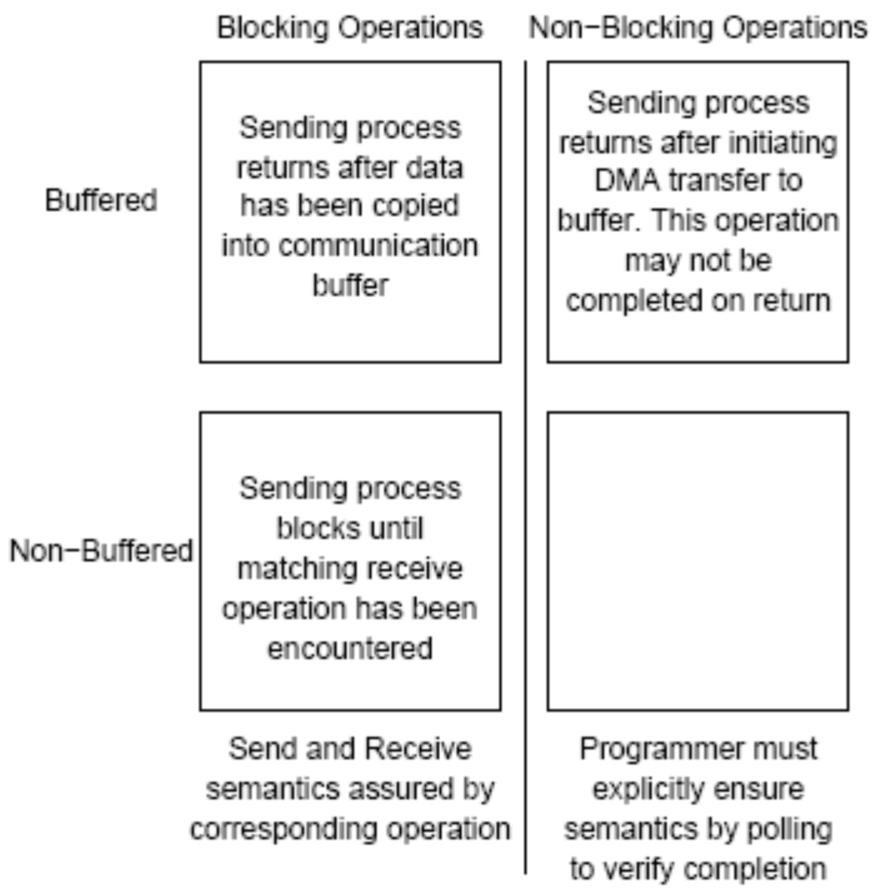
Non-Blocking Message Passing Operations



(a) Without hardware support (b) With hardware support

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

Send and Receive Protocols



Space of possible protocols for send and receive operations.

+

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`.

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 sa,sb;
...
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,&sb);
    MPI_Recv(a, 10, MPI_INT, 0, 1, MPI_COMM_WORLD,&sa);
}
...
```

If MPI_Send is blocking, there is a deadlock.

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,&status);
...
```

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,&status);
}
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,&status);
}
...
```

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)
```

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

- 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.

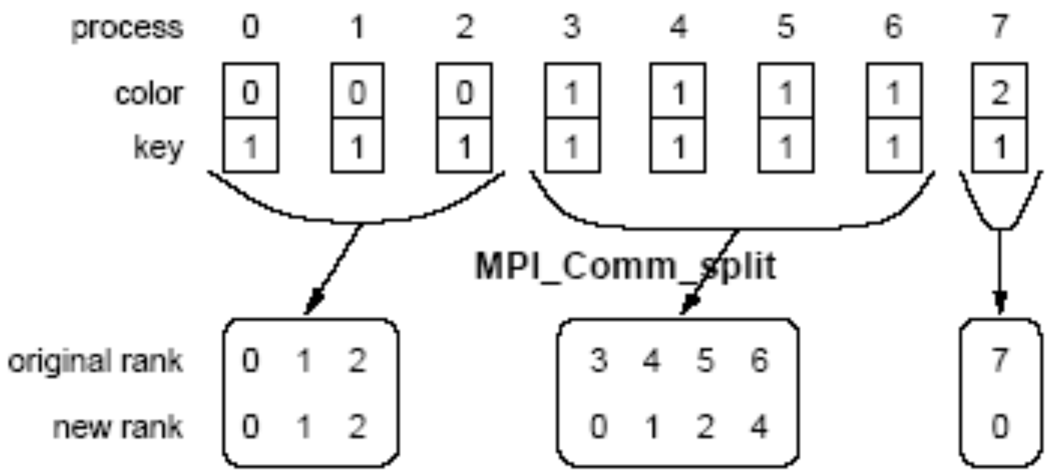
Groups and Communicators

- 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.

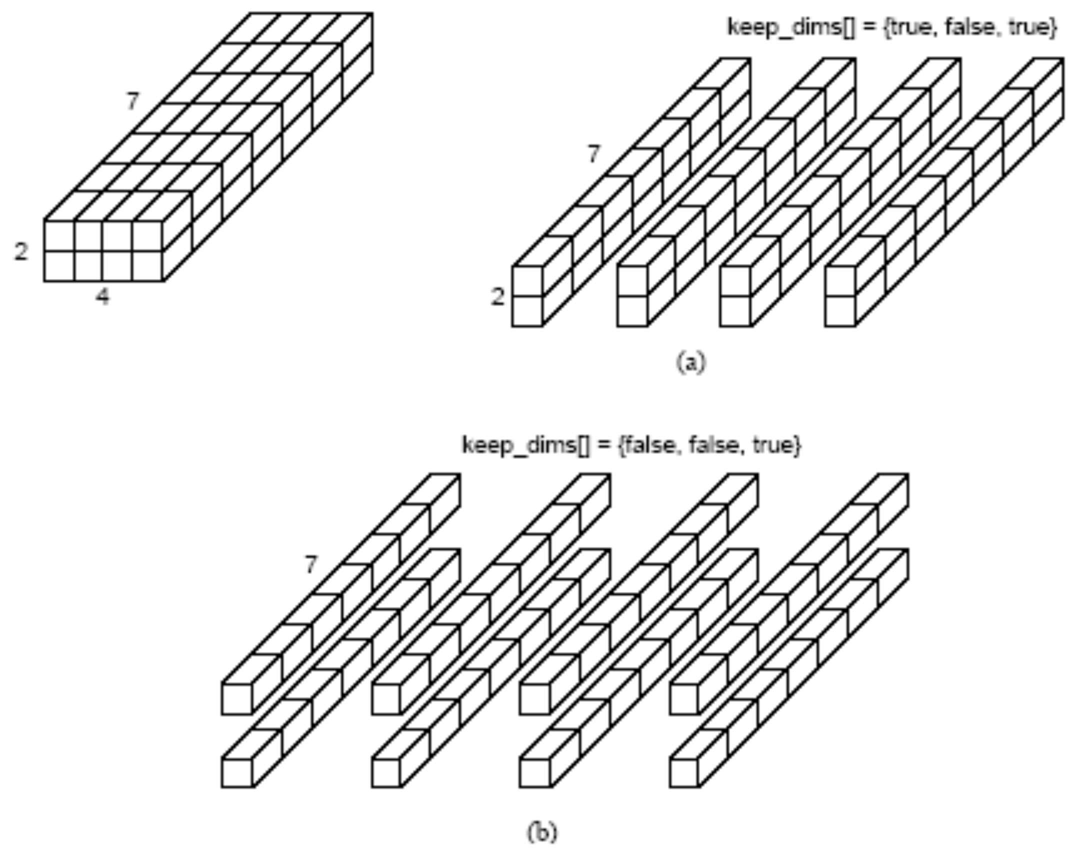
- MPI provides a convenient way to partition a Cartesian topology to form lower-dimensional grids:

```
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.