

Chapter 6

Programming Using the Message Passing Paradigm

(Original slides with modifications)

A. Grama, A. Gupta, G. Karypis, and V. Kumar

To accompany the text “Introduction to Parallel Computing”,
Addison Wesley, 2003.

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.
- Data must be explicitly partitioned and placed.
- All interactions (read-only or read/write) require cooperation between the involved processes.
- These two constraints make underlying costs very explicit to the programmer.
- 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

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

P1

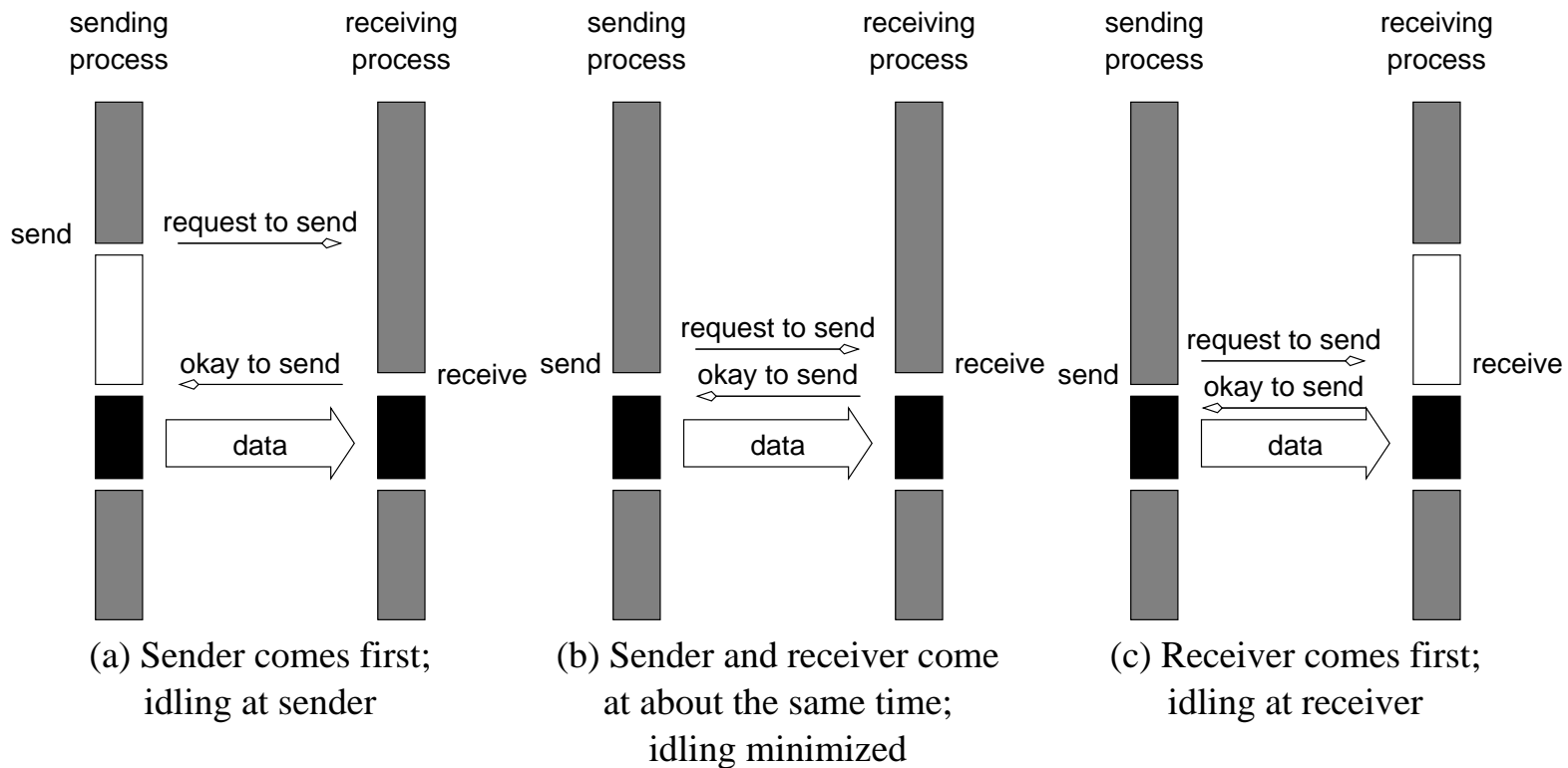
```
receive(&a, 1, 0)
printf("%d\n", a);
```

- The semantics of the send operation require that the value received by process P1 is 100 (not 0 as is changed afterwards on P0).

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.

Non-Buffered Blocking Message Passing Operations

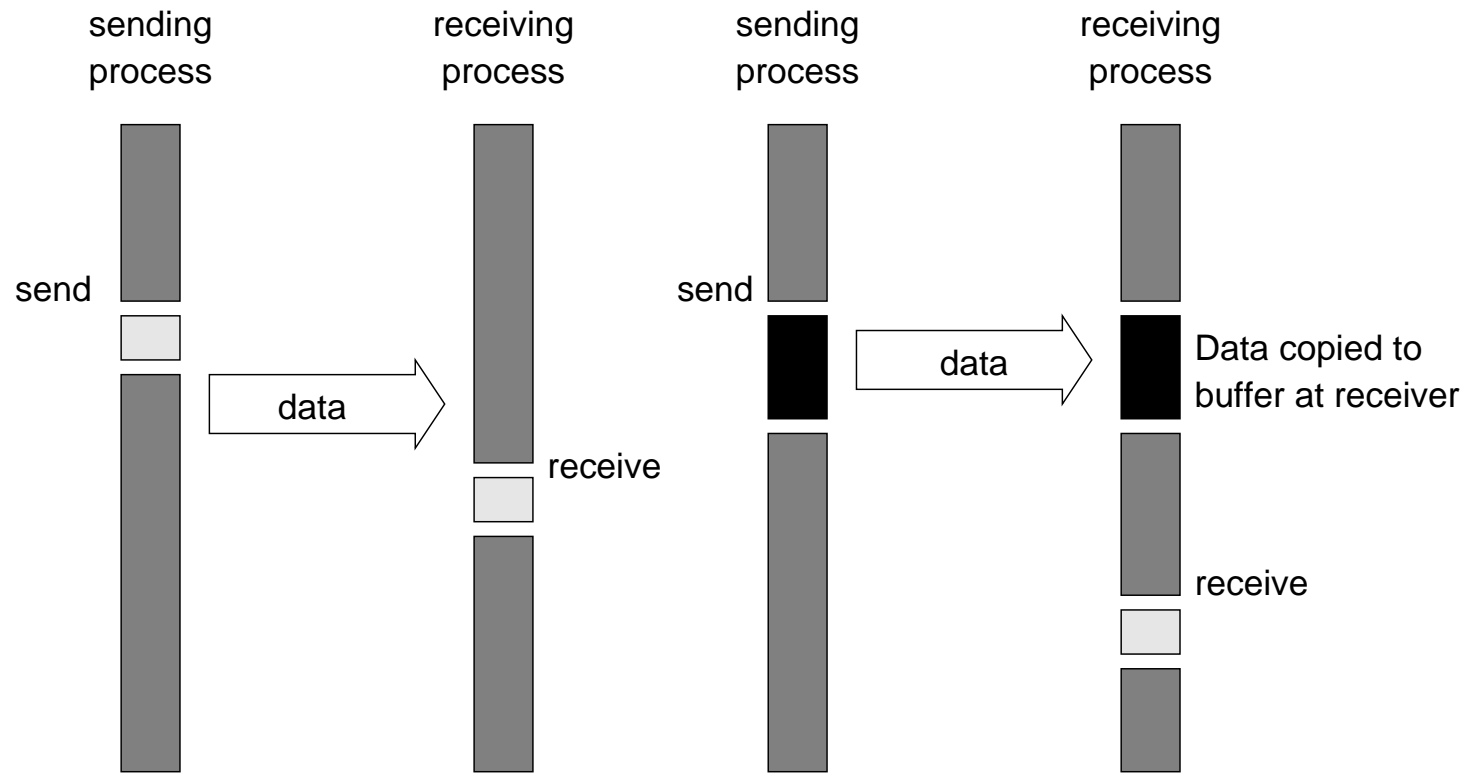


Handshake for a blocking non-buffered send/receive operation. It is easy to see that in cases where sender and receiver do not reach communication point at similar times, there can be considerable idling overheads.

Buffered Blocking Message Passing Operations

- A simple solution to the idling and deadlocking problem is to rely on buffers at the sending and receiving ends.
- The sender simply copies the data into a 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 P1 was much slower than P0?

Buffered Blocking Message Passing Operations

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);
```

Whereas the following situation is fine (provided that the buffer is large enough)

P0

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

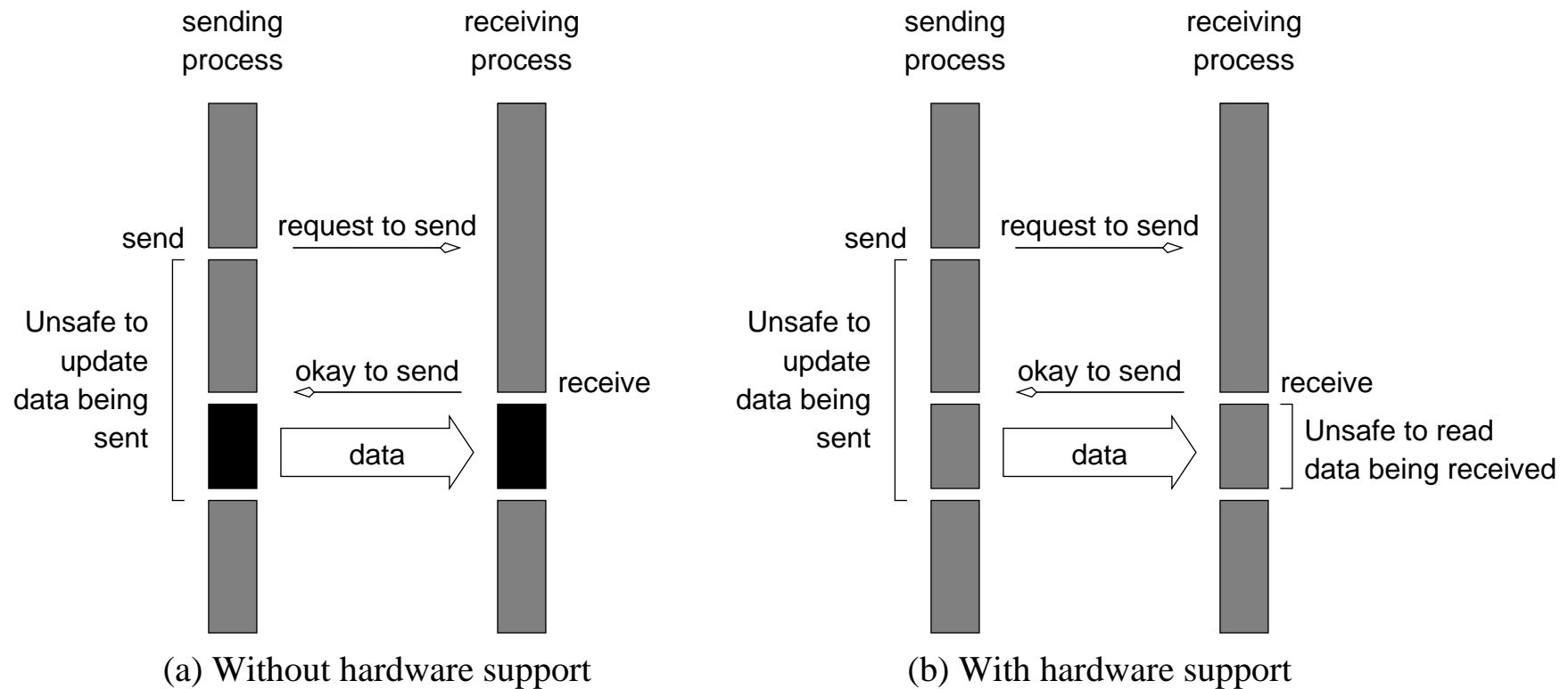
P1

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



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

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/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.

<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 the calling process.
<code>MPI_Send</code>	Sends a message.
<code>MPI_Recv</code>	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()
```

- 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 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 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 and non-buffered, 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 $i - 1$ (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.

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)
```

Creating and Using 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`.

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)  
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)
```

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)
```

Avoiding Deadlocks

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.

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.

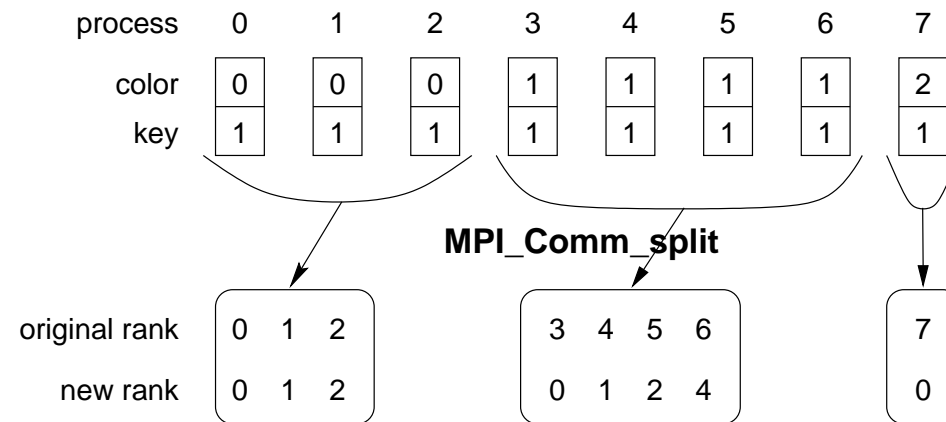
Groups and Communicators

- 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.