- *Avoiding Deadlocks:* The semantics of MPI_Send and MPI_Recv place some restrictions on how we can mix and match send and receive operations.
  - For example, consider the following piece of code in which process 0 sends two messages with different tags to process 1, and process 1 receives them in the reverse order.

```
1   int a[10], b[10], myrank;
2   MPI_Status status;
3   ...
4   MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
5   if (myrank == 0) {
6     MPI_Send(a, 10, MPI_INT, 1, 1, MPI_COMM_WORLD);
7     MPI_Send(b, 10, MPI_INT, 1, 2, MPI_COMM_WORLD);
8   }
9   else if (myrank == 1) {
10    MPI_Recv(b, 10, MPI_INT, 0, 2, MPI_COMM_WORLD);
11    MPI_Recv(a, 10, MPI_INT, 0, 1, MPI_COMM_WORLD);
12  }
13  ...
```

  - If MPI_Send is implemented using buffering, then this code will run correctly provided that sufficient buffer space is available.
  - However, if MPI_Send is implemented by blocking until the matching receive has been issued, then neither of the two processes will be able to proceed. This code fragment is not safe, as its behavior is implementation dependent. It is up to the programmer to ensure that his or her program will run correctly on any MPI implementation.
  - The problem in this program can be corrected by matching the order in which the send and receive operations are issued. Similar deadlock situations can also occur when a process sends a message to itself. Even though this is legal, its behavior is implementation dependent and must be avoided.
  - Improper use of MPI_Send and MPI_Recv can also lead to deadlocks in situations when each processor needs to send and receive a message in a circular fashion.
  - 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).

```
1   int a[10], b[10], npes, myrank;
2   MPI_Status status;
3   ...
4   MPI_Comm_size(MPI_COMM_WORLD, &npes);
5   MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
6   MPI_Send(a, 10, MPI_INT, (myrank+1)%npes, 1, MPI_COMM_WORLD);
7   MPI_Recv(b, 10, MPI_INT, (myrank-1+npes)%npes, 1, MPI_COMM_WORLD);
8   ...
```

- When MPI_Send is implemented using buffering, the program will work correctly, since every call to MPI_Send will get buffered, allowing the call of the MPI_Recv to be performed, which will transfer the required data.
- However, if MPI_Send blocks until the matching receive has been issued, all processes will enter an infinite wait state, waiting for the neighboring process to issue a MPI_Recv operation.
- Note that the deadlock still remains even when we have only two processes. Thus, when pairs of processes need to exchange data, the above method leads to an unsafe program. The above example can be made safe, by rewriting it as follows:

```
1   int a[10], b[10], npes, myrank;
2   MPI_Status status;
3   ...
4   MPI_Comm_size(MPI_COMM_WORLD, &npes);
5   MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
6   if (myrank%2 == 1) {
7     MPI_Send(a, 10, MPI_INT, (myrank+1)%npes, 1, MPI_COMM_WORLD);
8     MPI_Recv(b, 10, MPI_INT, (myrank-1+npes)%npes, 1, MPI_COMM_WORLD);
9   }
10  else {
11    MPI_Recv(b, 10, MPI_INT, (myrank-1+npes)%npes, 1, MPI_COMM_WORLD);
12    MPI_Send(a, 10, MPI_INT, (myrank+1)%npes, 1, MPI_COMM_WORLD);
13  }
14  ...
```

This new implementation partitions the processes into two groups. One consists of the odd-numbered processes and the other of the even-numbered processes.

- *Sending and Receiving Messages Simultaneously:* The above communication pattern appears frequently in many message-passing programs, and for this reason MPI provides the MPI_Sendrecv function that both sends and receives a message.
  - MPI_Sendrecv does not suffer from the circular deadlock problems of MPI_Send and MPI_Recv.
  - You can think of MPI_Sendrecv as allowing data to travel for both send and receive simultaneously. The calling sequence of MPI_Sendrecv is the following:

    ```
    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 of MPI_Sendrecv are essentially the combination of the arguments of MPI_Send and MPI_Recv.
  - The send and receive buffers must be disjoint, and the source and destination of the messages can be the same or different.
  - The safe version of our earlier example using MPI_Sendrecv is as follows.

    ```
    1   int a[10], b[10], npes, myrank;
    2   MPI_Status status;
    3   ...
    4   MPI_Comm_size(MPI_COMM_WORLD, &npes);
    5   MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
    6   MPI_SendRecv(a, 10, MPI_INT, (myrank+1)%npes, 1,
    7                b, 10, MPI_INT, (myrank-1+npes)%npes, 1,
    8                MPI_COMM_WORLD, &status);
    9   ...
    ```

  - In many programs, the requirement for the send and receive buffers of MPI_Sendrecv be disjoint may force us to use a temporary buffer. This increases the amount of memory required by the program and also increases the overall run time due to the extra copy.
  - This problem can be solved by using that MPI_Sendrecv_replace MPI function. This function performs a blocking send and receive, but it uses a single buffer for both the send and receive operation.

Machine	T_s (µs/mesg)	T_b (µs/mesg)	T_{fp} (µs/mesg)
iPSC	4100	2.8	25
nCUBE/10	400	2.6	8.3
iPSC/2	700	0.36	3.4
nCUBE/2	160	0.45	0.50
iPSC/860	160	0.36	0.033
CM-5	86	0.12	0.33

Figure 1: Performance of Send/Receive on a Number of Message Passing Machines.

That is, the received data replaces the data that was sent out of the buffer.

- Figure 1 shows the performance of the send/receive on a number of message passing machines. In this table, T_s represents the message start-up cost, T_b represents the per-byte cost, and T_{fp} is the average cost of a floating-point operation. It should be noted that the CM-5 is blocking and uses a three-phase protocol. The iPSC long messages also use a three-phase protocol in order to guarantee that enough buffer space is available at the receiving node.

0.1 Overlapping Communication with Computation

- The MPI programs we developed so far used blocking send and receive operations whenever they needed to perform point-to-point communication. Recall that a blocking send operation remains blocked until the message has been copied out of the send buffer (either into a system buffer at the source process or sent to the destination process).

- Similarly, a blocking receive operation returns only after the message has been received and copied into the receive buffer.

- It will be preferable if we can overlap the transmission of the data with the computation, as many recent distributed-memory parallel computers have dedicated communication controllers that can perform the transmission of messages without interrupting the CPUs.

0.1.1 Non-Blocking Communication Operations

- In order to overlap communication with computation, MPI provides a pair of functions for performing non-blocking send and receive operations. These functions are MPI_Isend and MPI_Irecv.
- MPI_Isend starts a send operation but does not complete, that is, it returns before the data is copied out of the buffer.
- Similarly, MPI_Irecv starts a receive operation but returns before the data has been received and copied into the buffer.
- With the support of appropriate hardware, the transmission and reception of messages can proceed concurrently with the computations performed by the program upon the return of the above functions.
- However, at a later point in the program, a process that has started a non-blocking send or receive operation must make sure that this operation has completed before it proceeds with its computations. This is because a process that has started a non-blocking send operation may want to overwrite the buffer that stores the data that are being sent, or a process that has started a non-blocking receive operation may want to use the data it requested.
- To check the completion of non-blocking send and receive operations, MPI provides a pair of functions MPI_Test and MPI_Wait. The first tests whether or not a non-blocking operation has finished and the second waits (i.e., gets blocked) until a non-blocking operation actually finishes.

```
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)
```

- MPI_Isend and MPI_Irecv functions allocate a request object and return a pointer to it in the *request* variable. This *request* object is used as an argument in the MPI_Test and MPI_Wait functions to identify the operation whose status we want to query or to wait for its completion.
- Note that the MPI_Irecv function does not take a *status* argument similar to the blocking receive function, but the status information associated with the receive operation is returned by the MPI_Test and MPI_Wait functions.

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

- MPI_Test tests whether or not the non-blocking send or receive operation identified by its *request* has finished.
  - It returns flag = true (non-zero value in C) if it completed, otherwise it returns false (a zero value in C).
  - In the case that the non-blocking operation has finished, the *request* object pointed to by *request* is deallocated and *request* is set to MPI_REQUEST_NULL. Also the *status* object is set to contain information about the operation.
  - If the operation has not finished, *request* is not modified and the value of the *status* object is undefined. The MPI_Wait function blocks until the non-blocking operation identified by *request* completes.
- For the cases that the programmer wants to explicitly deallocate a *request* object, MPI provides the following function.

  ```
  int MPI_Request_free(MPI_Request *request)
  ```

  Note that the deallocation of the request object does not have any effect on the associated non-blocking send or receive operation. That is, if it has not yet completed it will proceed until its completion. Hence, one must be careful before explicitly deallocating a request object, since without it, we cannot check whether or not the non-blocking operation has completed.
- A non-blocking communication operation can be matched with a corresponding blocking operation. For example, a process can send a message using a non-blocking send operation and this message can be received by the other process using a blocking receive operation.
- *Avoiding Deadlocks;* by using non-blocking communication operations we can remove most of the deadlocks associated with their blocking counterparts. For example, the following piece of code is not safe.

```
1   int a[10], b[10], myrank;
2   MPI_Status status;
3   ...
```

```
 4   MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
 5   if (myrank == 0) {
 6     MPI_Send(a, 10, MPI_INT, 1, 1, MPI_COMM_WORLD);
 7     MPI_Send(b, 10, MPI_INT, 1, 2, MPI_COMM_WORLD);
 8   }
 9   else if (myrank == 1) {
10     MPI_Recv(b, 10, MPI_INT, 0, 2, &status, MPI_COMM_WORLD);
11     MPI_Recv(a, 10, MPI_INT, 0, 1, &status, MPI_COMM_WORLD);
12   }
13   ...
```

However, if we replace either the send or receive operations with their non-blocking counterparts, then the code will be safe, and will correctly run on any MPI implementation.

```
 1   int a[10], b[10], myrank;
 2   MPI_Status status;
 3   MPI_Request requests[2];
 4   ...
 5   MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
 6   if (myrank == 0) {
 7     MPI_Send(a, 10, MPI_INT, 1, 1, MPI_COMM_WORLD);
 8     MPI_Send(b, 10, MPI_INT, 1, 2, MPI_COMM_WORLD);
 9   }
10   else if (myrank == 1) {
11     MPI_Irecv(b, 10, MPI_INT, 0, 2, &requests[0], MPI_COMM_WORLD);
12     MPI_Irecv(a, 10, MPI_INT, 0, 1, &requests[1], MPI_COMM_WORLD);
13   }
14   ...
```

This example also illustrates that the non-blocking operations started by any process can finish in any order depending on the transmission or reception of the corresponding messages. For example, the second receive operation will finish before the first does.

0.2 Collective Communication and Computation Operations

- MPI provides an extensive set of functions for performing many commonly used collective communication operations. All of the collective

communication functions provided by MPI take as an argument a communicator that defines the group of processes that participate in the collective operation.

- All the processes that belong to this communicator participate in the operation, and all of them must call the collective communication function.

- Even though collective communication operations do not act like barriers, these act like a virtual synchronization step in the following sense: the parallel program should be written such that it behaves correctly even if a global synchronization is performed before and after the collective call.

  - *Barrier;* the barrier synchronization operation is performed in MPI using the MPI_Barrier function.

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

    The only argument of MPI_Barrier is the communicator that defines the group of processes that are synchronized. The call to MPI_Barrier returns only after all the processes in the group have called this function.

  - *Broadcast;* the one-to-all broadcast operation is performed in MPI using the MPI_Bcast function.

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

    MPI_Bcast sends the data stored in the buffer *buf* of process *source* to all the other processes in the group. The data received by each process is stored in the buffer *buf*. The data that is broadcast consist of *count* entries of type *datatype*.

- Since the operations are virtually synchronous, they do not require tags.

- In some of the collective functions data is required to be sent from a single process (source-process) or to be received by a single process (target-process). In these functions, the source- or target-process is one of the arguments supplied to the routines. All the processes in the group (i.e., communicator) must specify the same source- or target-process.

Table 1: 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

0.2.1 Reduction

- The all-to-one reduction operation is performed in MPI using the MPI_Reduce function.

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

- MPI_Reduce combines the elements stored in the buffer *sendbuf* of each process in the group, using the operation specified in *op*, and returns the combined values in the buffer *recvbuf* of the process with rank *target*.

- Both the *sendbuf* and *recvbuf* must have the same number of *count* items of type *datatype*. Note that all processes must provide a *recvbuf* array, even if they are not the *target* of the reduction operation.

- When *count* is more than one, then the combine operation is applied element-wise on each entry of the sequence.

- MPI provides a list of predefined operations that can be used to combine the elements stored in *sendbuf* (See Table 1). MPI also allows programmers to define their own operations.

0.2.2 Gather

- The gather operation is performed in MPI using the MPI_Gather function.

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

- Each process, including the *target* process, sends the data stored in the array *sendbuf* to the *target* process. As a result, if p is the number of processors in the communication *comm*, the *target* process receives a total of p buffers.

- The data is stored in the array *recvbuf* of the target process, in a rank order. That is, the data from process with rank i are stored in the *recvbuf* starting at location i * *sendcount* (assuming that the array *recvbuf* is of the same type as *recvdatatype*).

- The data sent by each process must be of the same size and type. That is, MPI_Gather must be called with the *sendcount* and *senddatatype* arguments having the same values at each process.

- The information about the receive buffer, its length and type applies only for the target process and is ignored for all the other processes. The argument *recvcount* specifies the number of elements received by each process and not the total number of elements it receives. So, *recvcount* must be the same as *sendcount* and their datatypes must be matching.

- MPI also provides the MPI_Allgather function in which the data are gathered to all the processes and not only at the target process.

```
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.

```
int MPI_Gatherv(void *sendbuf, int sendcount,
        MPI_Datatype senddatatype, void *recvbuf,
        int *recvcounts, int *displs,
        MPI_Datatype recvdatatype, int target, MPI_Comm comm)

int MPI_Allgatherv(void *sendbuf, int sendcount,
        MPI_Datatype senddatatype, void *recvbuf,
        int *recvcounts, int *displs, MPI_Datatype recvdatatype,
        MPI_Comm comm)
```

  These functions allow a different number of data elements to be sent by each process by replacing the *recvcount* parameter with the array *recvcounts*.

0.2.3 Scatter

- The scatter operation is performed in MPI using the MPI_Scatter function.

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

- The *source* process sends a different part of the send buffer *sendbuf* to each processes, including itself.
- The data that are received are stored in *recvbuf*. Process i receives *sendcount* contiguous elements of type *senddatatype* starting from the i * *sendcount* location of the *sendbuf* of the *source* process (assuming that *sendbuf* is of the same type as *senddatatype*).
- Similarly to the gather operation, MPI provides a vector variant of the scatter operation, called MPI_Scatterv, that allows different amounts of data to be sent to different processes.

0.2.4 All-to-All

- The all-to-all personalized communication operation is performed in MPI by using the MPI_Alltoall function.

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

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

- Each process sends to process *i* *sendcount* contiguous elements of type *senddatatype* starting from the *i * sendcount* location of its *sendbuf* array.

- The data that are received are stored in the *recvbuf* array. Each process receives from process *i* *recvcount* elements of type *recvdatatype* and stores them in its *recvbuf* array starting at location *i * recvcount*.

- MPI also provides a vector variant of the all-to-all personalized communication operation called MPI_Alltoallv that allows different amounts of data to be sent to and

0.3 Groups and Communicators

- In many parallel algorithms, communication operations need to be restricted to certain subsets of processes. MPI provides several mechanisms for partitioning the group of processes that belong to a communicator into subgroups each corresponding to a different communicator. A general method for partitioning a graph of processes is to use MPI_Comm_split that is defined as follows:

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

- This function is a collective operation, and thus needs to be called by all the processes in the communicator *comm*.

- The function takes *color* and *key* as input parameters in addition to the communicator, and partitions the group of processes in the communicator *comm* into disjoint subgroups.

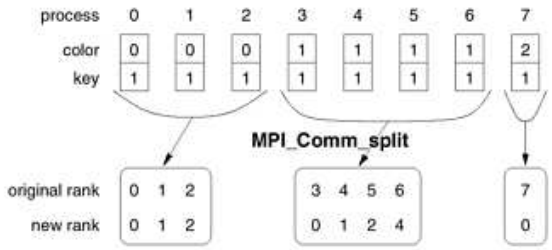

Figure 2: Using MPI_Comm_split to split a group of processes in a communicator into subgroups.

- Each subgroup contains all processes that have supplied the same value for the *color* parameter.
- Within each subgroup, the processes are ranked in the order defined by the value of the *key* parameter, with ties broken according to their rank in the old communicator (i.e., *comm*).
- A new communicator for each subgroup is returned in the *newcomm* parameter.
- Figure 2 shows an example of splitting a communicator using the MPI_Comm_split function. If each process called MPI_Comm_split using the values of parameters *color* and *key* as shown in Fig 2, then three communicators will be created, containing processes 0, 1, 2, 3, 4, 5, 6, and 7, respectively.