6.6.1 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.

- MPI_Barrier directive is used to synchronize all the processes within a given communicator.
- When a process reaches an MPI_Barrier call, it will block until all other processes in the same communicator have also reached the barrier.
- Only once all processes have reached the barrier can they all proceed past it.
- This is useful for ensuring that all processes reach a certain point in the program before any can continue, which can help in coordinating parallel tasks and ensuring that certain operations do not commence until all necessary preceding steps have been completed by all processes.

6.6.2 Broadcast

The one-to-all broadcast operation described in Section 4.1 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. The amount of data sent by the source process must be equal to the amount of data that is being received by each process; i.e., the count and datatype fields must match on all processes.

- Buffer of the source process is copied to the buffers of other processes
- MPI_Bcast (short for "broadcast") is a collective communication operation in the Message Passing Interface (MPI) used to distribute data from one process (the root process) to all other processes within a specified communicator.

Here’s a brief overview of how MPI_Bcast works:

1. Root Process: The root process (specified by a rank within the communicator) holds the data that needs to be broadcasted.
2. Broadcast Operation: When MPI_Bcast is called, the data from the root process is sent to all other processes in the communicator.
3. Synchronization: All processes, including the root, call MPI_Bcast with the same arguments. The call ensures that every process receives the same data from the root process.

Using MPI_Bcast helps ensure all processes have a consistent view of the data, which can be critical for parallel computations requiring synchronized state or initial parameters.

6.6.3 Reduction

The all-to-one reduction operation described in Section 4.1 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 is a collective communication operation in the Message Passing Interface (MPI) that combines data from all processes in a communicator and returns the result to a single process, known as the root.

This operation is useful for performing reductions such as sum, product, maximum, minimum, and other associative operations across multiple processes.

Parameters:

1. sendbuf: Address of the buffer holding the data to be reduced. Each process supplies its own data here.
2. recvbuf: Address of the buffer where the reduced result will be stored. Only meaningful at the root process.
3. count: Number of elements in the send buffer.
4. datatype: Data type of the elements to be reduced (e.g., MPI_INT, MPI_FLOAT).
5. op: Reduction operation to apply (e.g., MPI_SUM, MPI_MAX).
6. root: Rank of the process that will receive the result of the reduction.
7. comm: Communicator encompassing the group of processes participating in the reduction.

Common Reduction Operations:

1. MPI_SUM: Sum of elements.
2. MPI_PROD: Product of elements.
3. MPI_MAX: Maximum element.
4. MPI_MIN: Minimum element.

Using MPI_Reduce allows efficient computation of global results from distributed data, which is essential for many parallel algorithms that require aggregation of results.

- Dual of one-to-all broadcast
- Every process including target provides sendbuf for its value that is to be used for the reduction
- After the reduction, reduced value is stored in recvbuf of target process
- Every process must also provide recvbuf, though it may not be target of the reduction

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. All the processes must call MPI_Reduce with the same value for count, datatype, op, target, and comm.

MPI provides a list of predefined operations that can be used to combine the elements stored in sendbuf. MPI also allows programmers to define their own operations, which is not covered in this book. The predefined operations are shown in Table 6.3. For example, in order to compute the maximum of the elements stored in sendbuf, the MPI_MAX value must be used for the op argument. Not all of these operations can be applied to all possible data-types supported by MPI. For example, a bit-wise OR operation (i.e., *op* = MPI_BOR) is not defined for real-valued data-types such as MPI_FLOAT and MPI_REAL. The last column of Table 6.3 shows the various data-types that can be used with each operation.

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

int MPI_Allreduce(void *sendbuf, void *recvbuf, int count,
    MPI_Datatype datatype, MPI_Op op, MPI_Comm comm)

Note that there is no target argument since all processes receive the result of the operation.

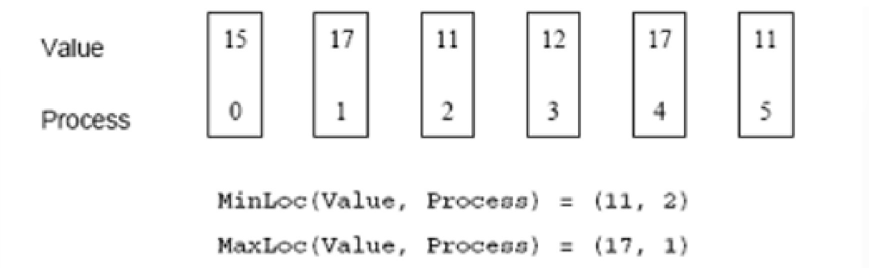
MPI_Allreduce is a collective communication operation in the Message Passing Interface (MPI) that combines data from all processes in a communicator and distributes the result back to all processes.

This operation is similar to MPI_Reduce, but instead of sending the result only to the root process, it sends the result to all participating processes.

- MPI_AllReduce is used when the result of the reduction operation is needed by all processes
- Equal to All-to-one reduction followed by one-to-all broadcast
- After Allreduce operation, recvbuf of all the processes contain reduced value
- Note: No target for reduction is given

MPI_MAXLOC and MPI_MINLOC:

- **The operation MPI_MAXLOC combines pairs of values (vi, li) and returns the pair (v, l) such that v is the maximum among all vi 's and l is the corresponding li (if there are more than one, it is the smallest among all these li 's)**
- **MPI_MINLOC does the same, except for minimum value of vi**



An example use of the MPI_MINLOC and MPI_MAXLOC operators.

6.6.4 Prefix

The prefix-sum operation described in Section 4.3 is performed in MPI using the MPI_Scan function.

int MPI_Scan(void *sendbuf, void *recvbuf, int count,
    MPI_Datatype datatype, MPI_Op op, MPI_Comm comm)

MPI_Scan performs a prefix reduction of the data stored in the buffer sendbuf at each process and returns the result in the buffer recvbuf. The receive buffer of the process with rank *i* will store, at the end of the operation, the reduction of the send buffers of the processes whose ranks range from 0 up to and including *i*. The type of supported operations (i.e., op) as well as the restrictions on the various arguments of MPI_Scan are the same as those for the reduction operation MPI_Reduce.

MPI_Scan is a collective communication operation in the Message Passing Interface (MPI) that performs a parallel prefix reduction (also known as a prefix sum) across all processes in a communicator. Each process receives the partial result of the reduction up to that process, which can be useful for various parallel algorithms that need partial sums or incremental results.

Parameters:

1. sendbuf: Address of the buffer holding the data to be reduced. Each process supplies its own data here.
2. recvbuf: Address of the buffer where the partial reduction result will be stored. Each process will get its own partial result.
3. count: Number of elements in the send buffer.
4. datatype: Data type of the elements to be reduced (e.g., MPI_INT, MPI_FLOAT).
5. op: Reduction operation to apply (e.g., MPI_SUM, MPI_MAX).

6. comm: Communicator encompassing the group of processes participating in the reduction.

- **After the operation, every process has sum of the buffers of the previous processes and its own**
- **MPI_Scan() is MPI primitive for the prefix operations**
- **All the operators that can be used for reduction can also be used for the scan operation**
- **If buffer is an array of elements, then recvbuf is also an array containing element-wise prefix at each position.**

Exclusive-Prefix (Exscan) Operation:

- **Exclusive-prefix-sum: After the operation, every process has sum of the buffers of the previous processes excluding its own**
- **MPI_Exscan() is MPI primitive for the exclusive-prefix operations**
- **The recvbuf of first process is remains unchanged as there is no process before it**
- **Some MPI distributions place identity value for the given associative operator**
- **0 for sum, -infinity for Max, infinity for Min, 1 for multiplication, and so on**

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

An exclusive prefix sum is a parallel algorithm that computes the prefix sum of an array where each element at position i in the output array is the sum of all the elements preceding i in the input array. Unlike an inclusive prefix sum, the element at position i does not include the value of the element at position i itself.

For example, given:

A= [3,1,4,1,5,9,2]

The exclusive prefix sum array PP would be:

P= [0,3,4,8,9,14,23]

Here's the breakdown:

- $P[1] = A[0] = 3$
- $P[2] = A[0] + A[1] = 3 + 1 = 4$
- $P[3] = A[0] + A[1] + A[2] = 3 + 1 + 4 = 8$
- $P[4] = A[0] + A[1] + A[2] + A[3] = 3 + 1 + 4 + 1 = 9$
- $P[5] = A[0] + A[1] + A[2] + A[3] + A[4] = 3 + 1 + 4 + 1 + 5 = 14$
- $P[6] = A[0] + A[1] + A[2] + A[3] + A[4] + A[5] = 3 + 1 + 4 + 1 + 5 + 9 = 23$

Comparison with MPI_Reduce and MPI_Allreduce:

1. MPI_Reduce: Combines data from all processes and returns the result to a single root process.
2. MPI_Allreduce: Combines data from all processes and returns the result to all processes.
3. MPI_Scan: Performs a parallel prefix reduction, giving each process the cumulative result up to that point.

6.6.5 Gather

MPI_Gather and its Variants:

- **Recall section 4.4: After the Gather operation, a single target process accumulates[concatenates] buffers of all the other processes without any reduction operator**
- **Each process sends element(s) in its *sendbuf* to the target process**
- **Total number of elements to be sent by each process must be same**
  - **This number is specified in *sendcount* and is equal to *recvcount*.**

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

MPI_Gather is a collective communication operation in the Message Passing Interface (MPI) that collects data from all processes in a communicator and gathers it into a single process, known as the root process. This operation is useful for aggregating data distributed across multiple processes into one location for further processing or analysis.

Parameters

1. sendbuf: Address of the buffer holding the data to be sent from each process.
2. sendcount: Number of elements in the send buffer.
3. sendtype: Data type of elements in the send buffer (e.g., MPI_INT, MPI_FLOAT).
4. recvbuf: Address of the buffer where the gathered data will be stored at the root process. Only significant at the root.
5. recvcount: Number of elements expected from each process.
6. recvtype: Data type of elements in the receive buffer.
7. root: Rank of the root process that will receive the gathered data.
8. comm: Communicator encompassing the group of processes participating in the gather operation.

Each process, including the target process, sends the data stored in the arraysendbuf 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 arrayrecvbuf of the target process, in a rank order. That is, the data from process with rank *i* are stored in therecvbuf starting at location *i* * sendcount (assuming that the array recvbuf is of the same type asrecvdatatype).

The data sent by each process must be of the same size and type. That is, MPI_Gather must be called with thesendcount 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 assendcount 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 arecvbuf array that will store the gathered data.

MPI_Gather and its Variants:

Gatherv:

- Each process can have different message length
- Recvcounts[i] = Total elements to be received by ith processing node
- Displs[i]= starting index in recvbuf to store message received from ith process

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

MPI_Gather and its Variants:

Gatherv (Displs Calculation Example):

- Let each process have elements one more than their rank
- Then calculation of displs[] at target is calculated as:

	P0	P1	P2	P3
	32	12, 15	4,9,14	20,23,27,31
Recvcounts	1	2	3	4
displs	0	0+1=1	1+2=3	3+3=6

MPI_Allgather:

- Same as All-to-All broadcast described in section 4.2
- Every process serve as target for the gather

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

- Note: No target for gather
- Unlike MPI_Gather, it gathers sendbufs of all the processes in recvbufs of all the processes

MPI_Allgatherv:

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

- Here every process will have to supply the valid calculated arrays of recvcounts and displs
- Furthermore, it is also necessary for all the processors to provide a recvbuf [an array] of sufficient size to store all the elements of all the processes

MPI_Gather vs. MPI_Gatherv:

- MPI_Gather requires the **same amount** of data from each process.
- MPI_Gatherv allows **different amounts** of data from each process, specified by recvcounts and displs.

MPI_Allgather vs. MPI_Allgatherv:

- MPI_Allgather requires the **same amount** of data from each process and distributes the collected data to all processes.
- MPI_Allgatherv allows **different amounts** of data from each process and distributes the collected data to all processes, specified by recvcounts and displs.

Gather vs. Allgather:

- MPI_Gather and MPI_Gatherv collect data to a **single root process**.
- MPI_Allgather and MPI_Allgatherv collect data and distribute the complete result to **all processes**.

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.

```
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. The amount of data sent by process *i* is equal to recvcounts[i]. Note that the size of recvcounts is equal to the size of the communicator comm. The array parameter displs, which is also of the same size, is used to determine where in recvbuf the data sent by each process will be stored. In particular, the data sent by process *i* are stored in recvbuf starting at location displs[i]. Note that, as opposed to the non-vector variants, the sendcount parameter can be different for different processes.

Comparison with Similar Functions

1. MPI_Scatter: Distributes data from one root process to all other processes, essentially the inverse operation of MPI_Gather.
2. MPI_Allgather: Similar to MPI_Gather, but the gathered data is distributed to all processes instead of just the root.

6.6.6 Scatter

The scatter operation described in Section 4.4 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). MPI_Scatter must be called by all the processes with the same values for the sendcount, senddatatype, recvcount, recvdatatype, source, and comm arguments. Note again that sendcount is the number of elements sent to each individual process.

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.

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

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. Note that by appropriately setting the displs array we can use MPI_Scatterv to send overlapping regions of sendbuf.

- **Sendcount and recvcount should be the same and represent total elements to be given to each process**

```
intial values at source::0::=83 86      77      15      93      35      86      92
rank=0: Received:83     86
rank=1: Received:77     15
rank=2: Received:93     35
rank=3: Received:86     92
```

MPI_Scatter distributes equal-sized chunks of data from the root process to all other processes in the communicator.

Parameters:

1. sendbuf: Starting address of the send buffer (significant only at the root).
2. sendcount: Number of elements sent to each process.
3. sendtype: Data type of elements in the send buffer.
4. recvbuf: Starting address of the receive buffer.
5. recvcount: Number of elements received by each process.
6. recvtype: Data type of elements in the receive buffer.
7. root: Rank of the root process.
8. comm: Communicator.

MPI_Scatterv extends MPI_Scatter by allowing each process to receive a different amount of data. This is useful for scenarios where the data is not evenly divisible or when the chunks of data are of varying sizes.

1. sendbuf: Starting address of the send buffer (significant only at the root).
2. sendcounts: Integer array specifying the number of elements sent to each process.
3. displs: Integer array specifying the displacement (offset) from the beginning of sendbuf for each process.
4. sendtype: Data type of elements in the send buffer.
5. recvbuf: Starting address of the receive buffer.
6. recvcount: Number of elements received by the calling process.
7. recvtype: Data type of elements in the receive buffer.
8. root: Rank of the root process.
9. comm: Communicator.

Comparison with Similar Functions

MPI_Scatter:

- Sends an equal amount of data from the root process to all other processes.
- Each process receives the same number of elements.

MPI_Scatterv:

- Sends varying amounts of data from the root process to each process.
- Allows different processes to receive different numbers of elements, specified by sendcounts.
- Displacements (displs) specify where in the send buffer each process's data begins.

6.6.7 All-to-All

The all-to-all personalized communication operation described in Section 4.5 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_Alltoall must be called by all the processes with the same values for the sendcount, senddatatype, recvcount, recvdatatype, and comm arguments. Note that sendcount and recvcount are the number of elements sent to, and received from, each individual process.

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 received from each process.

```
int MPI_Alltoallv(void *sendbuf, int *sendcounts, int *sdispls
      MPI_Datatype senddatatype, void *recvbuf, int *recvcounts,
      int *rdispls, MPI_Datatype recvdatatype, MPI_Comm comm)
```

The parameter sendcounts is used to specify the number of elements sent to each process, and the parameter sdispls is used to specify the location in sendbuf in which these elements are stored. In particular, each process sends to process *i*, starting at location sdispls[i] of the array sendbuf, sendcounts[i] contiguous elements. The parameter recvcounts is used to specify the number of elements received by each process, and the parameter rdispls is used to specify the location in recvbuf in which these elements are stored. In particular, each process receives from process *i* recvcounts[i] elements that are stored in contiguous locations of recvbuf starting at location rdispls[i]. MPI_Alltoallv must be called by all the processes with the same values for the senddatatype, recvdatatype, and comm arguments.

MPI_Alltoall is used for performing a simple all-to-all communication where each process sends the same amount of data to every other process.

Parameters:

1. sendbuf: Starting address of the send buffer. Each process sends data from this buffer.
2. sendcount: Number of elements to send to each process.
3. sendtype: Data type of elements to send.
4. recvbuf: Starting address of the receive buffer. Each process receives data into this buffer.
5. recvcount: Number of elements to receive from each process.
6. recvtype: Data type of elements to receive.
7. comm: Communicator

Usage:

- Each process sends sendcount elements to every other process.
- Each process receives recvcount elements from every other process.
- This function assumes that the amount of data sent to and received from each process is the same.

MPI_Alltoallv extends MPI_Alltoall by allowing each process to send and receive varying amounts of data to/from every other process.

Parameters:

1. sendbuf: Starting address of the send buffer.
2. sendcounts: Array specifying the number of elements to send to each process.
3. sdispls: Array specifying the displacement (offset) in the send buffer for each process's data.
4. sendtype: Data type of elements to send.
5. recvbuf: Starting address of the receive buffer.
6. recvcounts: Array specifying the number of elements to receive from each process.
7. rdispls: Array specifying the displacement (offset) in the receive buffer for each process's data.
8. recvtype: Data type of elements to receive.

9. comm: Communicator.

Usage:

- Allows each process to send a different number of elements to each process and to receive a different number of elements from each process.
- sendcounts and recvcounts specify the number of elements to send to and receive from each process, respectively.
- sdispls and rdispls specify the offsets in the send and receive buffers, respectively.

MPI_Alltoallw provides the most general form of all-to-all communication, allowing each process to send and receive varying amounts of data with potentially different data types.

Parameters:

1. sendbuf: Starting address of the send buffer.
2. sendcounts: Array specifying the number of elements to send to each process.
3. sdispls: Array specifying the displacement (offset) in the send buffer for each process's data.
4. sendtypes: Array specifying the data type of elements to send to each process.
5. recvbuf: Starting address of the receive buffer.
6. recvcounts: Array specifying the number of elements to receive from each process.
7. rdispls: Array specifying the displacement (offset) in the receive buffer for each process's data.
8. recvtypes: Array specifying the data type of elements to receive from each process.
9. comm: Communicator.

Usage:

- Allows complete flexibility in the communication patterns.
- Each process can send a different number of elements, with different data types, to each process, and receive a different number of elements, with different data types, from each process.
- sendcounts, recvcounts, sdispls, rdispls, sendtypes, and recvtypes provide full control over the number, location, and type of elements sent and received.

Summary of Differences

MPI_Alltoall:

1. Each process sends and receives the same amount of data to/from all other processes.
2. Simple and straightforward, but limited to uniform data distribution.

MPI_Alltoallv:

1. Each process can send and receive varying amounts of data to/from all other processes.
2. More flexible than MPI_Alltoall, but requires arrays to specify counts and displacements.

MPI_Alltoallw:

1. The most general form, allowing varying amounts and types of data to be sent and received.
2. Provides complete control over the data exchange patterns, but with more complex setup involving counts, displacements, and data types arrays.

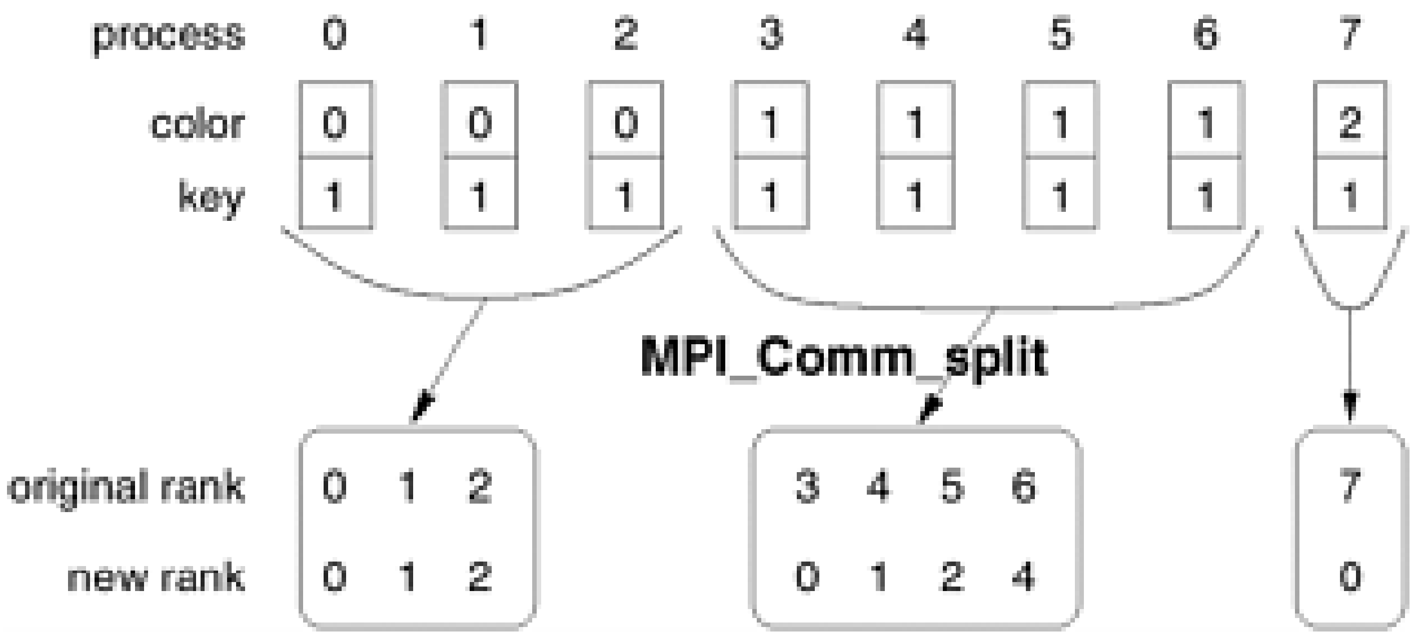
6.7 Groups and Communicators

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. 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 6.7 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 Figure 6.7, then three communicators will be created, containing processes {0, 1, 2}, {3, 4, 5, 6}, and {7}, respectively.

Figure 6.7. Using MPI_Comm_split to split a group of processes in a communicator into subgroups.



The **MPI_Comm_split** function in MPI is used to divide an existing communicator into several, smaller sub-communicators based on specified color and key values. This is particularly useful for creating groups of processes that can communicate within subgroups while remaining part of the larger global communicator.

Parameters

1. comm: The original communicator (input). This is the communicator that will be split.
2. color: An integer value used to determine the group assignment of each process (input). All processes with the same color are assigned to the same new communicator. If a process sets color to MPI_UNDEFINED, it will not be part of any new communicator.
3. key: An integer value used to determine the rank order within the new communicator (input). This determines the rank ordering of the processes in the new communicator. Processes with the same color but different keys will be ranked according to their keys, with ties broken by the rank in the original communicator.
4. newcomm: The new communicator created by the split (output). This is the communicator that will include all processes with the same color value.

How It Works:

1. **Group Assignment (color):**
    - Each process provides a color value.
    - Processes with the same color value are grouped into the same new communicator.
    - If a process provides MPI_UNDEFINED as the color, it will not be included in any new communicator.
2. **Rank Assignment (key):**

- Within each new communicator, the processes are assigned new ranks based on the key value.
- The ranks are assigned in ascending order of the key values. If two processes have the same key, their relative ranks are determined by their ranks in the original communicator.

3. **Creation of New Communicator (newcomm):**

- Each process gets a new communicator handle (newcomm), which represents its membership in the newly formed sub-communicator.
- The function returns an MPI communicator for each subgroup of processes sharing the same color.