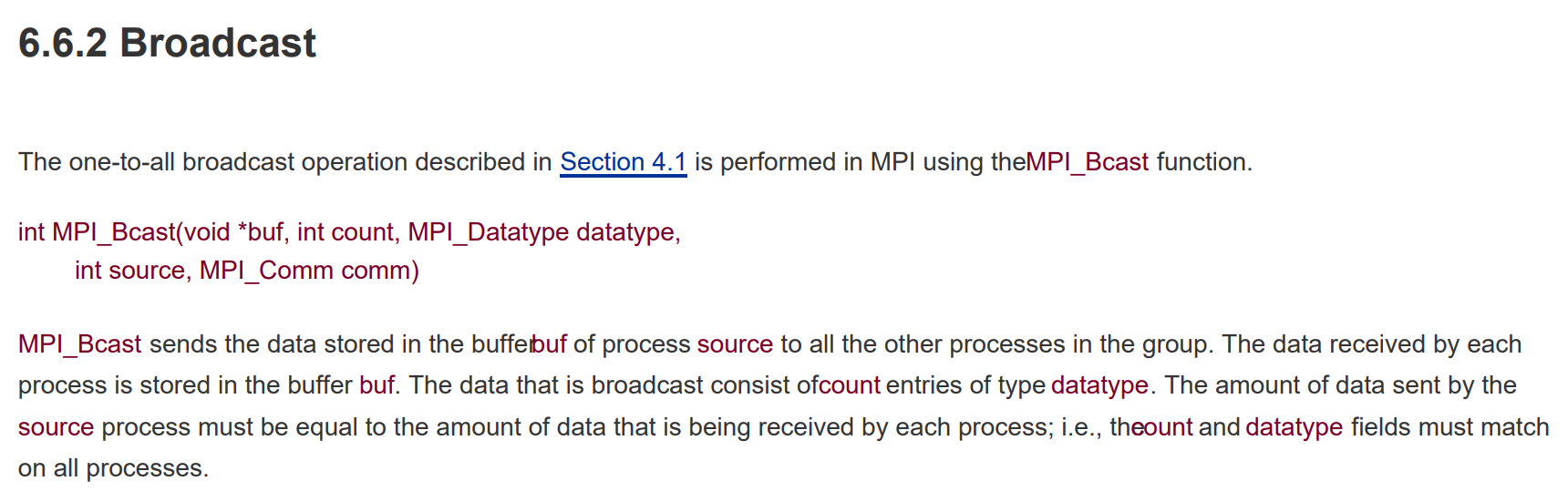


* MPI\_Barrier directive is used to synchronize all the processes within a given communicator.
* When a process reaches an MPI\_Barrier call, it will block until all other processes in the same communicator have also reached the barrier.
* Only once all processes have reached the barrier can they all 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.

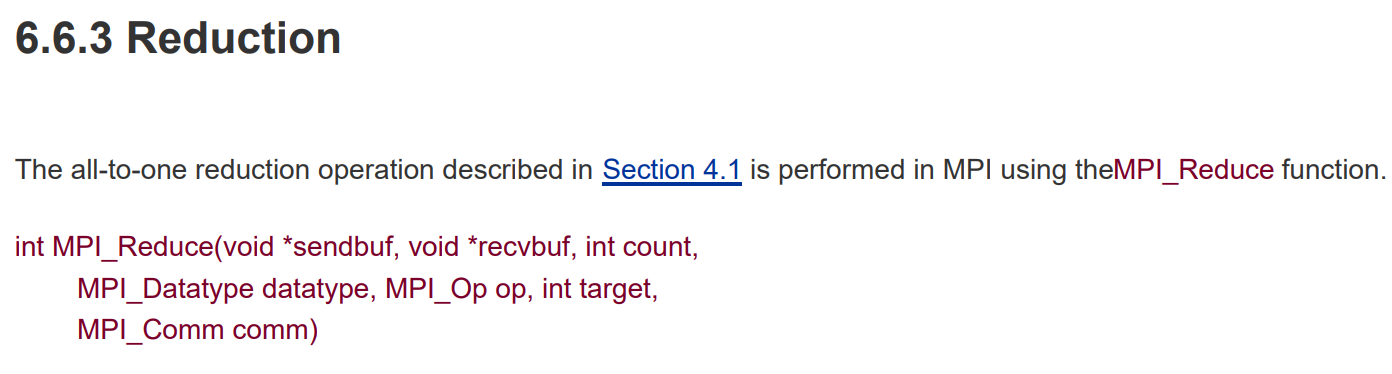


* Buffer of the source process is copied to the buffers of other processes
* MPI\_Bcast (short for "broadcast") is a collective communication operation in the Message Passing Interface (MPI) used to distribute data from one process (the root process) to all other processes within a specified communicator.

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.



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:**

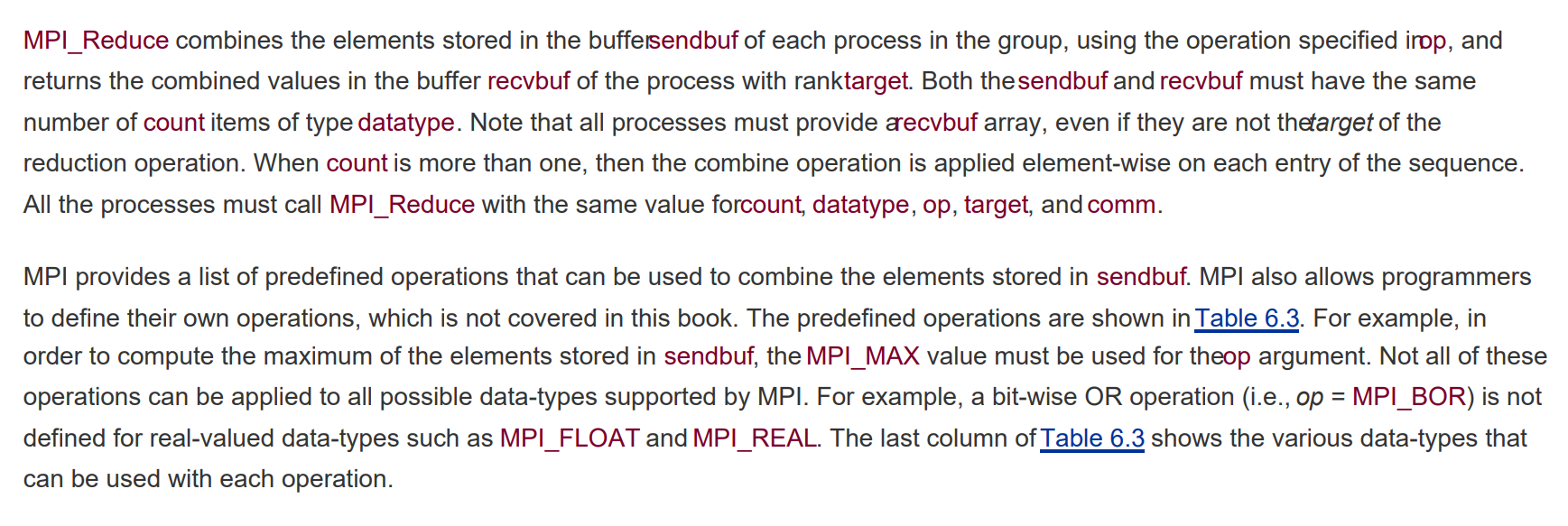
1. sendbuf: Address of the buffer holding the data to be reduced. Each process supplies its own data here.
2. recvbuf: Address of the buffer where the reduced result will be stored. Only meaningful at the root process.
3. count: Number of elements in the send buffer.
4. datatype: Data type of the elements 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.

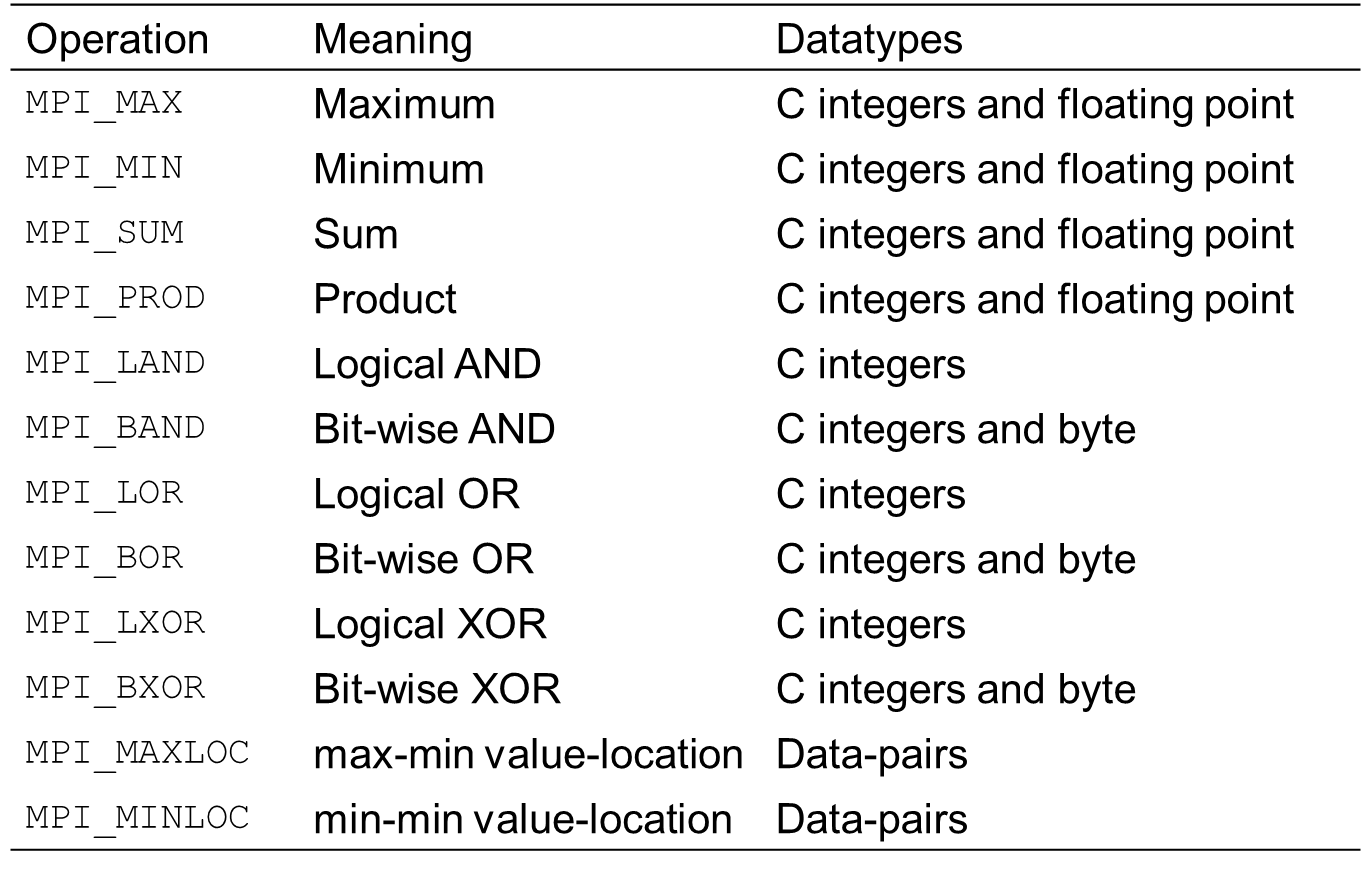
**Common Reduction Operations:**

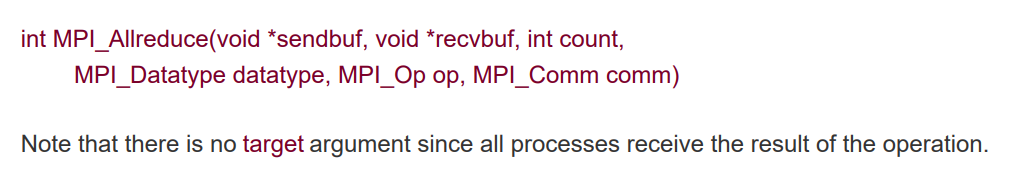
1. MPI\_SUM: Sum of elements.
2. MPI\_PROD: Product of elements.
3. MPI\_MAX: Maximum element.
4. MPI\_MIN: Minimum element.

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



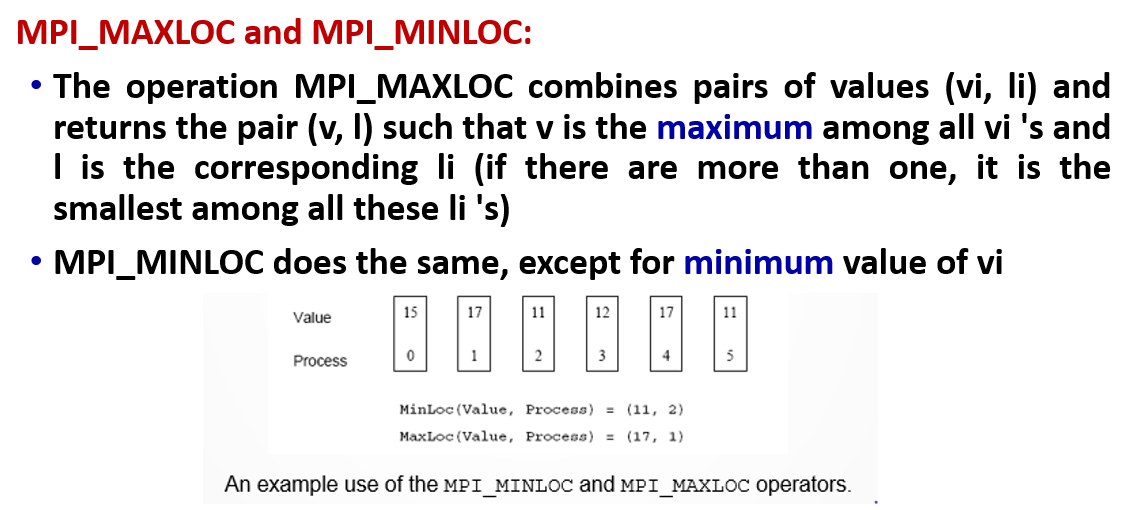


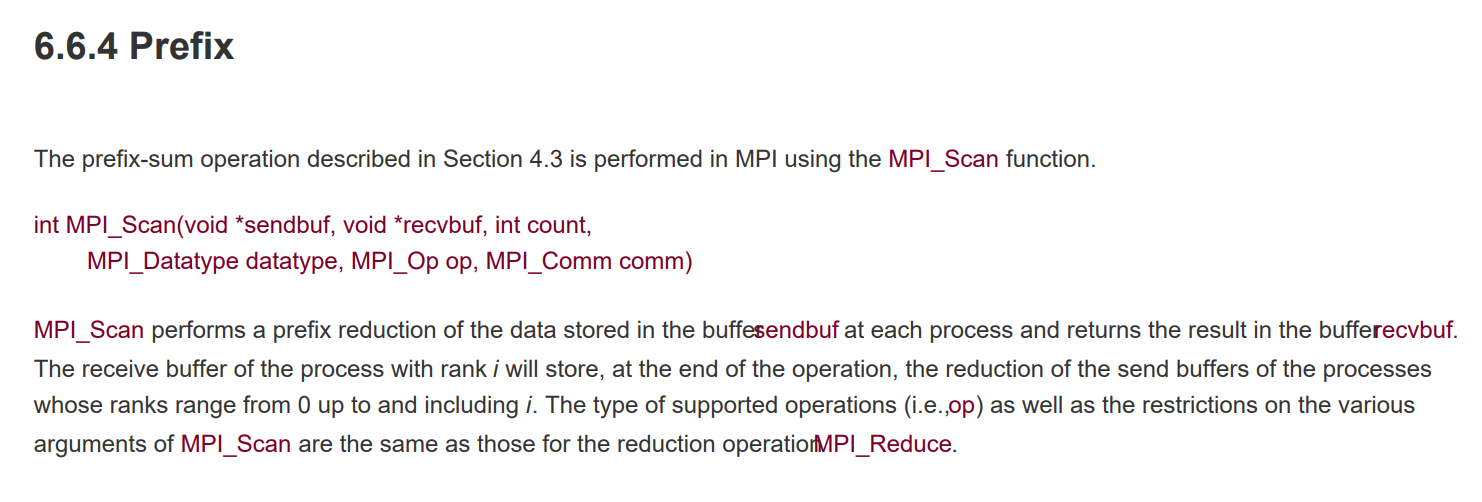


MPI\_Allreduce is a collective communication operation in the Message Passing Interface (MPI) that combines data from all processes in a communicator and distributes the result back to all processes.

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



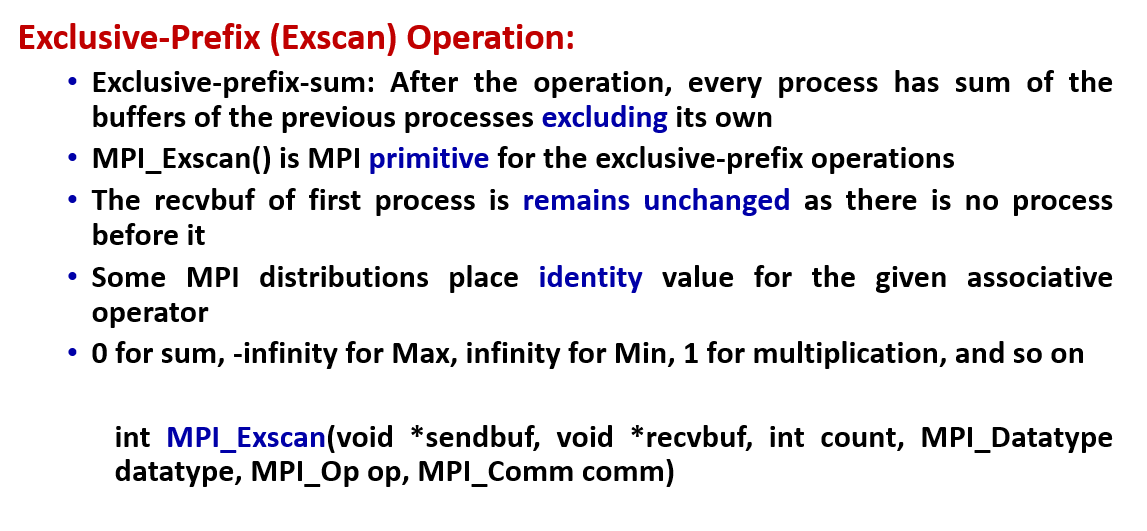


MPI\_Scan is a collective communication operation in the Message Passing Interface (MPI) that performs a parallel prefix reduction (also known as a prefix sum) across all processes in a communicator. Each process receives the partial result of the reduction up to that process, which can be useful for 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.**

****

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.

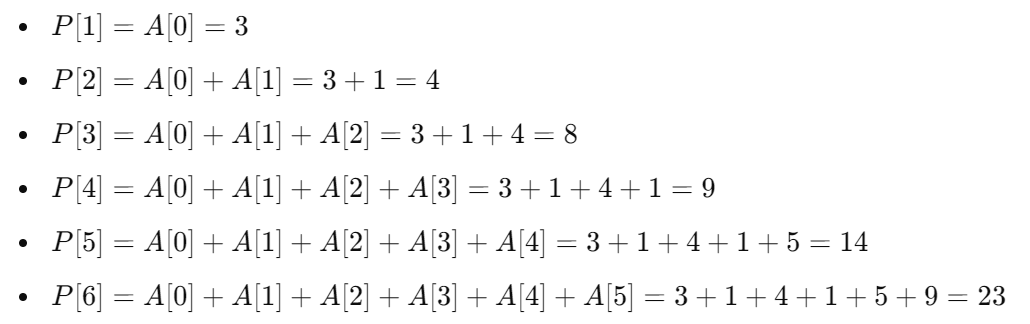
For example, given:

𝐴= [3,1,4,1,5,9,2]

The exclusive prefix sum array 𝑃*P* would be:

𝑃= [0,3,4,8,9,14,23]

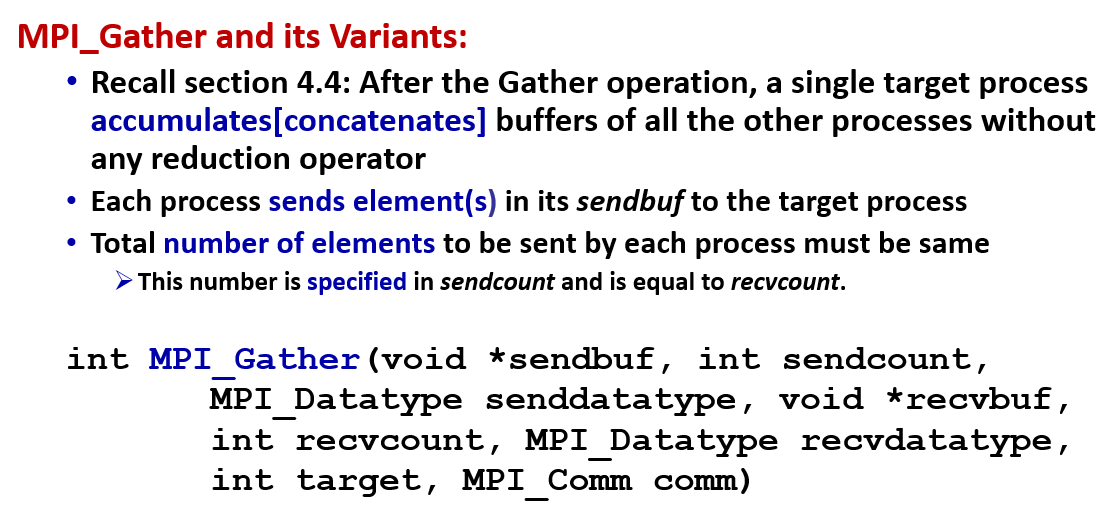
Here’s the breakdown:



**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.

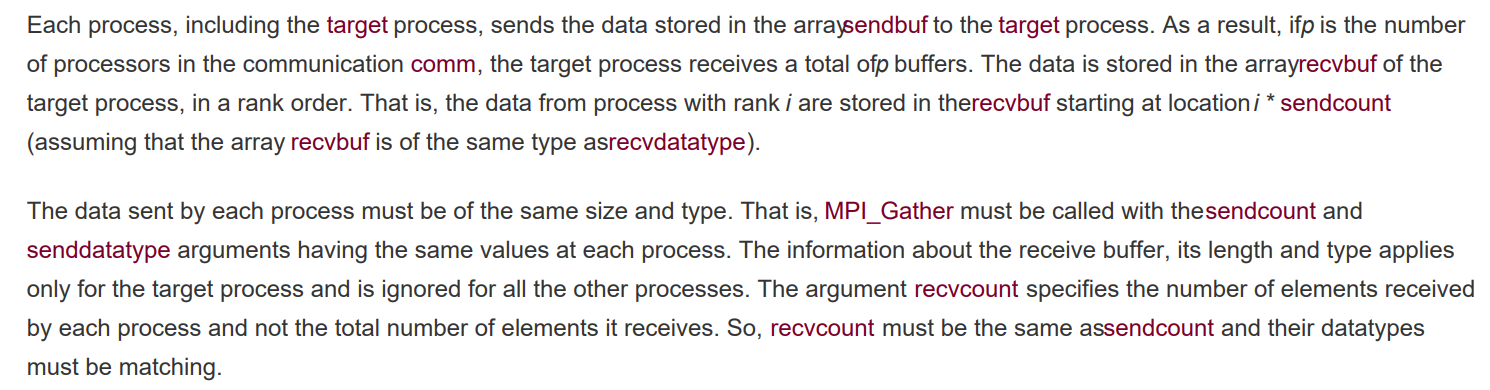


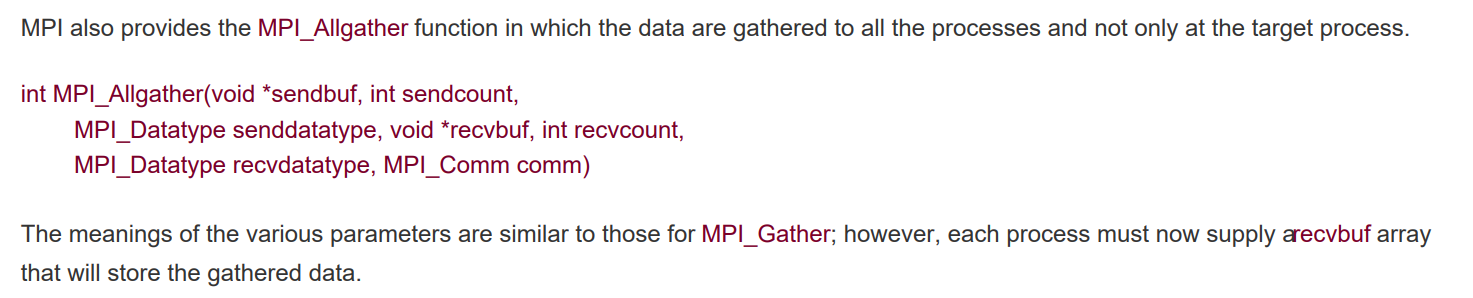


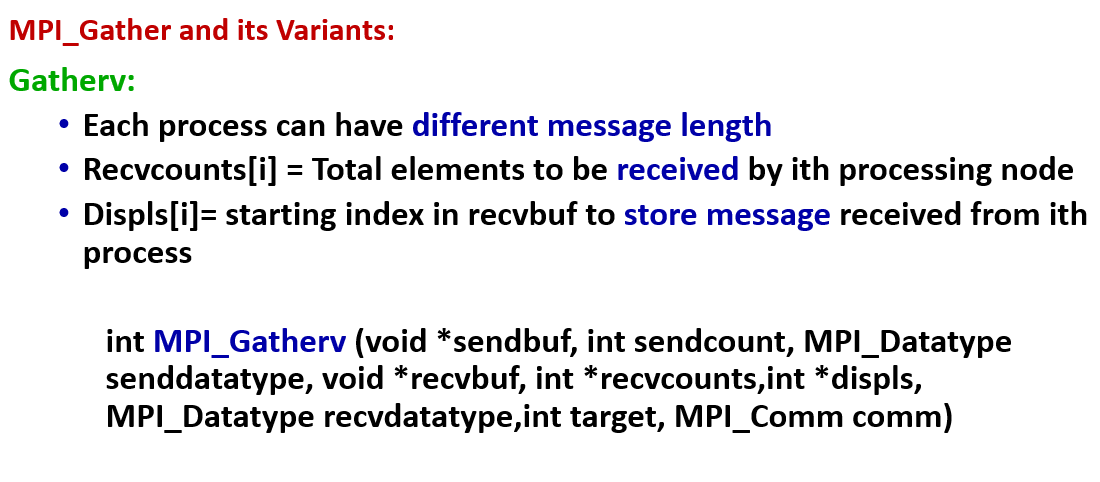
MPI\_Gather is a collective communication operation in the Message Passing Interface (MPI) that collects data from all processes in a communicator and gathers it into a single process, known as the root process. This operation is useful for aggregating data distributed across multiple processes into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

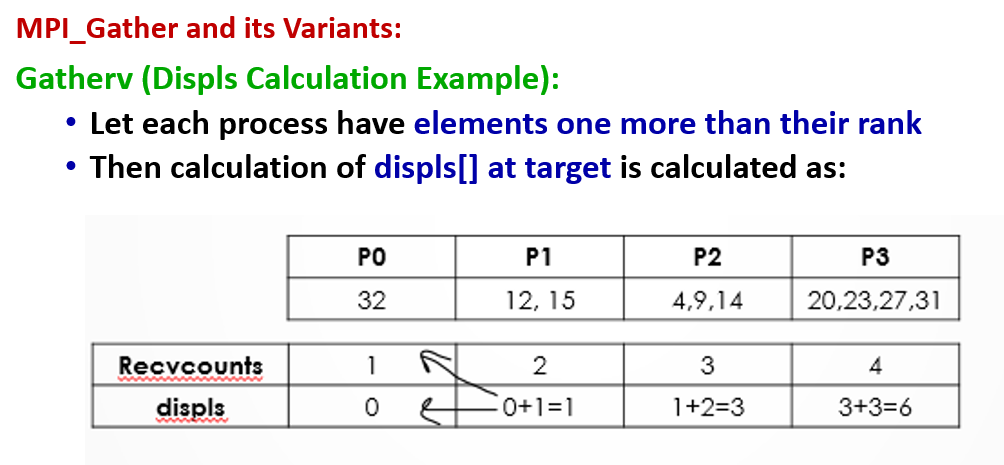
**Parameters**

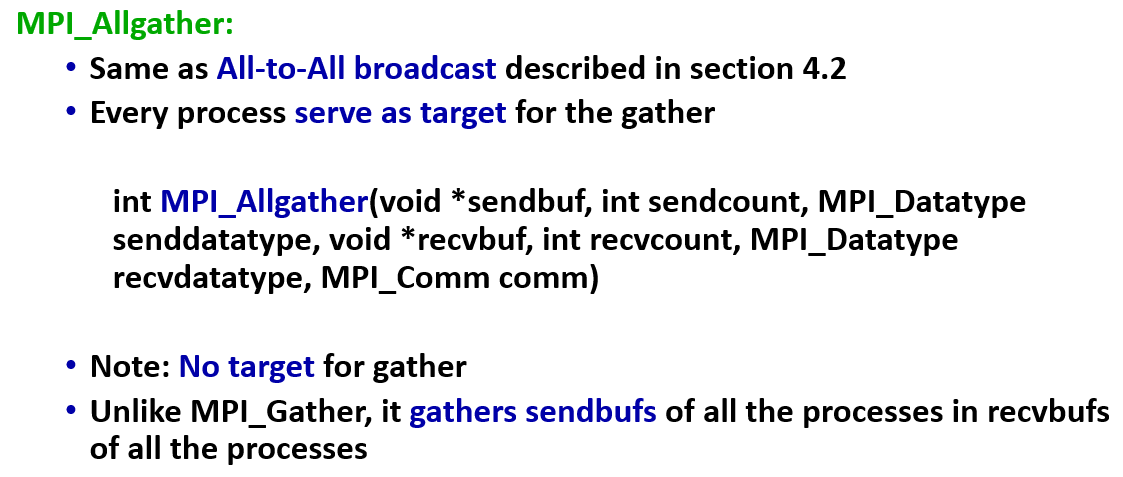
1. sendbuf: Address of the buffer holding the data to be sent from each process.
2. sendcount: Number of elements in the send buffer.
3. sendtype: Data type of elements in the send buffer (e.g., MPI\_INT, MPI\_FLOAT).
4. recvbuf: Address of the buffer where the gathered data will be stored at the root 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.

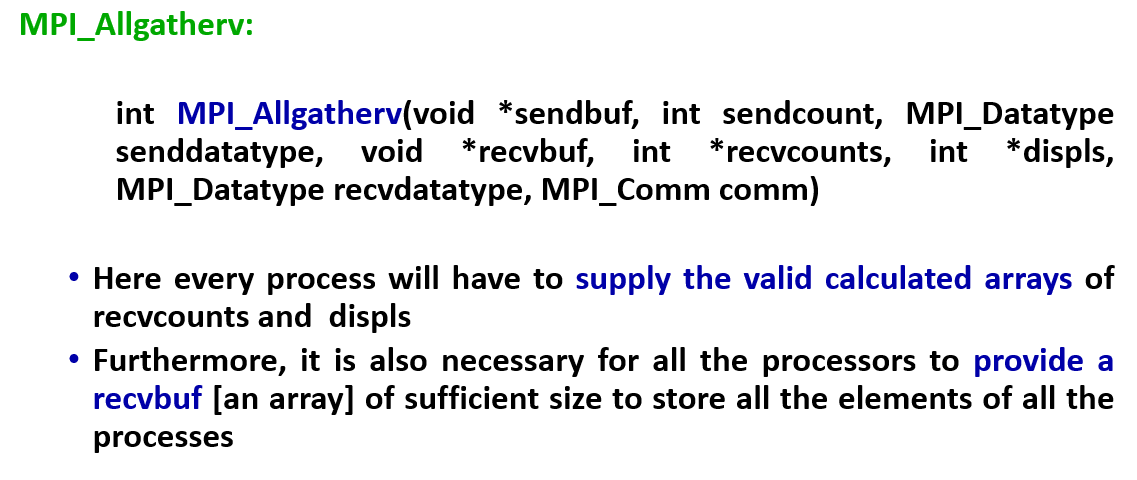












**MPI\_Gather vs. MPI\_Gatherv:**

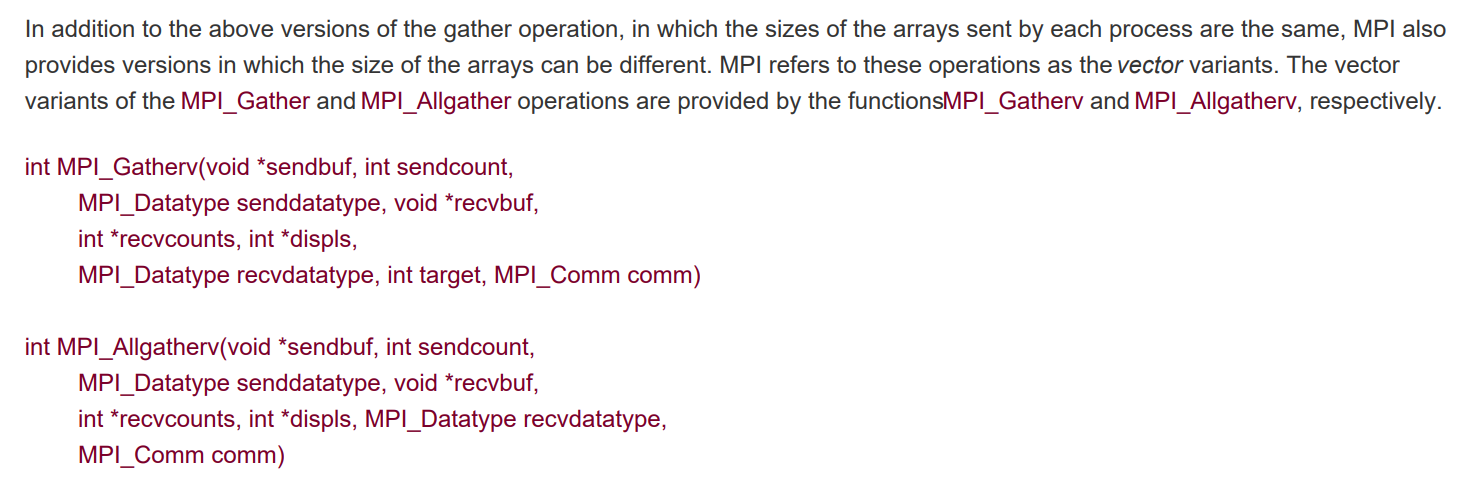
* + MPI\_Gather requires the **same amount** of data from each process.
  + MPI\_Gatherv allows **different amounts** of data from each process, specified by recvcounts and displs.

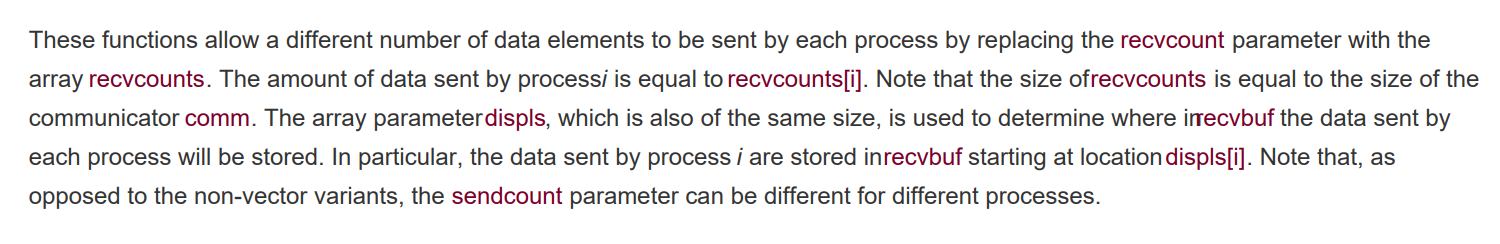
**MPI\_Allgather vs. MPI\_Allgatherv:**

* + 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:**

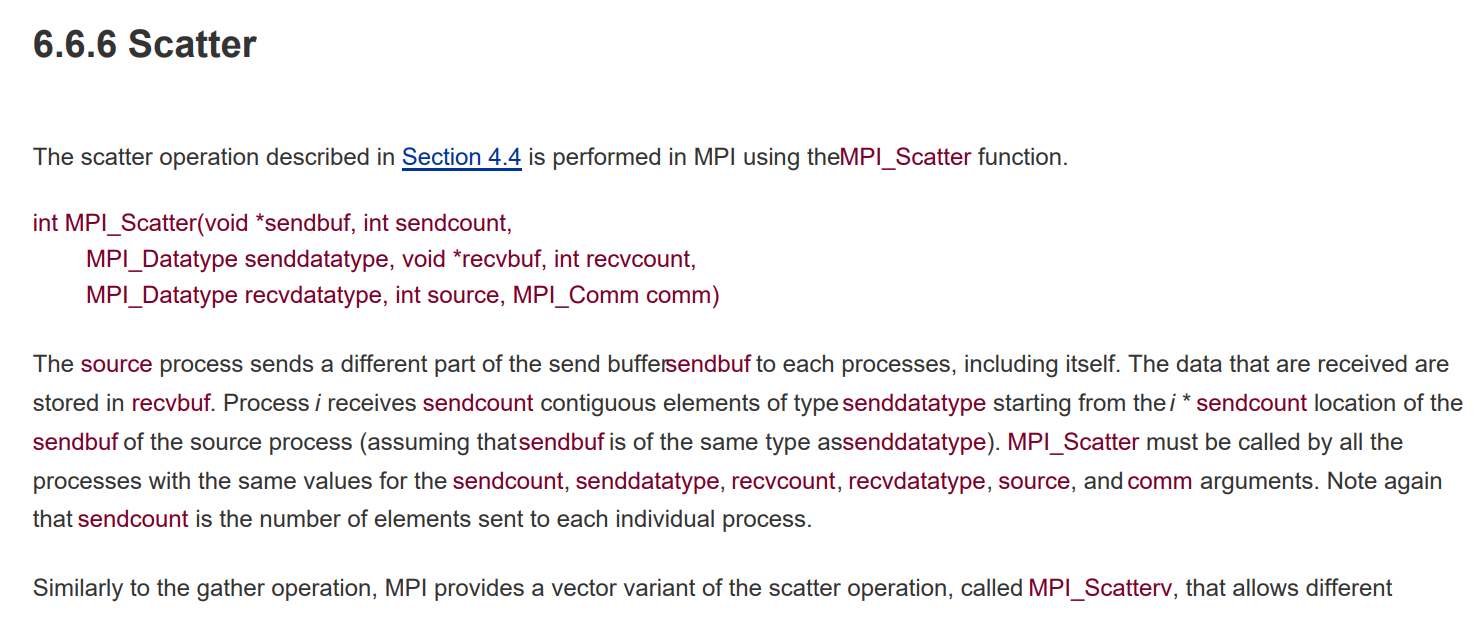
* + MPI\_Gather and MPI\_Gatherv collect data to a **single root process**.
  + MPI\_Allgather and MPI\_Allgatherv collect data and distribute the complete result to **all processes**.

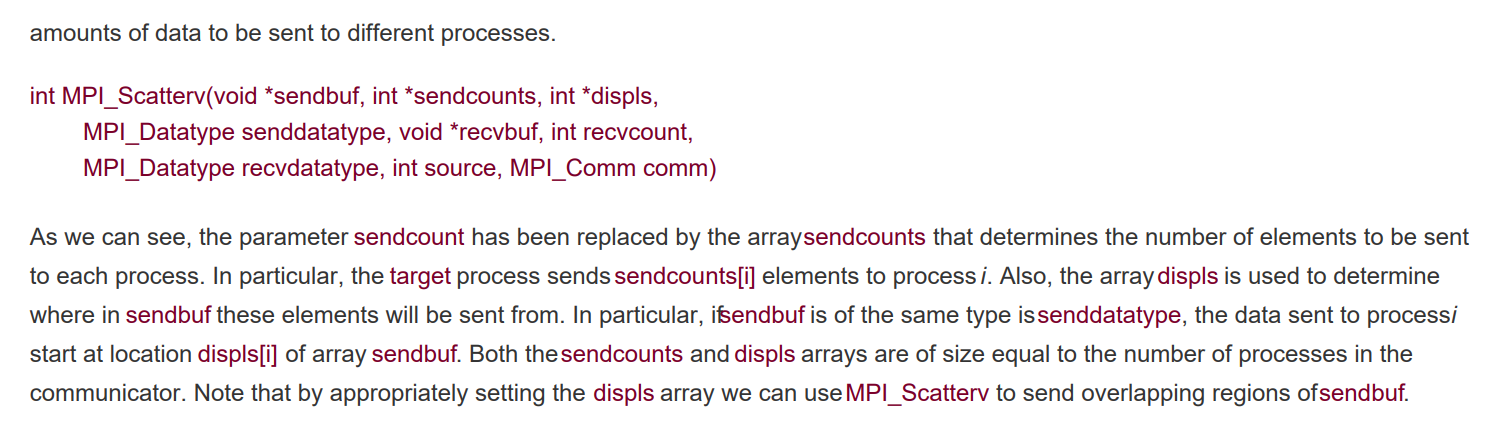


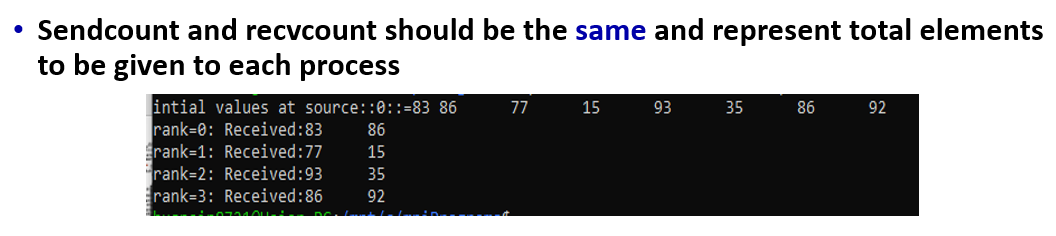


**Comparison with Similar Functions**

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.







**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.

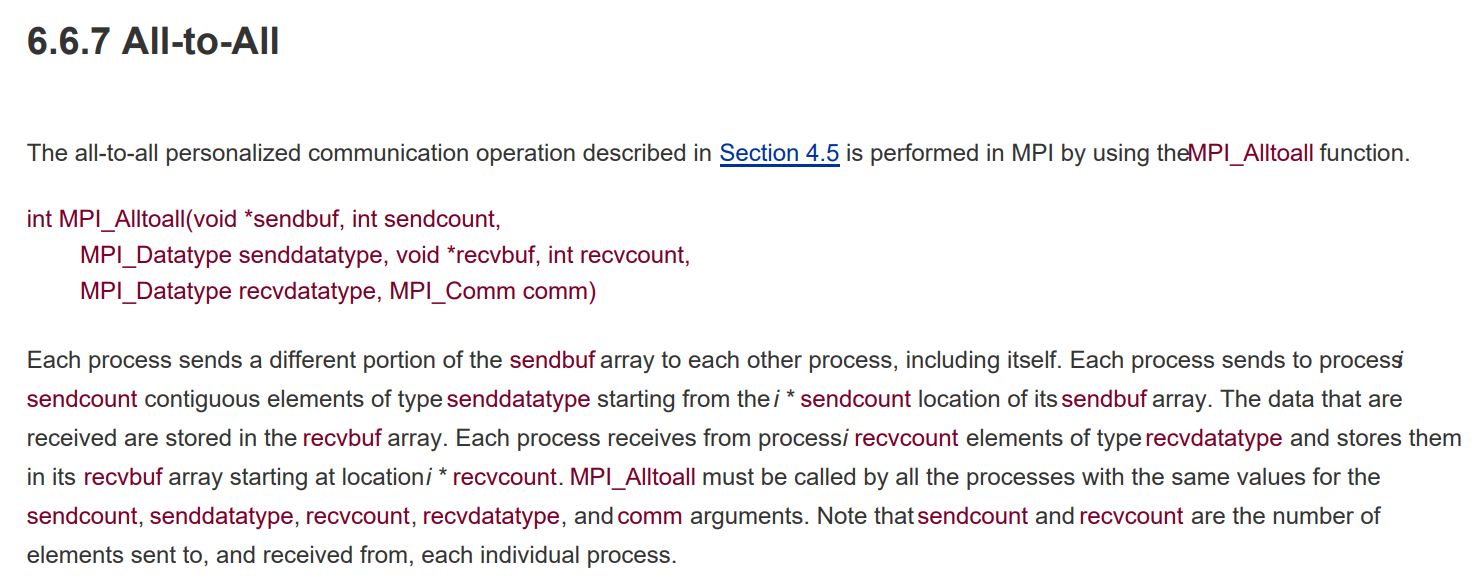
**Comparison with Similar Functions**

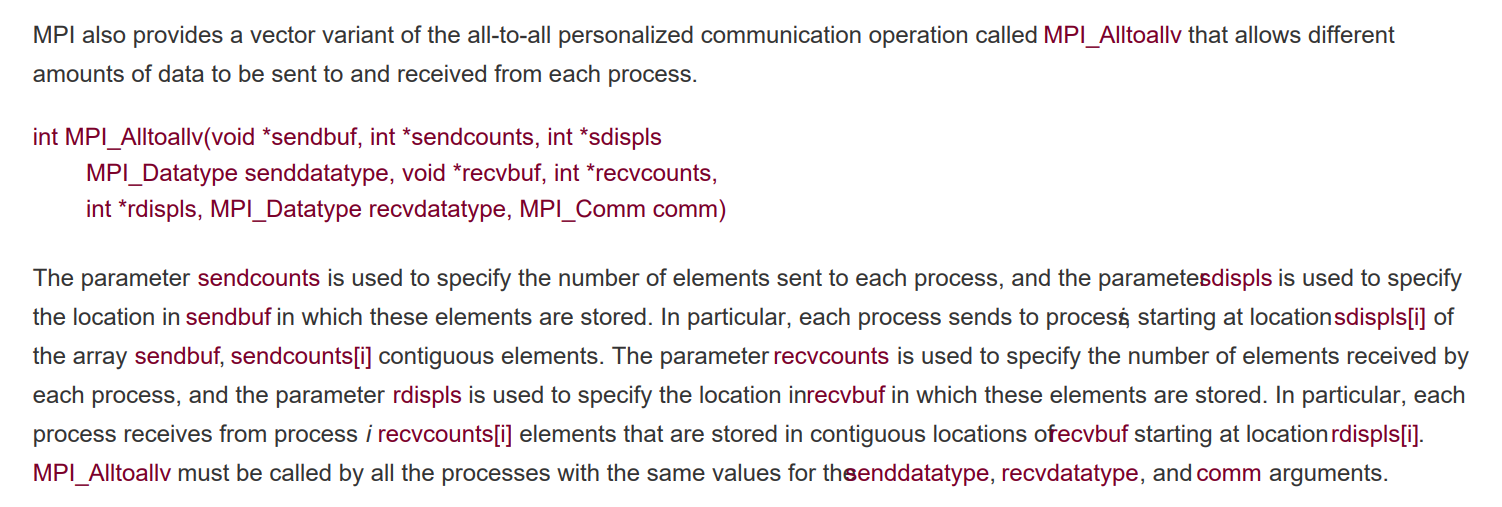
**MPI\_Scatter:**

* 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.





**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:**

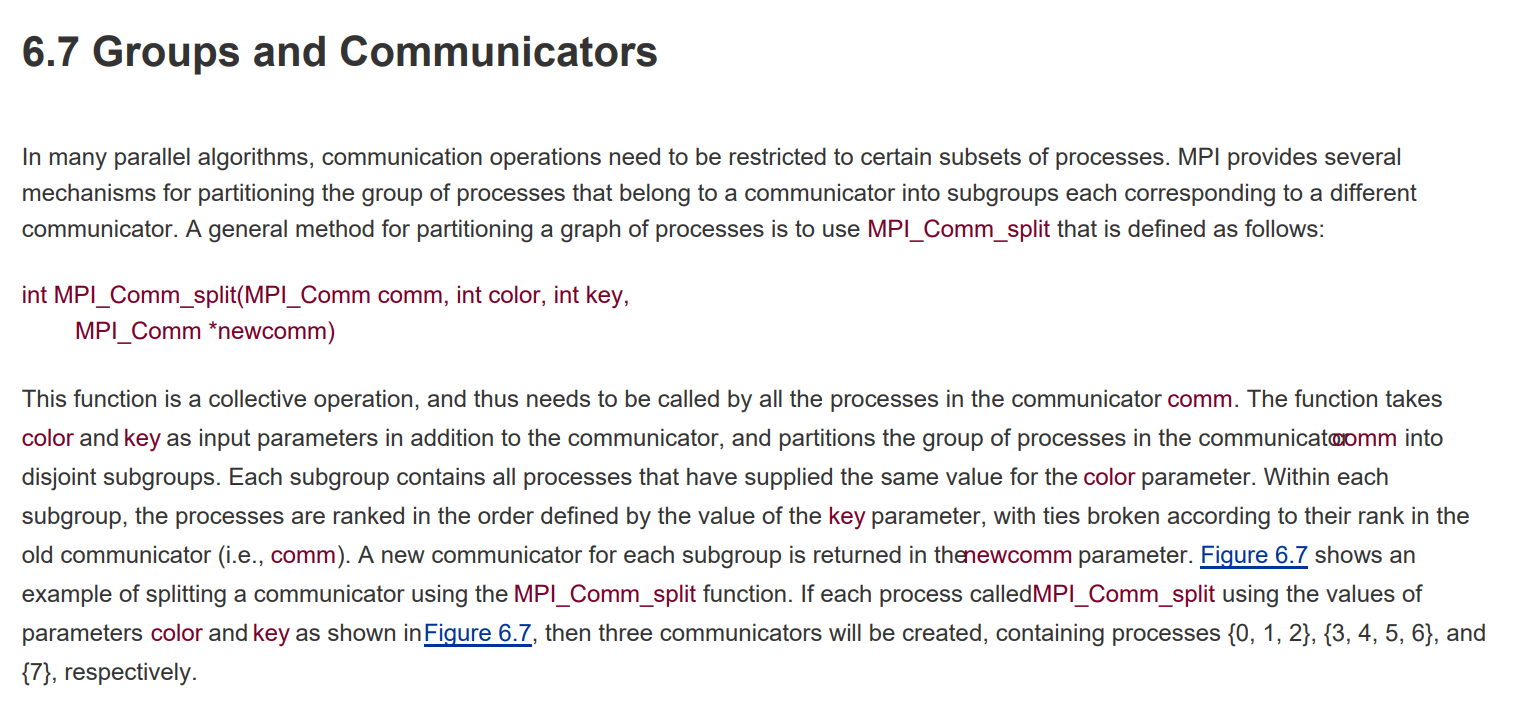
1. Each process sends and receives the same amount of data to/from all other processes.
2. Simple and straightforward, but limited to uniform data distribution.

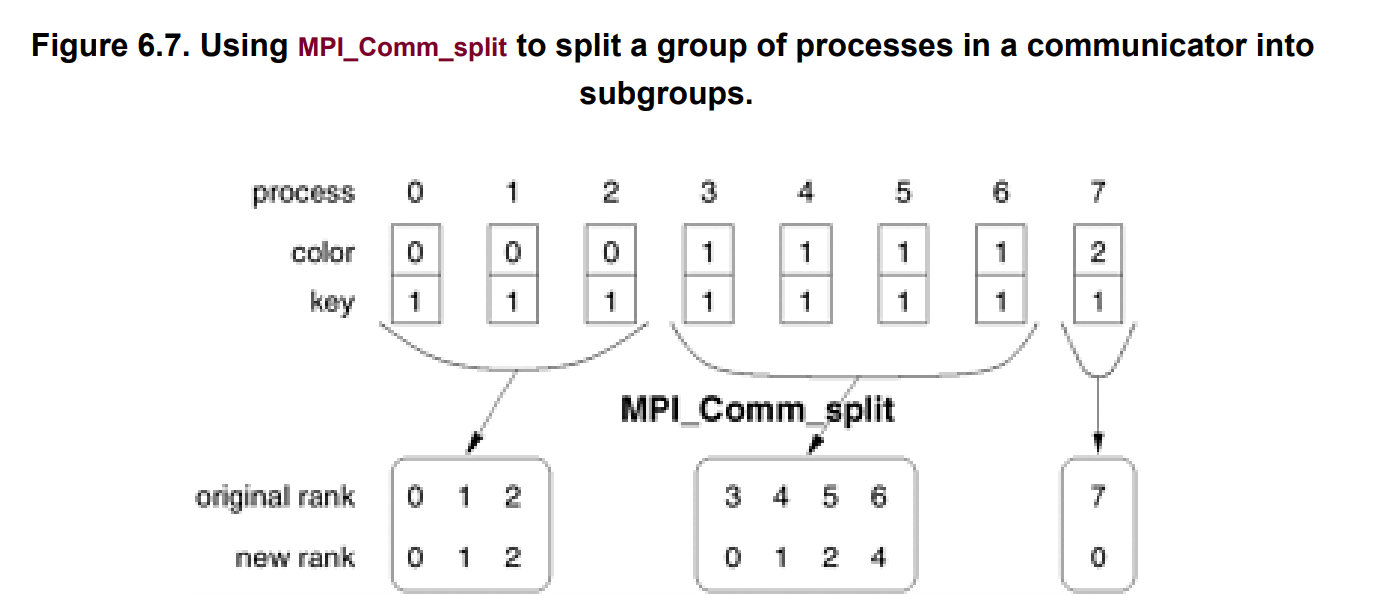
**MPI\_Alltoallv:**

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.





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.