Week # 11 incuding Lab # 8

- MPI syllabus as per course outline
- MPI Coverage started from Week # 8 (Lab # 5) see Lab slides.

MPI Syllabus (as per Course outline)

Programming Using the Message Passing Paradigm: Principles of MPI, The Building Blocks: Send and Receive Operations, Buffered and non buffered MPI, MPI interface, Starting and Terminating the MPI Library, Communicators, Querying Information, Sending and Receiving Messages, overlapping communication with computation.

Lab3: MPI installation, Communication rank and size in MPI, 1 MPI_send / 2 MPI_Recv, 3 MPI_status, 4 MPI_Tag.

(Book : Book#1)

(9th Oct,23 – 13th Oct,23)

Programming Using the Message Passing Paradigm: collective communication and computation opertaions: 7 barrier, broadcast, 8 reduction, 9 prefix, scatter, gather.

Lab4: 1 MPI Scatter, 2 Gather, 3 Bcast, 4 MPI_wait, 5 MPI_Test, 6 MPI_Allgather.

(Book : Book#1)

(16th Oct,23-20th Oct,23)

9 Scatter, gather, Bcast, MPI_wait / MPI_TEST, Allgather, reduction, prefix, barrier.

6.3.4 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)
```

```
// Example using MPI_Recv
int source_rank = 0; // The source rank of the message
int tag = 1;         // The tag of the message
MPI_Status status;
MPI_Recv(receive_buffer, count, MPI_DATATYPE, source_rank, tag, MPI_COMM_WORLD, &status);

// Now you can access status information
int received_count;
MPI_Get_count(&status, MPI_DATATYPE, &received_count);
int error_code = status.MPI_ERROR;
```

→ TAG OR SOURCE (see next slide).

Status variable in MPI_Recv

- 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 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.

Destination

```
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);
}
...
```

Send

Tag = 1

Tag = 2

Recv.

Tag = 2

Tag = 1

Source

The above example can be made safe, by rewriting it 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) { //odd process rank
   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);
}                                  -> Source.
else { // even
   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);
}
...
```

This new implementation partitions the processes into two groups. One consists of the odd-numbered processes and the other of the even-numbered processes. The odd-numbered processes perform a send followed by a receive, and the even-numbered processes perform a receive followed by a send. Thus, when an odd-numbered process calls `MPI_Send` ,the target process (which has an even number) will call `MPI_Recv` to receive that message, before attempting to send its own message.

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.

```
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_SendRecv(a, 10, MPI_INT, (myrank+1)%npes, 1,
             b, 10, MPI_INT, (myrank-1+npes)%npes, 1,
             MPI_COMM_WORLD, &status);
...
```

6.3.5 Example: Odd-Even Sort

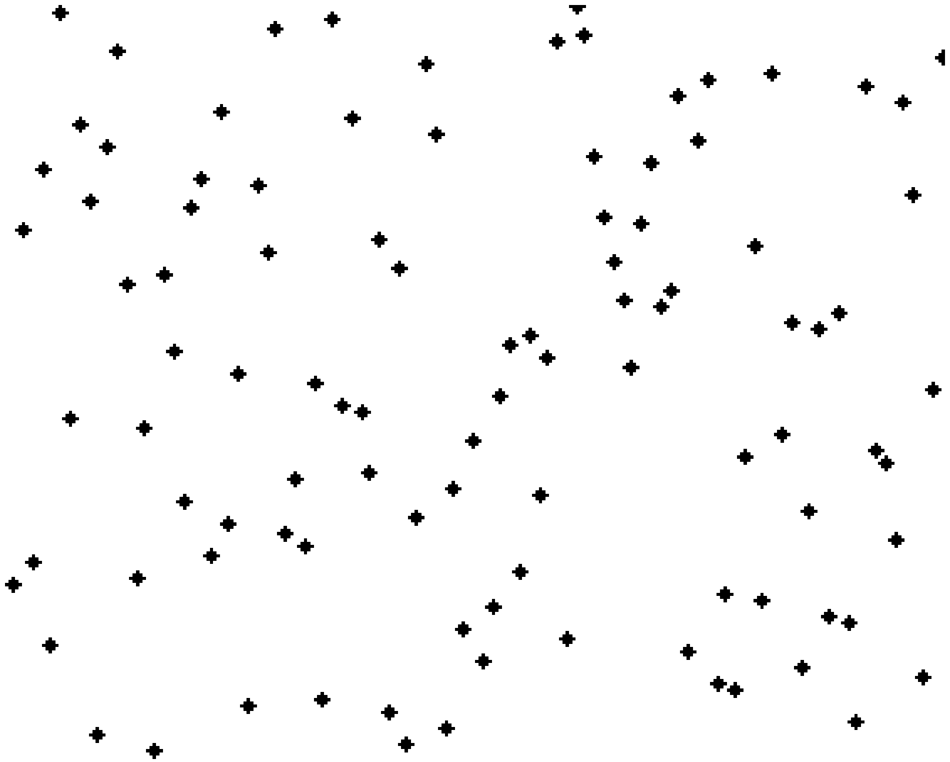
https://en.wikipedia.org/wiki/Odd–even_sort

- Odd-Even sorting algorithm sorts a sequence of *n* elements using *p* processes in a total of *p* phases.
- In each phase, each process performs a compare-split step with its right neighbor.
- The MPI program for performing the odd-even sort in parallel is shown below.
- To simplify the presentation, this program assumes that *n* is divisible by *p* .

In computing, an **odd–even sort** or **odd–even transposition sort** (also known as **brick sort** or **parity sort**) is a relatively simple sorting algorithm, developed originally for use on parallel processors with local interconnections.

working

It functions by comparing all odd/even indexed pairs of adjacent elements in the list and, if a pair is in the wrong order (the first is larger than the second) the elements are switched. The next step repeats this for even/odd indexed pairs (of adjacent elements). Then it alternates between odd/even and even/odd steps until the list is sorted.



```c
#include <stdlib.h>
#include <mpi.h> /* Include MPI's header file */

int main(int argc, char *argv[]) {
    int n;          /* The total number of elements to be sorted */
    int npes;       /* The total number of processes */
    int myrank;     /* The rank of the calling process */
    int nlocal;     /* The local number of elements, and the array that stores them */
    int *elmnts;    /* The array that stores the local elements */
    int *relmnts;   /* The array that stores the received elements */
    int oddrank;    /* The rank of the process during odd-phase communication */
    int evenrank;   /* The rank of the process during even-phase communication */
    int *wspace;    /* Working space during the compare-split operation */
    int i;
    MPI_Status status;

    /* Initialize MPI and get system information */
    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &npes);
    MPI_Comm_rank(MPI_COMM_WORLD, &myrank);

    n = atoi(argv[1]);
    nlocal = n / npes; /* Compute the number of elements to be stored locally. */
```

ction: 1-9

```c
/* Allocate memory for the various arrays */
elmnts = (int *)malloc(nlocal * sizeof(int));
relmnts = (int *)malloc(nlocal * sizeof(int));
wspace = (int *)malloc(nlocal * sizeof(int));

/* Fill-in the elmnts array with random elements */
srandom(myrank);
for (i = 0; i < nlocal; i++)
    elmnts[i] = random();

/* Sort the local elements using the built-in quicksort routine */
qsort(elmnts, nlocal, sizeof(int), IncOrder);
```

```c
/* The IncOrder function that is called by qsort is defined as follows */
int IncOrder(const void *e1, const void *e2) {
  return (*((int *)e1) - *((int *)e2));
}
```

```
/* Determine the rank of the processors that myrank needs to communicate during */
/* the odd and even phases of the algorithm */
if (myrank % 2 == 0) // even
{
    oddrank = myrank - 1;
    evenrank = myrank + 1;
}
else // odd
{
    oddrank = myrank + 1;
    evenrank = myrank - 1;
}

/* Set the ranks of the processors at the end of the linear */
if (oddrank == -1 || oddrank == npes)
    oddrank = MPI_PROC_NULL;
if (evenrank == -1 || evenrank == npes)
    evenrank = MPI_PROC_NULL;
```

```c
    /* Get into the main loop of the odd-even sorting algorithm */
    for (i = 0; i < npes - 1; i++) {
        if (i % 2 == 1) /* Odd phase */
            MPI_Sendrecv( elmnts, nlocal, MPI_INT, oddrank, 1,
                          relmnts, nlocal, MPI_INT, oddrank, 1, MPI_COMM_WORLD, &status);
        else /* Even phase */
            MPI_Sendrecv( elmnts, nlocal, MPI_INT, evenrank, 1,
                          relmnts, nlocal, MPI_INT, evenrank, 1, MPI_COMM_WORLD, &status);

        CompareSplit(nlocal, elmnts, relmnts, wspace, myrank < status.MPI_SOURCE);
    }

    free(elmnts); free(relmnts); free(wspace);
    MPI_Finalize();
}
```

OR

Who send the message

```c
CompareSplit(int nlocal, int *elmnts, int *relmnts, int *wspace, int keepsmall) {
    int i, j, k;

    for (i = 0; i < nlocal; i++)
        wspace[i] = elmnts[i]; /* Copy the elmnts array into the wspace array */

    if (keepsmall) { /* Keep the nlocal smaller elements */
        for (i = j = k = 0; k < nlocal; k++) {
            if (j == nlocal || (i < nlocal && wspace[i] < relmnts[j]))
                elmnts[k] = wspace[i++];
            else
                elmnts[k] = relmnts[j++];
        }
    }
    else { /* Keep the nlocal larger elements */
        for (i = k = nlocal - 1, j = nlocal - 1; k >= 0; k--) {
            if (j == 0 || (i >= 0 && wspace[i] >= relmnts[j]))
                elmnts[k] = wspace[i--];
            else
                elmnts[k] = relmnts[j--];
        }
    }
}
```

True
myrank < status.SOURCE

FALSE
myrank > status.SOURCE

6.5 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. For example, consider Cannon's matrix-matrix multiplication program described in Program 6.2. During each iteration of its main computational loop (lines 47– 57), it first computes the matrix multiplication of the sub-matrices stored in `a` and `b`, and then shifts the blocks of `a` and `b`, using `MPI_Sendrecv_replace` which blocks until the specified matrix block has been sent and received by the corresponding processes. In each iteration, each process spends $O(n^3/p^{1.5})$ time for performing the matrix-matrix multiplication and $O(n^2/p)$ time for shifting the blocks of matrices A and B. Now, since the blocks of matrices A and B do not change as they are shifted among the processors, it will be preferable if we can overlap the transmission of these blocks with the computation for the matrix-matrix multiplication, as many recent distributed-memory parallel computers have dedicated communication controllers that can perform the transmission of messages without interrupting the CPUs.

6.5.1 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.

The calling sequences of `MPI_Isend` and `MPI_Irecv` are the following:

```
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)
```

Note that these functions have similar arguments as the corresponding blocking send and receive functions. The main difference is that they take an additional argument `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)
```

Test

`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.

wait

The `MPI_Wait` function blocks until the non-blocking operation identified by `request` completes. In that case it deal-locates the `request` object, sets it to `MPI_REQUEST_NULL`, and returns information about the completed operation in the `status` object.

MPI_Wait and MPI_Test

```
MPI_Request send_request, recv_request;
int data_to_send = 42;
int received_data;
int tag = 0;
MPI_Status send_status, recv_status;

MPI_Isend(&data_to_send, 1, MPI_INT, dest_rank, tag, MPI_COMM_WORLD, &send_request);
MPI_Irecv(&received_data, 1, MPI_INT, source_rank, tag, MPI_COMM_WORLD, &recv_request);
```

```
MPI_Wait(&send_request, &send_status);
MPI_Wait(&recv_request, &recv_status);
```

```
MPI_Test(&send_request, &send_complete, &send_status);
MPI_Test(&recv_request, &recv_complete, &recv_status);

if (send_complete)
{
    // Send operation is complete
}

if (recv_complete)
{
    // Receive operation is complete
}
```

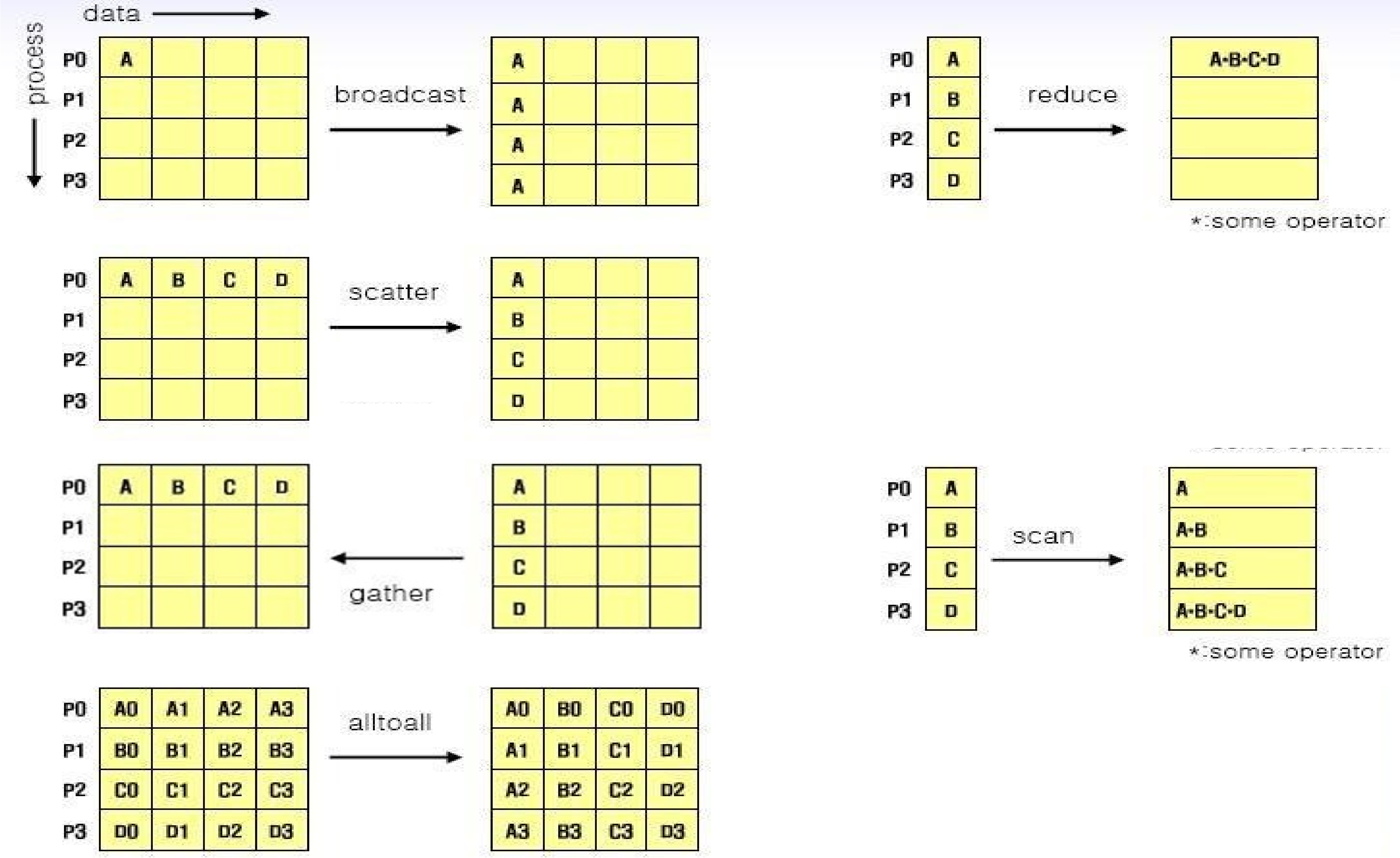
Avoiding Deadlocks By using non-blocking communication operations we can remove most of the deadlocks associated with their blocking counterparts. For example, as we discussed in Section 6.3 the following piece of code is not safe.

```
int a[10], b[10], myrank;
MPI_Status status;
MPI_Request requests[2];
...
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_Irecv(b, 10, MPI_INT, 0, 2, &requests[0], MPI_COMM_WORLD);
  MPI_Irecv(a, 10, MPI_INT, 0, 1, &requests[1], MPI_COMM_WORLD);
}
...
```

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.

6.6 Collective Communication and Computation Operations

MPI provides an extensive set of functions for performing many commonly used collective communication operations. In particular, the majority of the basic communication operations described in Chapter 4 are supported by MPI. 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 (i.e., it is possible for a processor to go past its call for the collective communication operation even before other processes have reached it), it acts 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. 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. For most collective communication operations, MPI provides two different variants. The first transfers equal-size data to or from each process, and the second transfers data that can be of different sizes.


data
process
P0
P1
P2
P3
A
broadcast
A
A
A
A
reduce
B
C
D
A*B*C*D
*:some operator
scatter
gather
scan
A*B
A*B*C
A*B*C*D
*:some operator
alltoall
A0 A1 A2 A3
B0 B1 B2 B3
C0 C1 C2 C3
D0 D1 D2 D3
A0 B0 C0 D0
A1 B1 C1 D1
A2 B2 C2 D2
A3 B3 C3 D3

6.6.1 Barrier

The barrier synchronization operation is performed in MPI using the `MPI_Barrier` function.

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

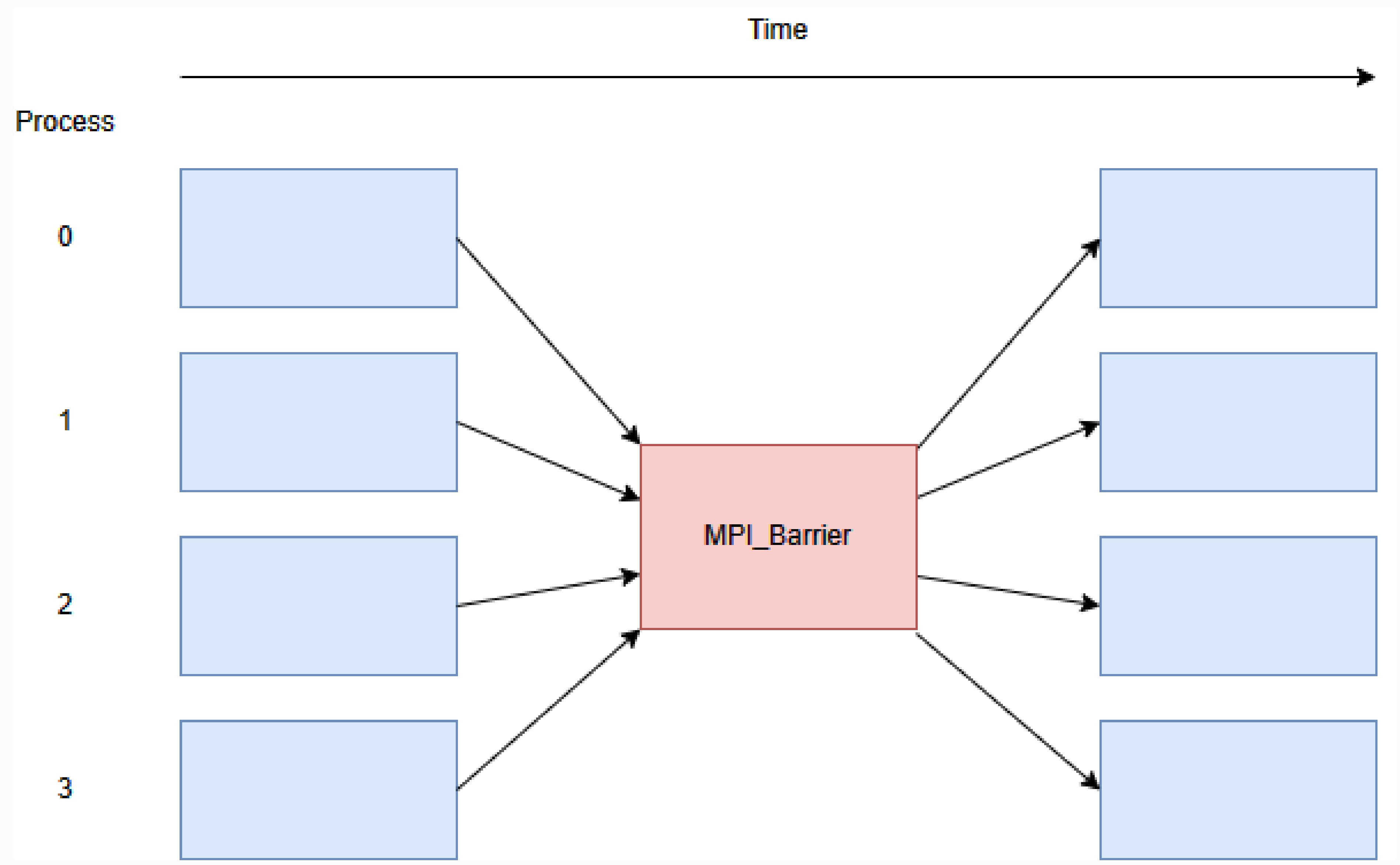
The only argument of `MPI_Barrier` is the communicator that defines the group of processes that are synchronized. The call to `MPI_Barrier` returns only after all the processes in the group have called this function.

6.6.2 Broadcast

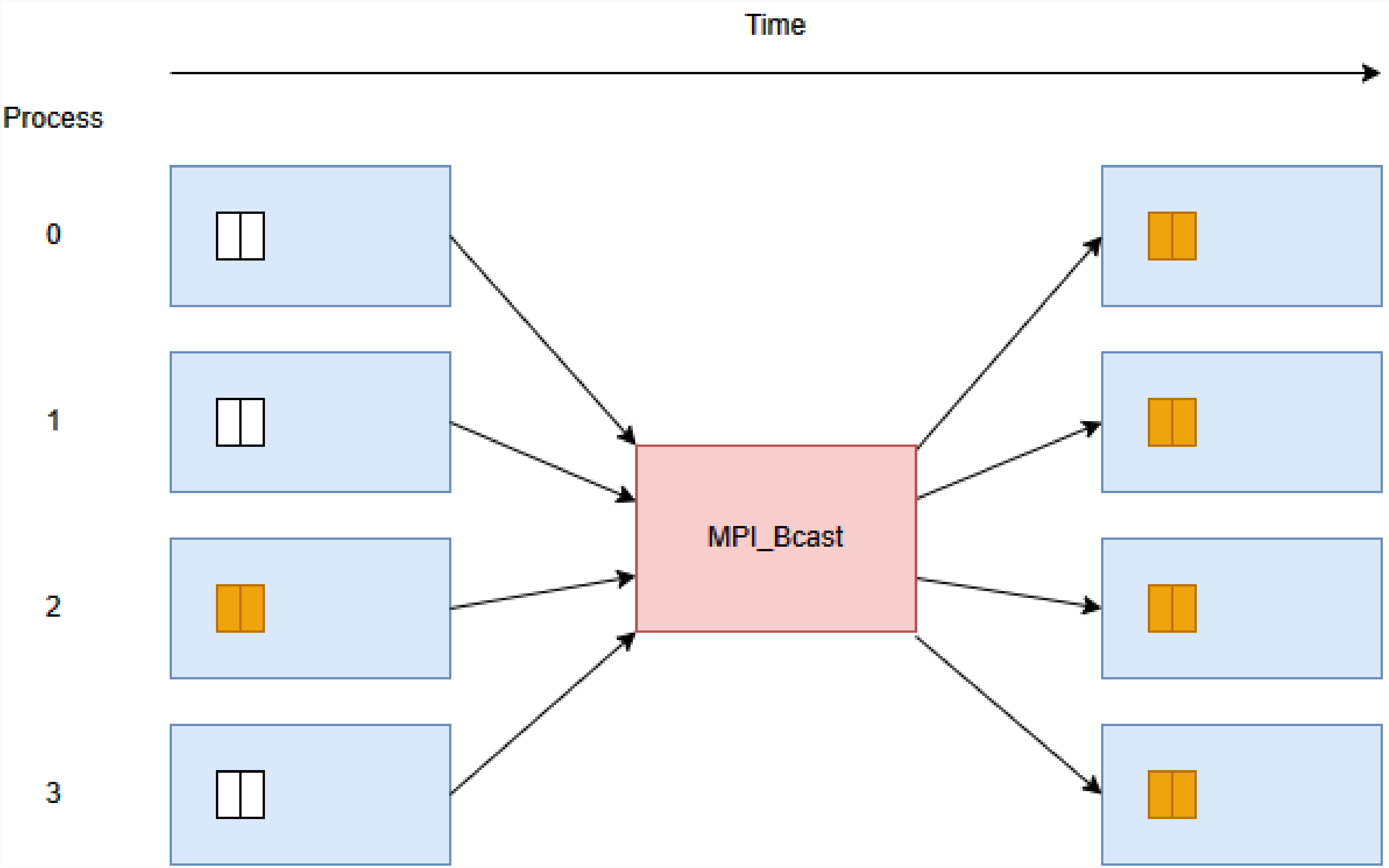
The one-to-all broadcast operation described in Section 4.1 is performed in MPI using the `MPI_Bcast` function.

```
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.



All ranks in the communicator reach the barrier before any continue past it



After the call, all ranks in the communicator agree on the two values sent.

6.6.3 Reduction

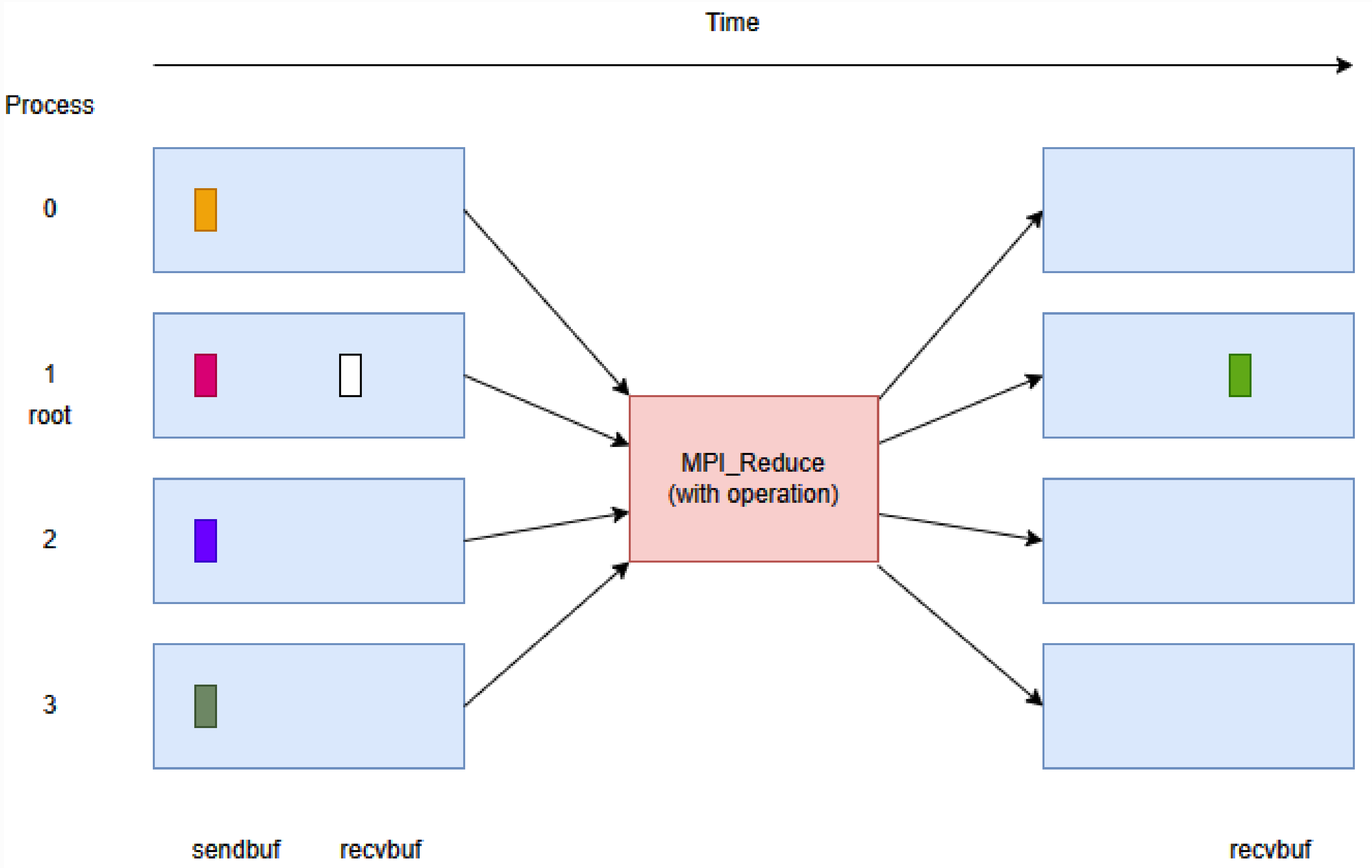
The all-to-one reduction operation described in Section 4.1 is performed in MPI using the `MPI_Reduce` function.

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

`MPI_Reduce` 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.

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



After the call, the root rank has a value computed by combining a value from each other rank in the communicator with an operation.

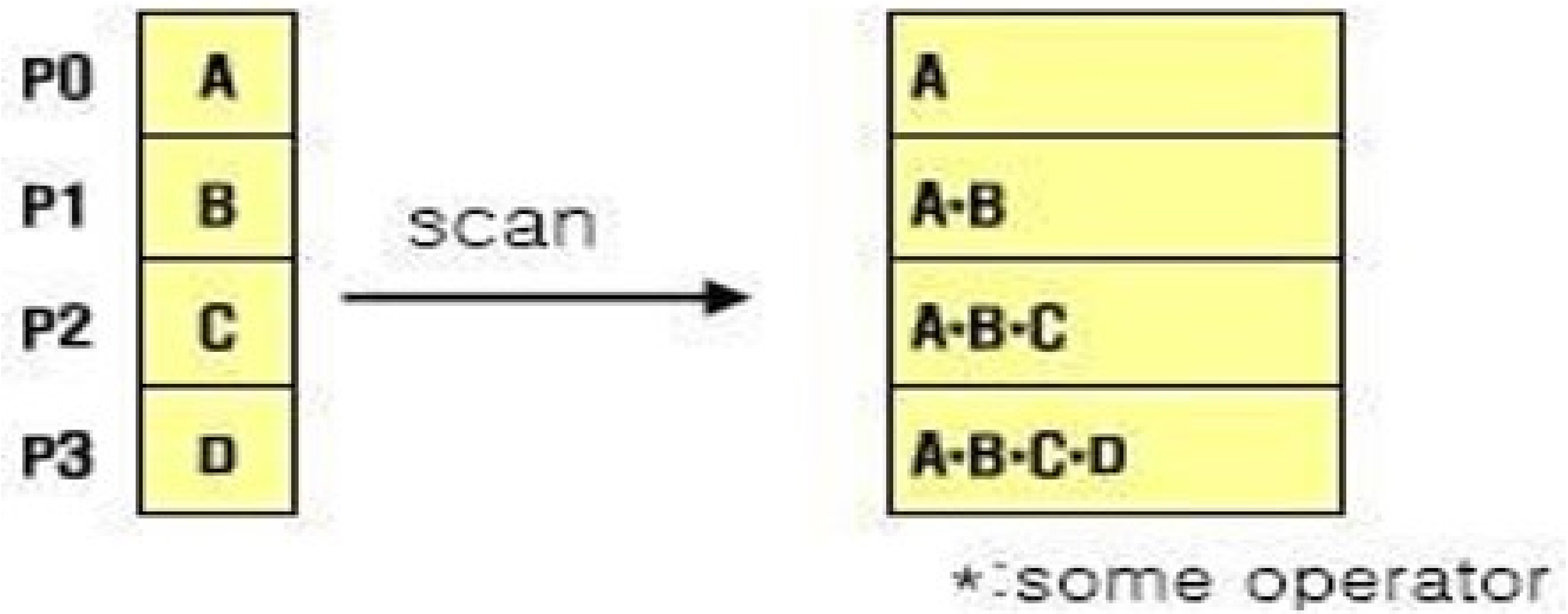
6.6.4 Prefix

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



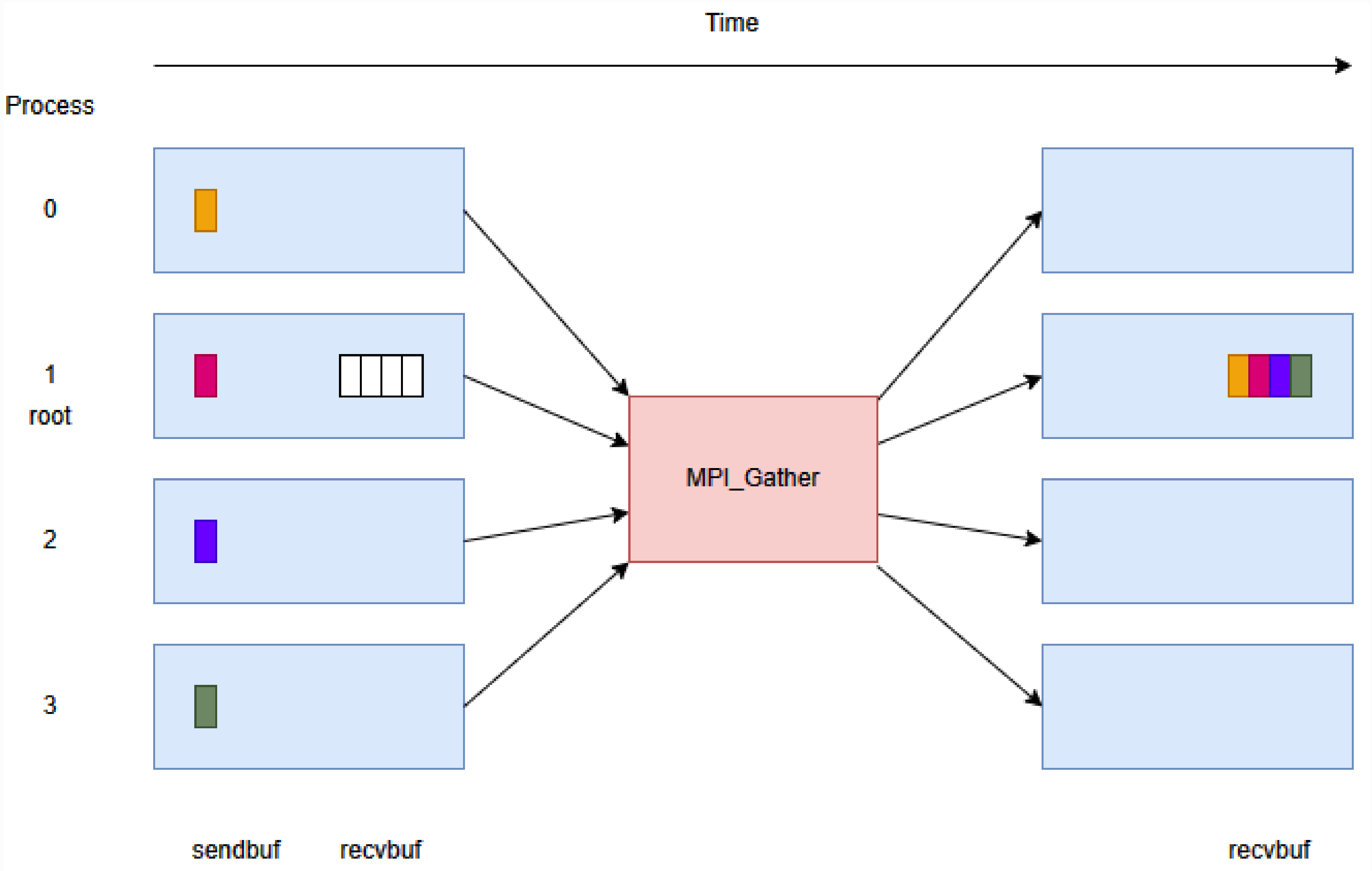
6.6.5 Gather

The gather operation described in Section 4.4 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.



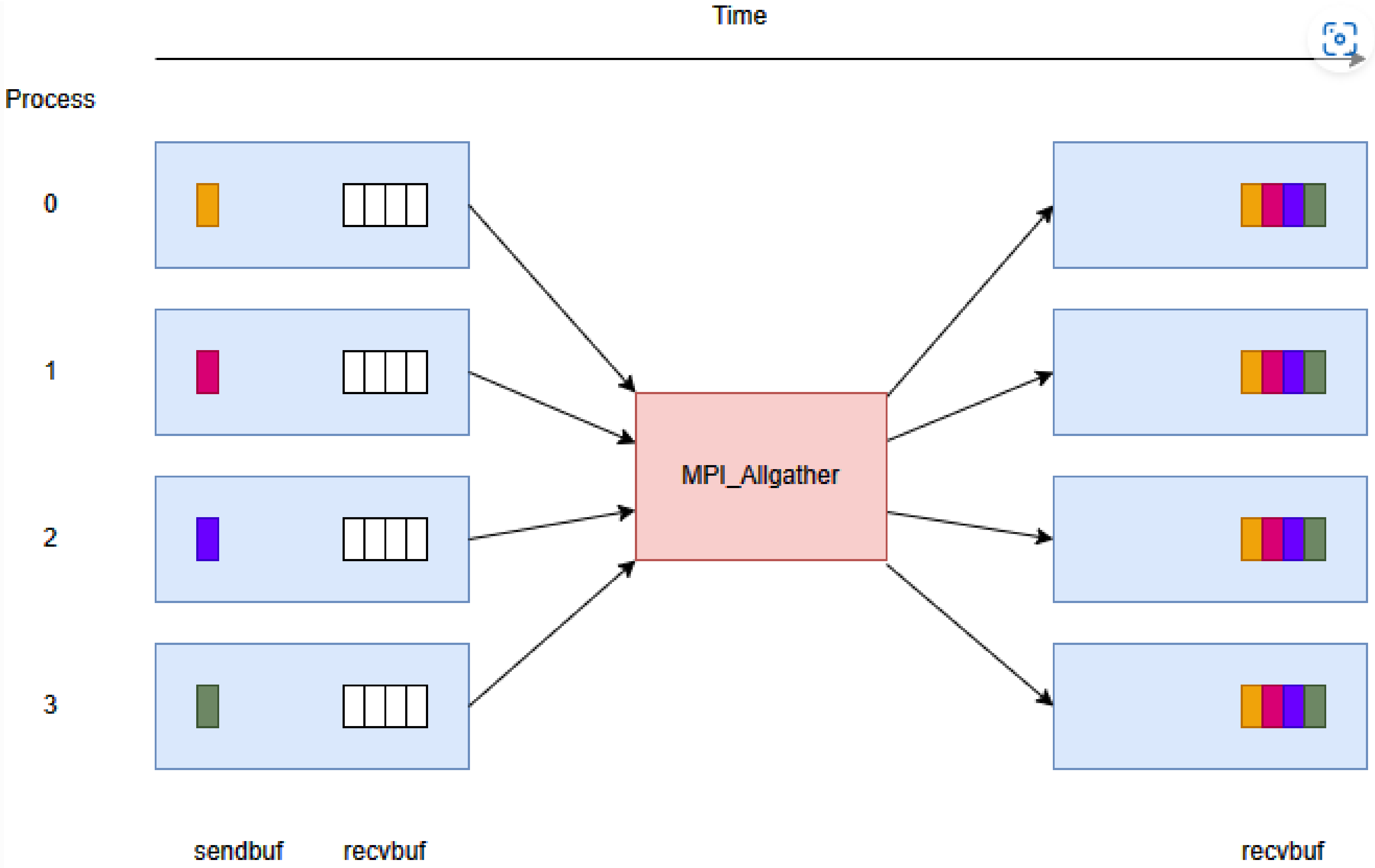
After the call, the root rank has one value from each other rank in the communicator, ordered by rank number.

6.6.5 Gather

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.



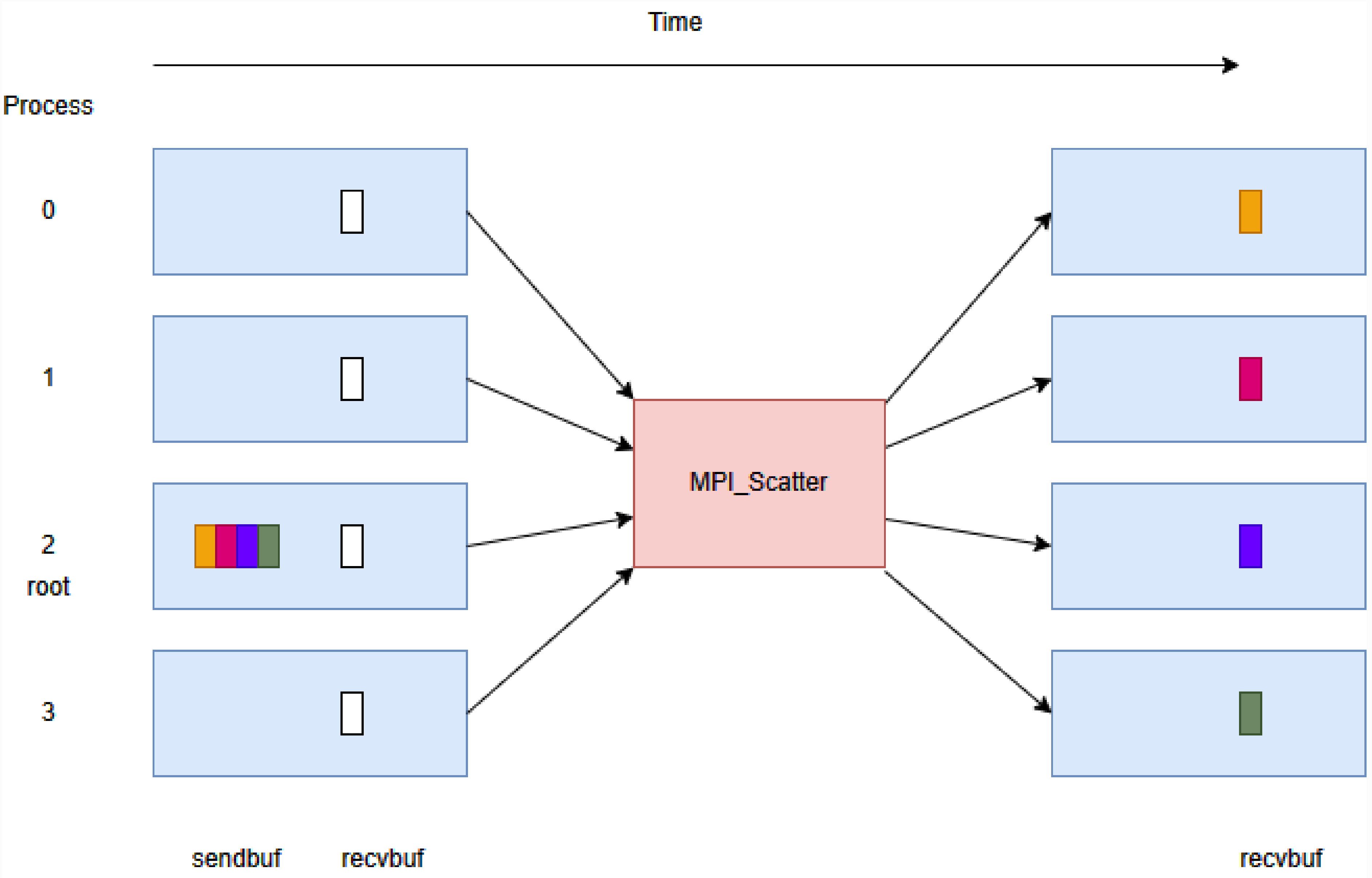
After the call, all ranks have one value from each other rank in the communicator, ordered by rank number.

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.



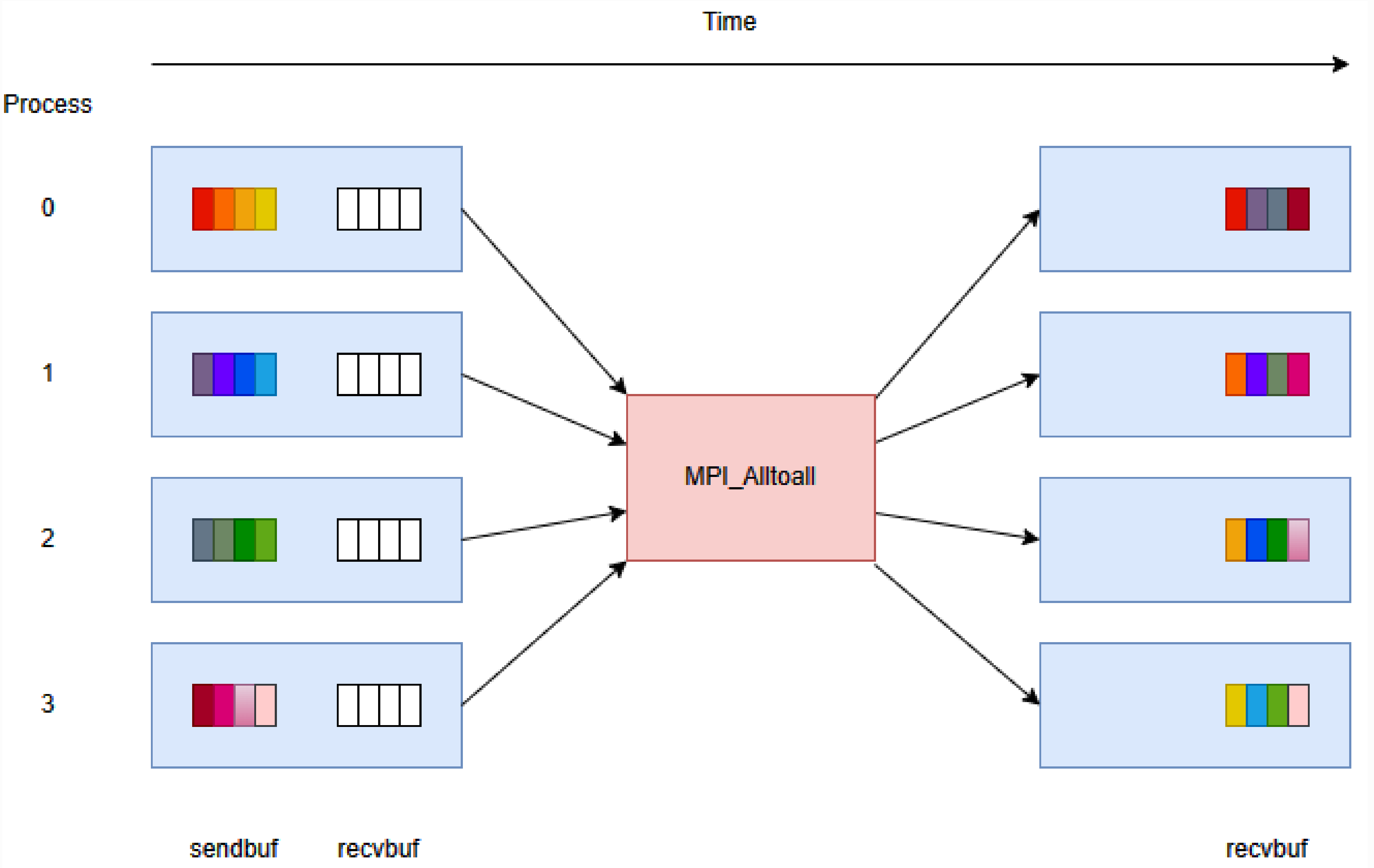
After the call, all ranks in the communicator have the one value sent from the root rank, ordered by rank number.

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.



After the call, all ranks have one value from each other rank in the communicator, ordered by rank number.