

Message-Passing Computing

More MPI routines:

Collective routines

Synchronous routines

Non-blocking routines

Collective message-passing routines

Have routines that send message(s) to a group of processes or receive message(s) from a group of processes

Higher efficiency than separate point-to-point routines although routines not absolutely necessary.

Collective Communication

Involves set of processes, defined by an intra-communicator. Message tags not present. Principal collective operations:

- `MPI_Bcast()` - Broadcast from root to all other processes
- `MPI_Gather()` - Gather values for group of processes
- `MPI_Scatter()` - Scatters buffer in parts to group of processes
- `MPI_Alltoall()` - Sends data from all processes to all processes
- `MPI_Reduce()` - Combine values on all processes to single value
- `MPI_Reduce_scatter()` - Combine values and scatter results
- `MPI_Scan()` - Compute prefix reductions of data on processes

- `MPI_Barrier()` - A means of synchronizing processes by stopping each one until they all have reached a specific “barrier” call.

Collective routines

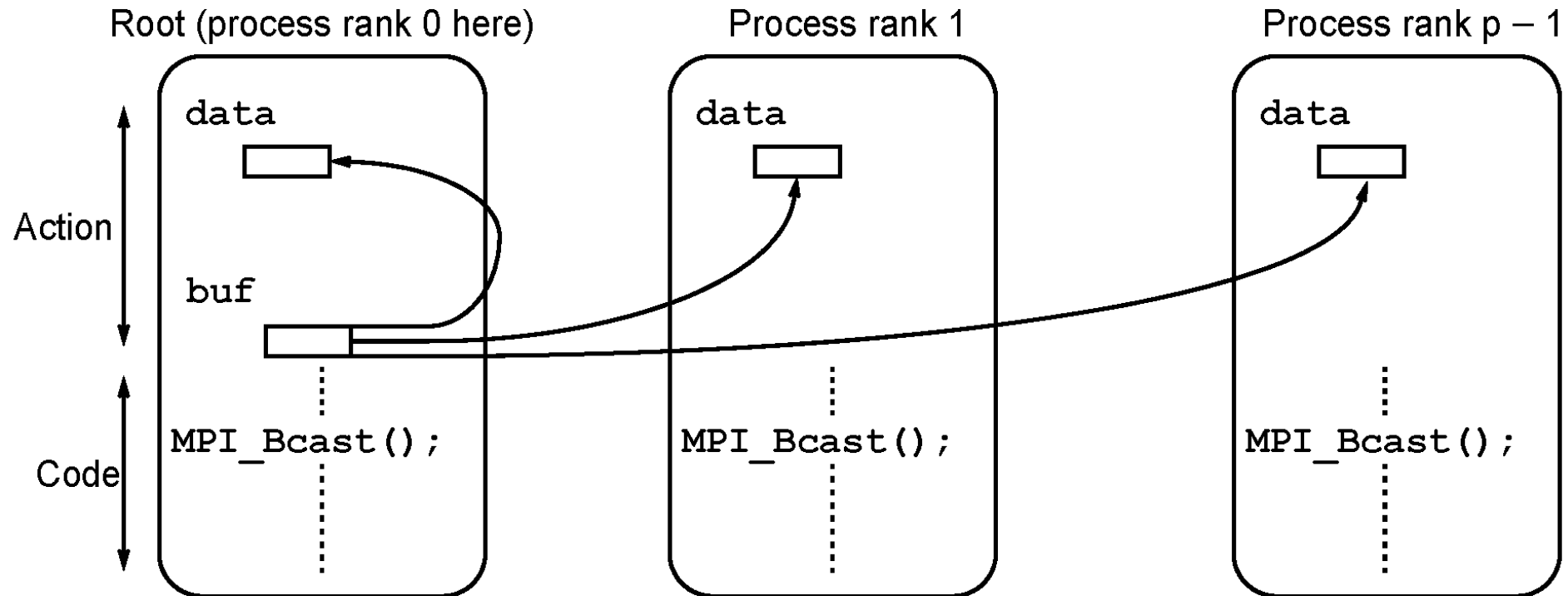
General features

- Performed on a group of processes, identified by a communicator
- Substitute for a sequence of point-to-point calls
- Communications are locally blocking
- Synchronization is *not* guaranteed (implementation dependent)
- Some routines use a *root* process to originate or receive all data
- Data amounts must exactly match
- Many variations to basic categories
- No message tags are needed

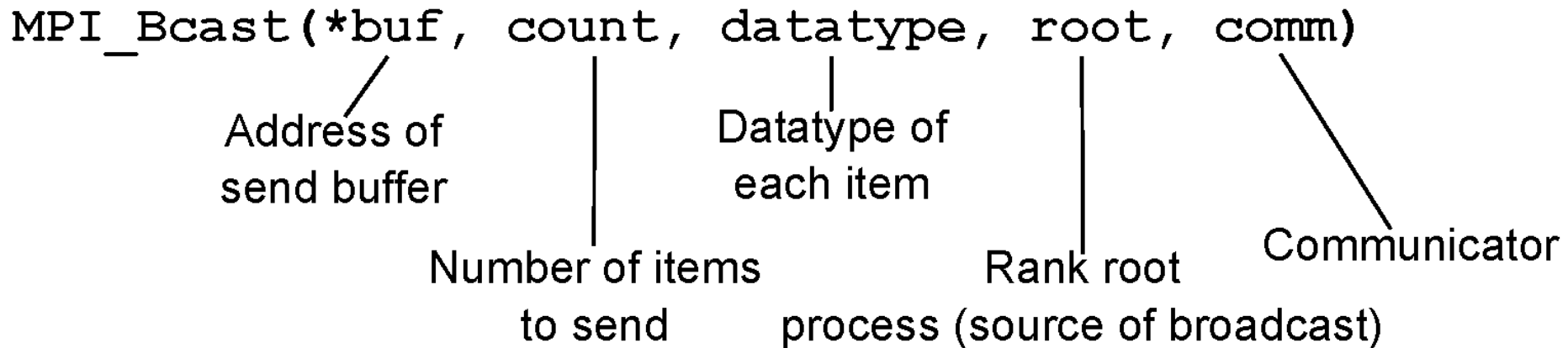
MPI broadcast operation

Sending same message to all processes in communicator.

Multicast - sending same message to defined group of processes.

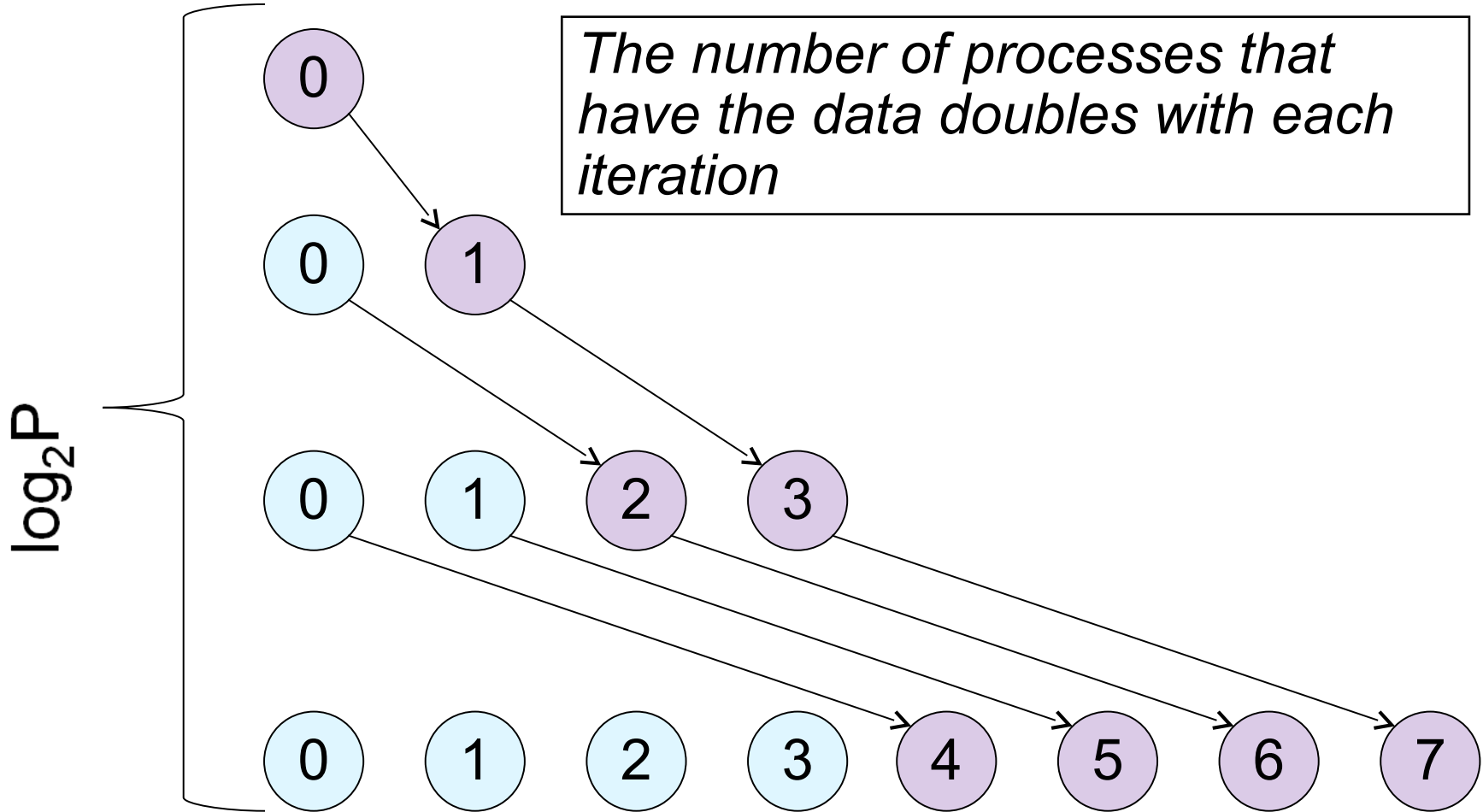


MPI_Bcast parameters



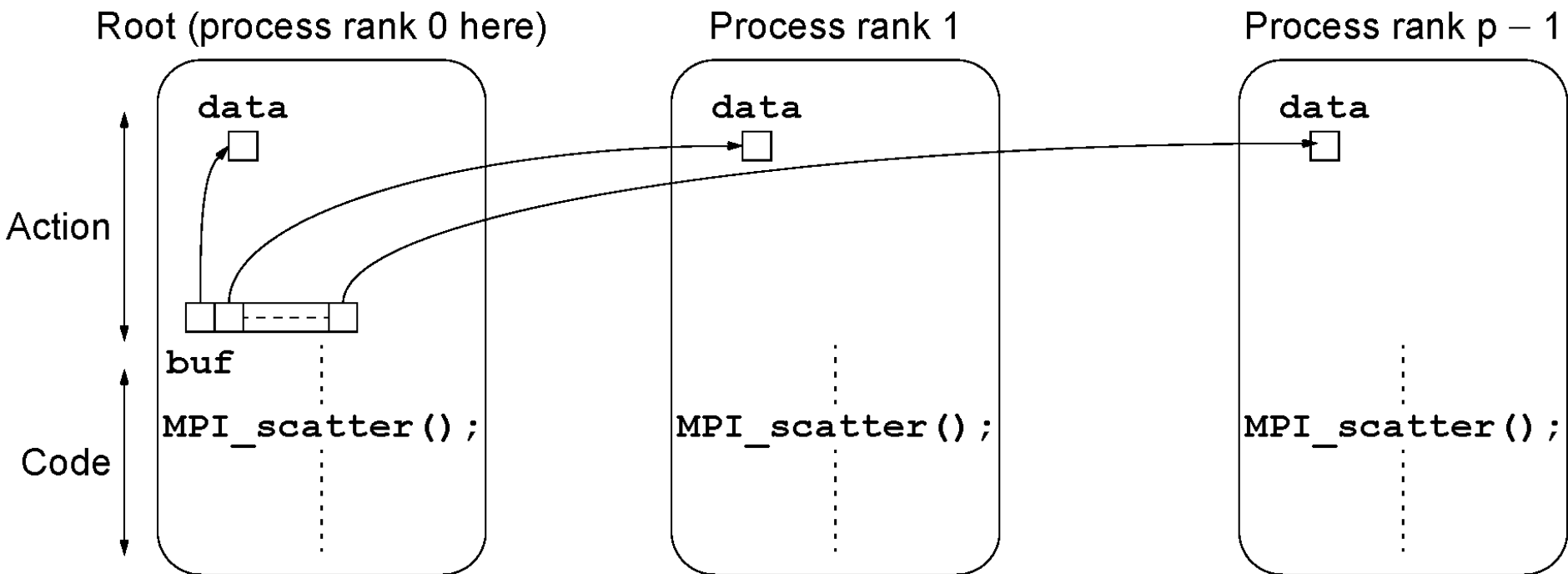
All processes in the Communicator must call the MPI_Bcast with the same parameters

Likely MPI_Bcast implementation

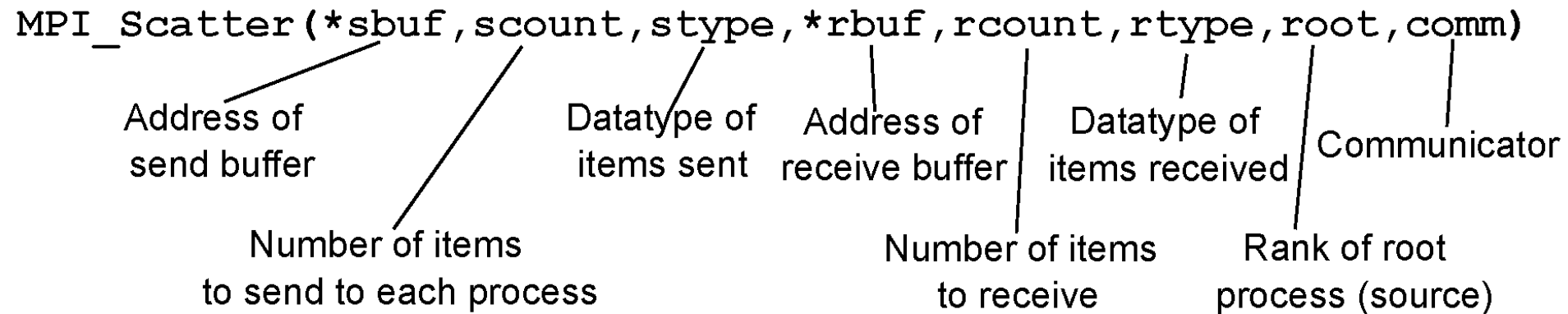


Basic MPI scatter operation

Sending each element of an array in root process to a separate process. Contents of i th location of array sent to i th process.



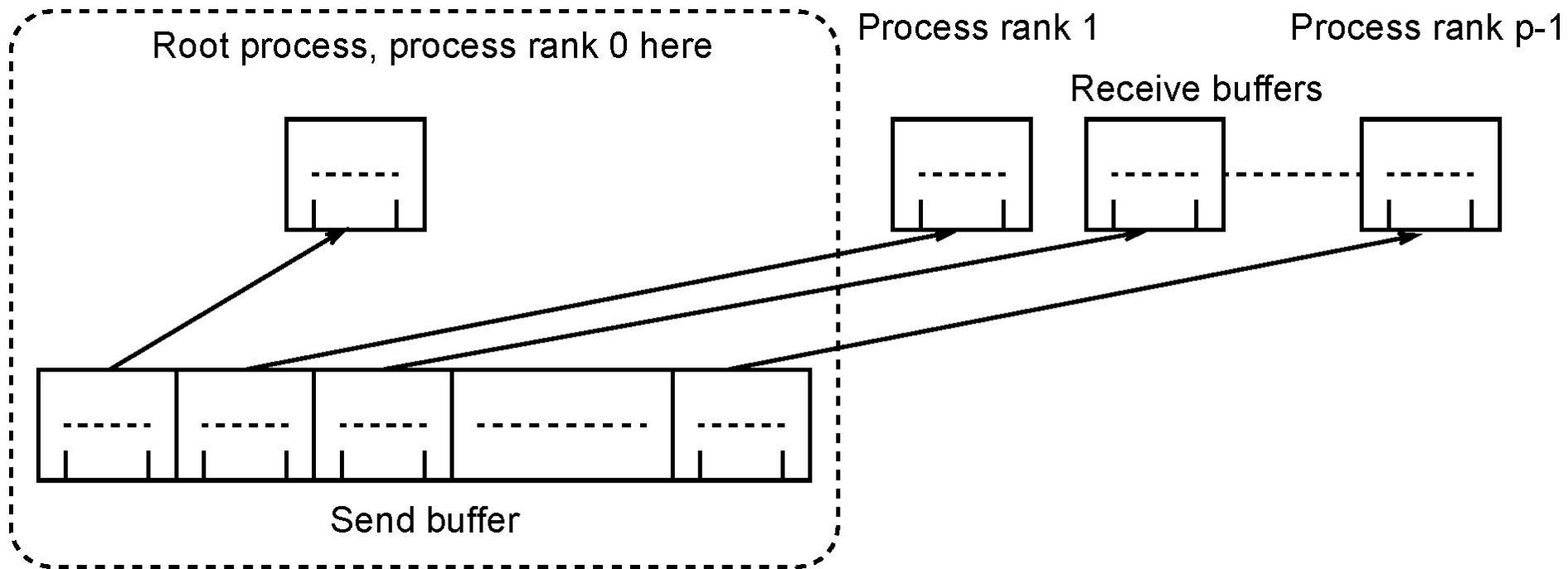
MPI scatter parameters



All processes in the Communicator must call the MPI_Scatter with the same parameters

- Simplest scatter would be as illustrated which one element of an array is sent to different processes.
- Extension provided in the `MPI_Scatter()` routine is to send a fixed number of **contiguous elements** to each process.

Scattering contiguous groups of elements to each process



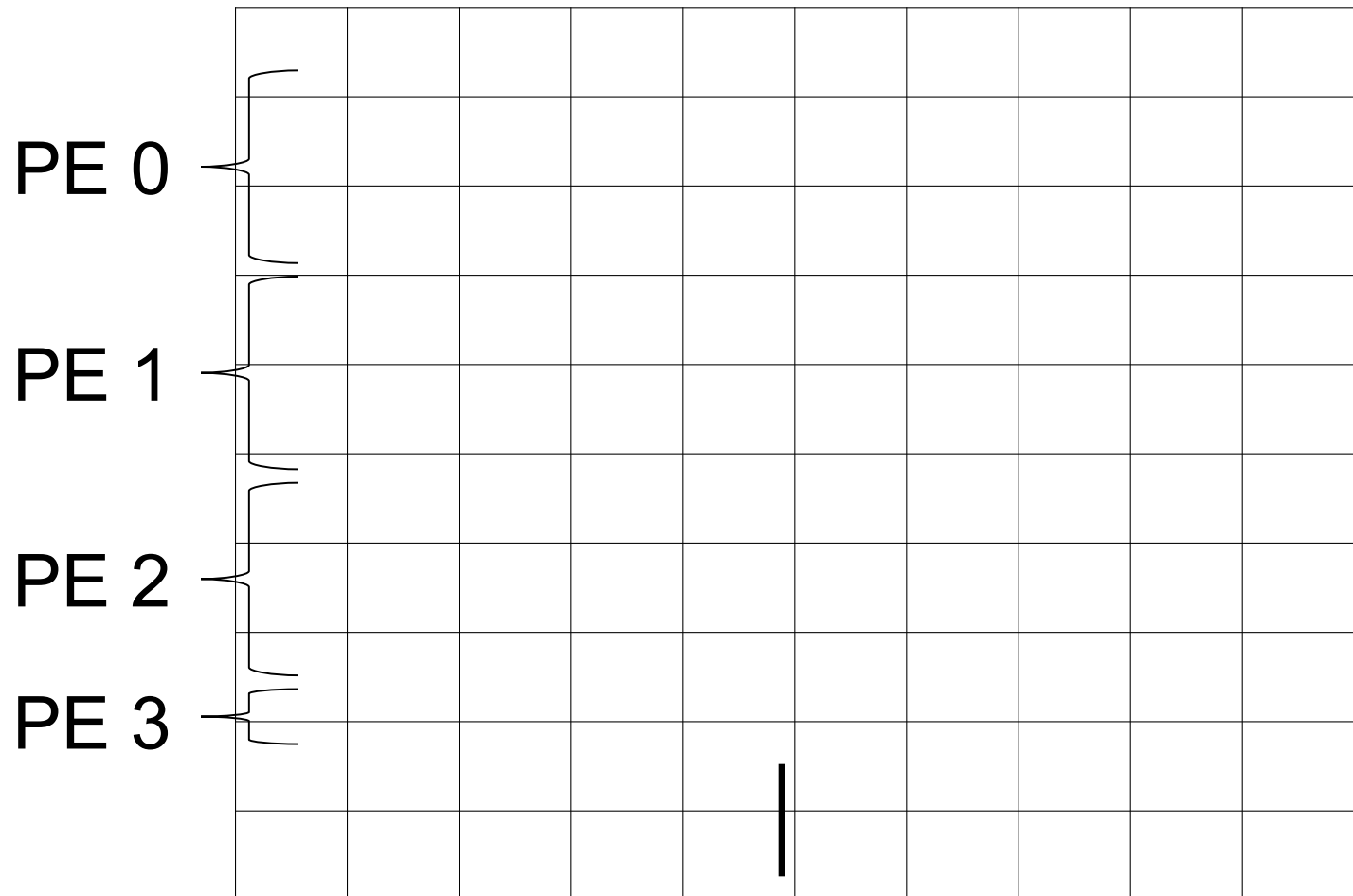
Example

In the following code, size of send buffer is given by $100 * \langle \text{number of processes} \rangle$ and 100 contiguous elements are send to each process:

```
main (int argc, char *argv[]) {  
    int size, *sendbuf, recvbuf[100];          /* for each process */  
    MPI_Init(&argc, &argv);      /* initialize MPI */  
    MPI_Comm_size(MPI_COMM_WORLD, &size);  
    sendbuf = (int *)malloc(size*100*sizeof(int));  
    .  
    MPI_Scatter(sendbuf,100,MPI_INT,recvbuf,100,MPI_INT,0,  
    MPI_COMM_WORLD);  
    .  
    MPI_Finalize();      /* terminate MPI */  
}
```

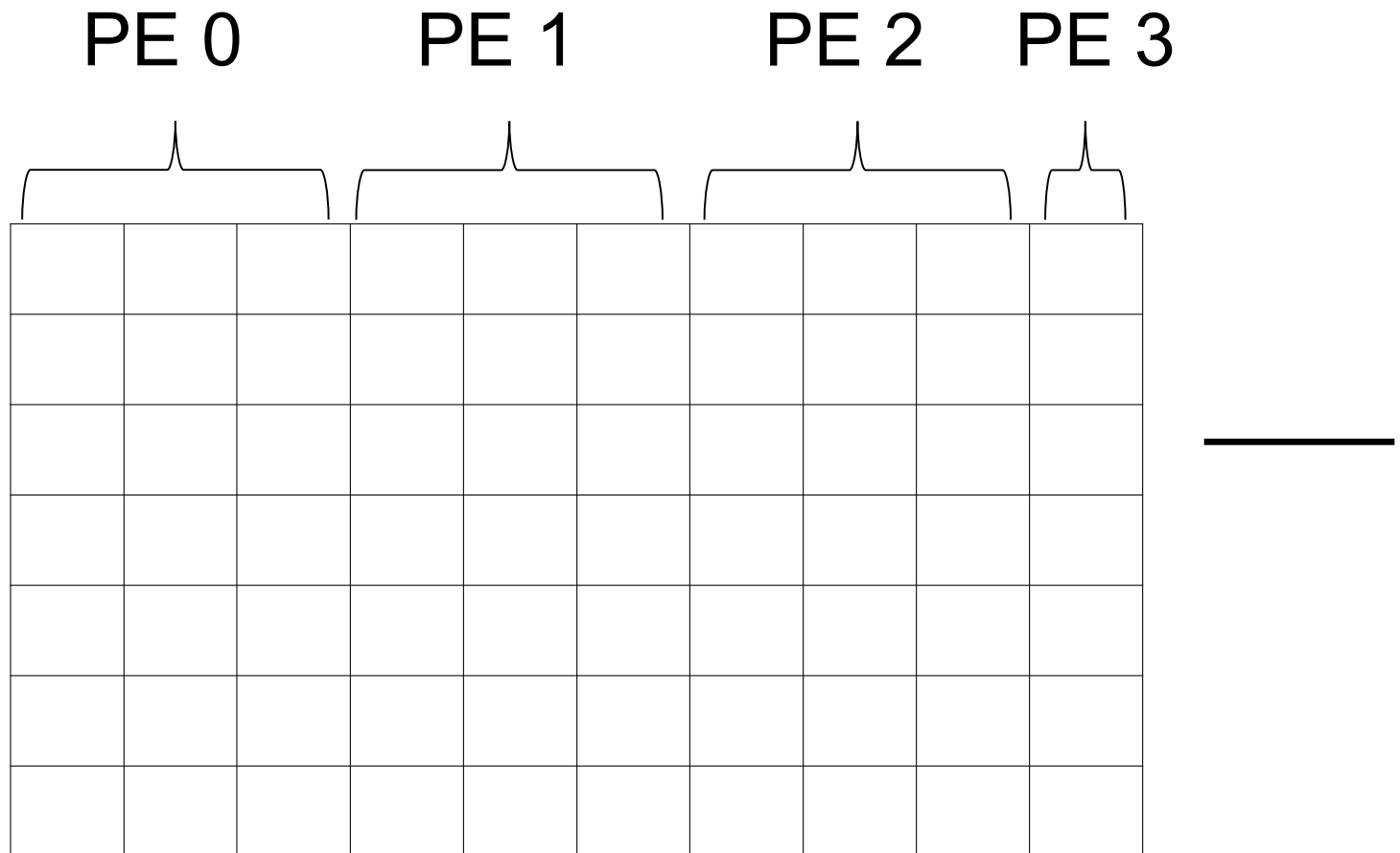
Scattering Columns of a Matrix

- Since C stores multi-dimensional arrays in row-major order, scattering rows of a matrix is easy



Scattering Columns of a Matrix

- What if we want to scatter columns?



Scattering Columns of a Matrix

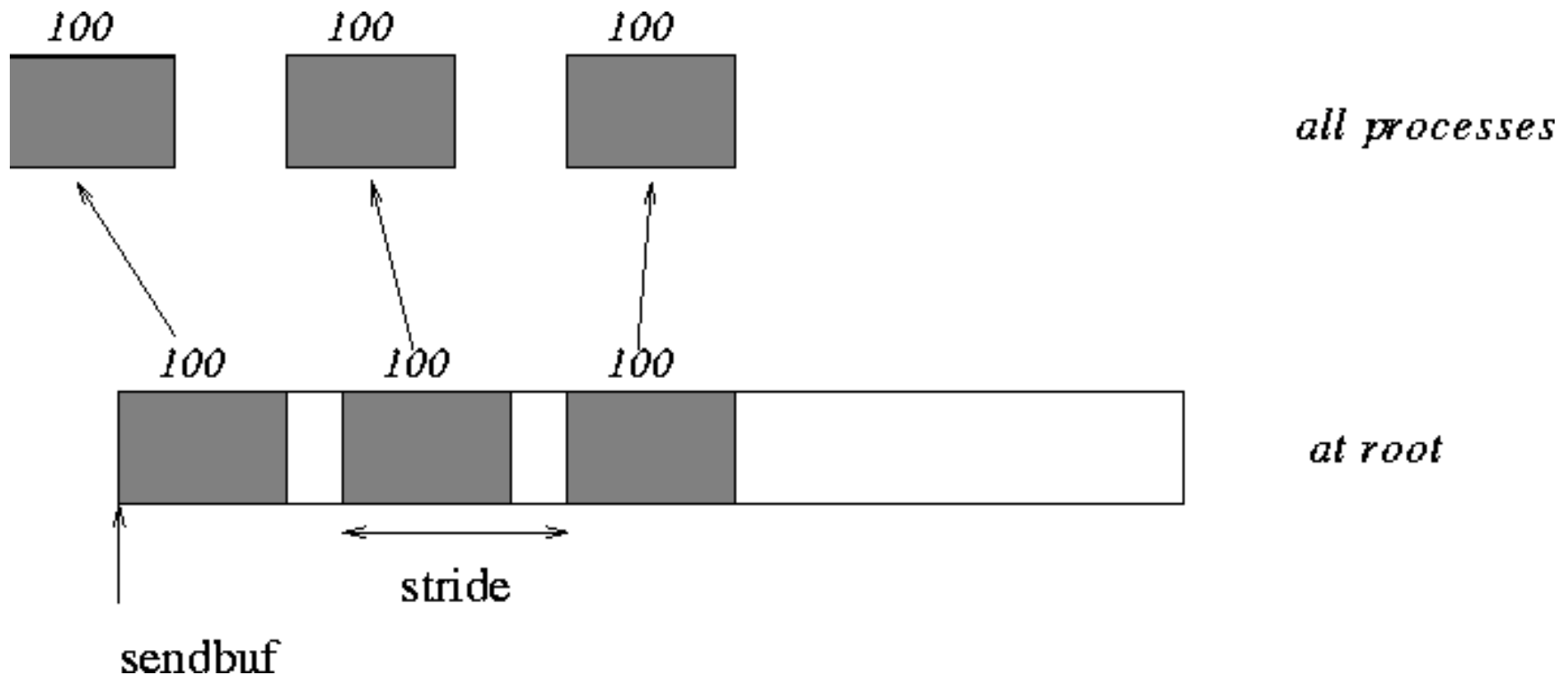
- Could use the `MPI_Datatype` and `MPI_Type_vector` features of MPI

OR

- An easier solution would be to transpose the matrix, then scatter the rows.

There is a version scatter called MPI_Scatterv, that can jump over parts of the array:

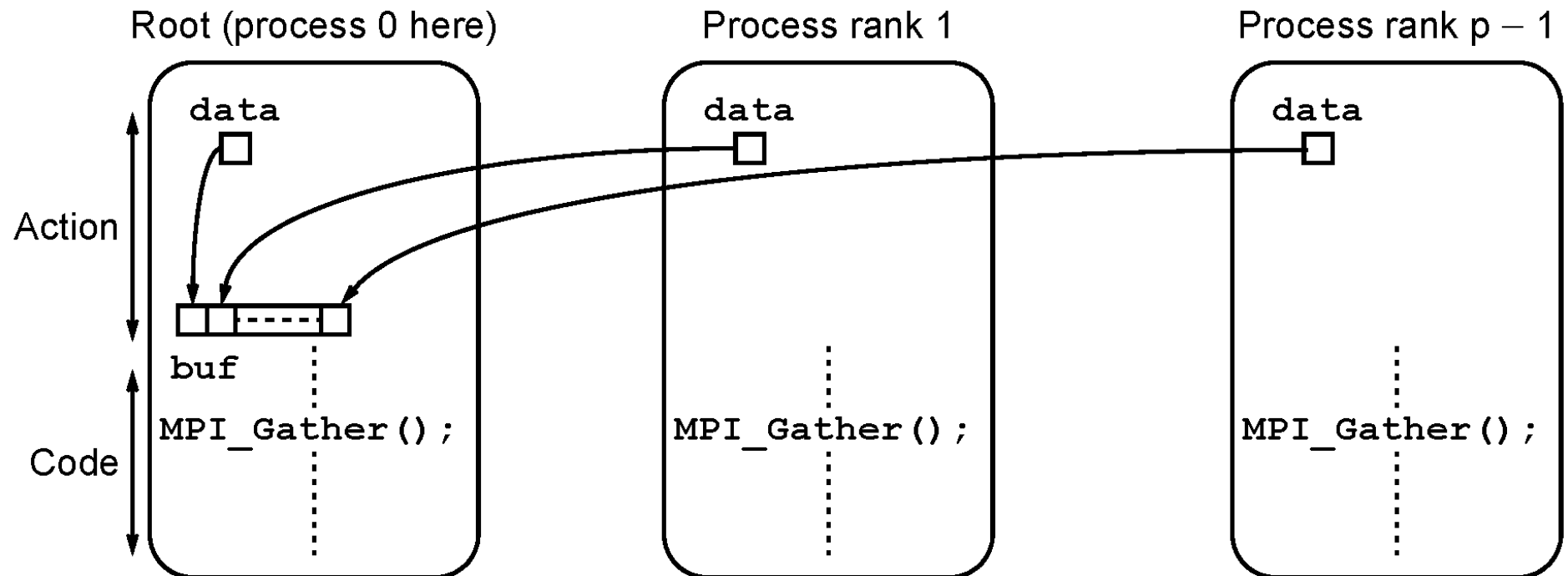
MPI_Scatterv Example



(source: <http://www.mpi-forum.org>)

Gather

Having one process collect individual values from set of processes.



Gather parameters

`MPI_Gather(*sbuf, scount, stype, *rbuf, rcount, rtype, root, comm)`

Address of send buffer
Datatype of items sent
Number of items to send to root process

Address of receive buffer
Datatype of items received
Number of items to receive
Rank of root process (destination)
Communicator

All processes in the Communicator must call the `MPI_Gather` with the same parameters

Gather Example

To gather items from group of processes into process 0, using dynamically allocated memory in root process:

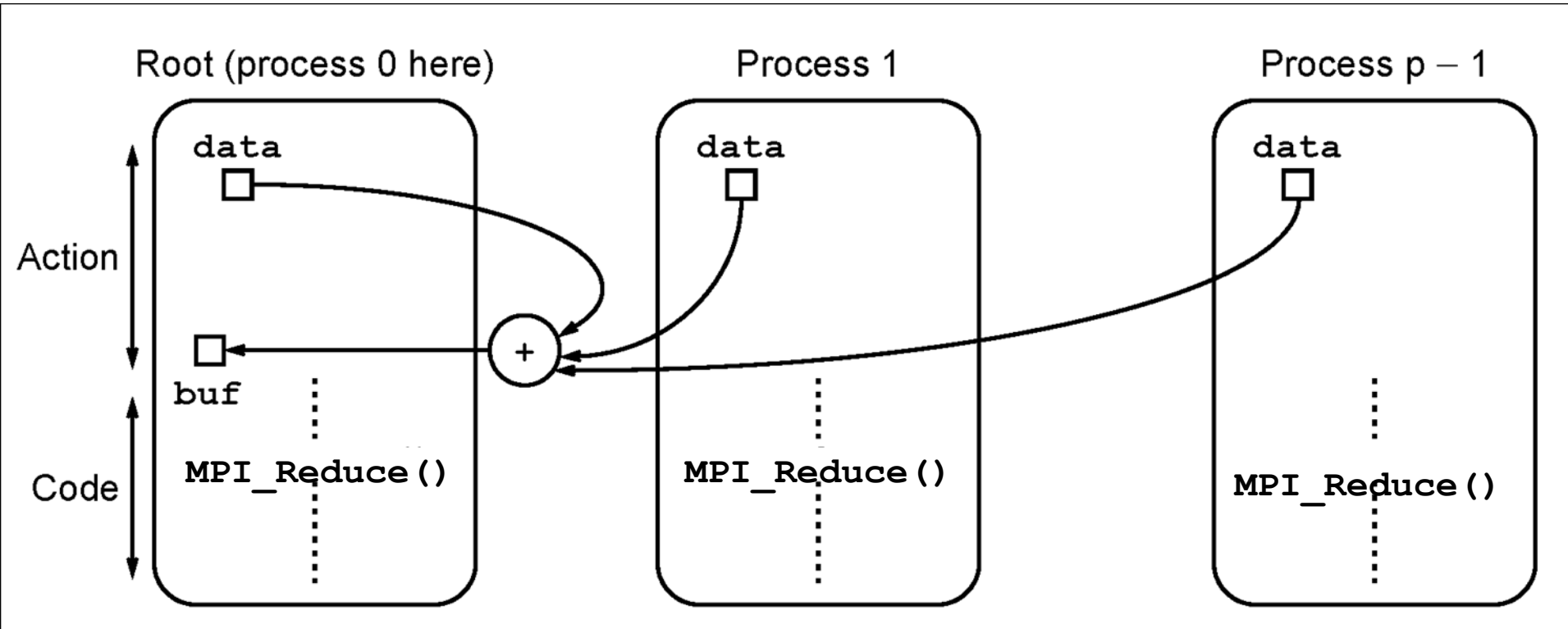
```
int data[10]; /*data to be gathered from processes*/
MPI_Comm_rank(MPI_COMM_WORLD, &myrank); /* find rank */
MPI_Comm_size(MPI_COMM_WORLD, &grp_size); /*find group size*/
if (myrank == 0) {
    buf = (int *)malloc(grp_size*10*sizeof (int)); /*alloc.
mem*/
}
MPI_Gather(data,10,MPI_INT,buf,grp_size*10,MPI_INT,0,MPI_COMM_W
ORLD) ;
...
```

MPI_Gather() gathers from all processes, including root.

Reduce

Gather operation combined with specified arithmetic/logical operation.

Example: Values could be gathered and then added together by root:



Reduce parameters

`MPI_Reduce(*sendbuf, *recvbuf, count, datatype, op, root, comm)`

Address of send buffer

Address of receive buffer

Number of items to send

Datatype of each item

Operation

Rank of root process (destination)

Communicator

All processes in the Communicator must call the `MPI_Reduce` with the same parameters

Reduce - operations

MPI_Reduce (*sendbuf, *recvbuf, count, datatype, op, root, comm)

Parameters:

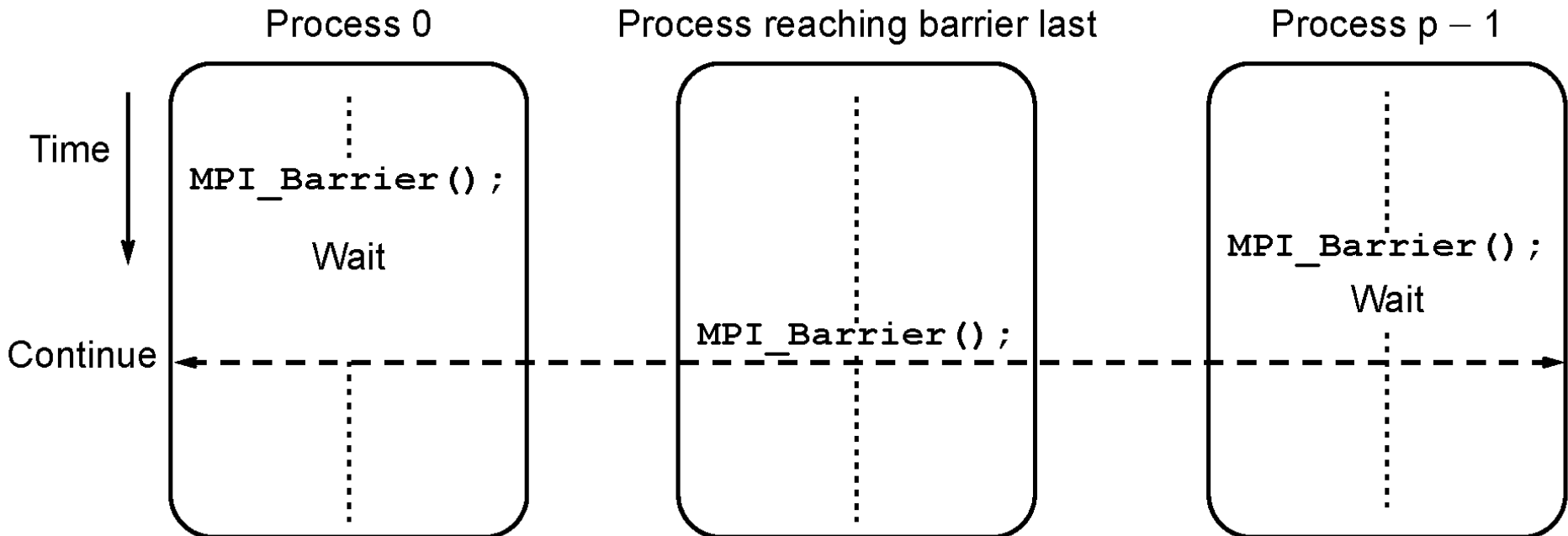
*sendbuf	send buffer address
*recvbuf	receive buffer address
count	number of send buffer elements
datatype	data type of send elements
op	reduce operation.
	Several operations, including
	MPI_MAX Maximum
	MPI_MIN Minimum
	MPI_SUM Sum
	MPI_PROD Product
root	root process rank for result
comm	communicator

Barrier

Block process until all processes have called it.
Synchronous operation.

`MPI_Barrier(comm)`

Communicator



MPI_Barrier use with time stamps

A common example of using a barrier is to synchronize the processors before taking a time stamp.

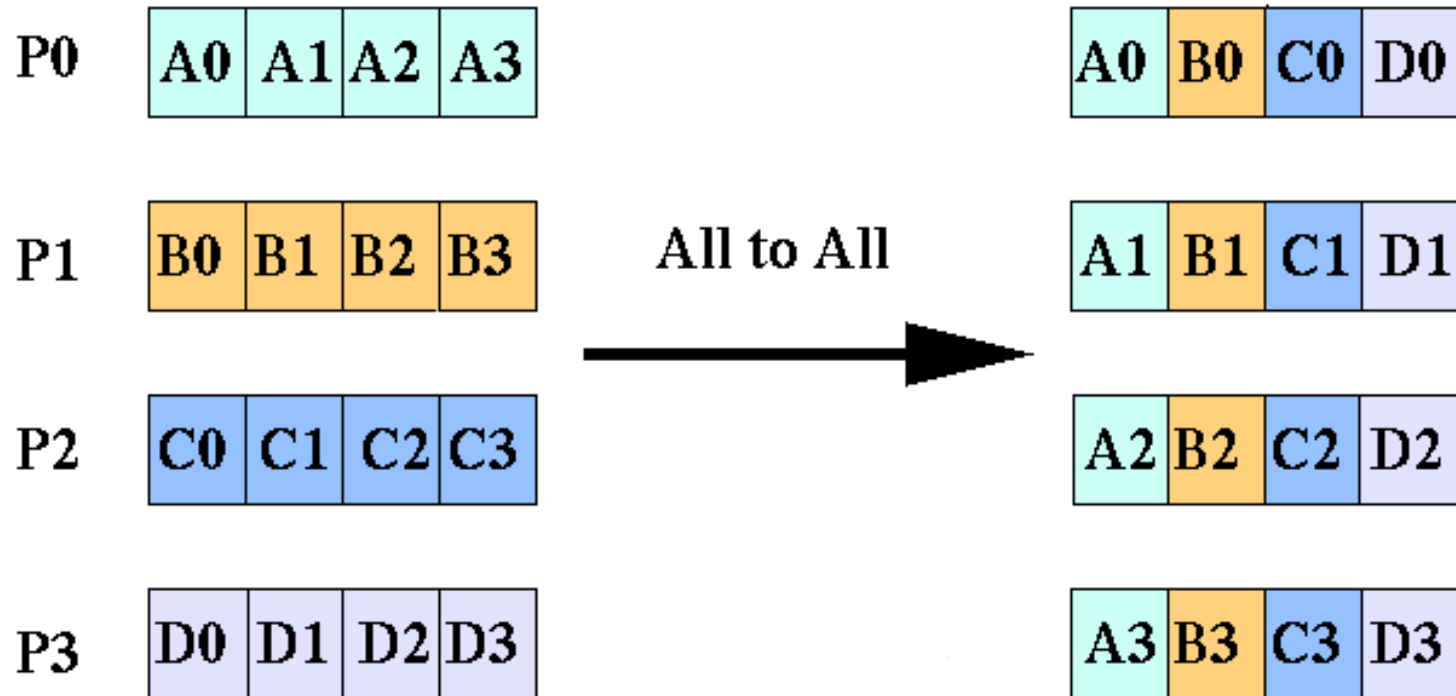
```
MPI_Barrier(MPI_COMM_WORLD);  
start_time = MPI_Wtime();  
... \\ Do work  
MPI_Barrier(MPI_COMM_WORLD);  
end_time = MPI_Wtime();
```

2nd barrier not always needed if there is a gather

Once the root has the correct data, who cares what the other processes are doing. We have the answer.

MPI_AlltoAll

Combines multiple scatters:



This is essentially matrix transposition

MPI_AlltoAll parameters

```
int MPI_Alltoall (  
    void *sendbuf,  
    int sendcount,  
    MPI_Datatype sendtype,  
    void *recvbuf,  
    int recvcount,  
    MPI_Datatype recvtype,  
    MPI_Comm comm );
```

Synchronous Message Passing

Routines that return when message transfer completed.

Synchronous send routine

- Waits until complete message can be accepted by the receiving process before sending the message.
In MPI, `MPI_SSend()` routine.

Synchronous receive routine

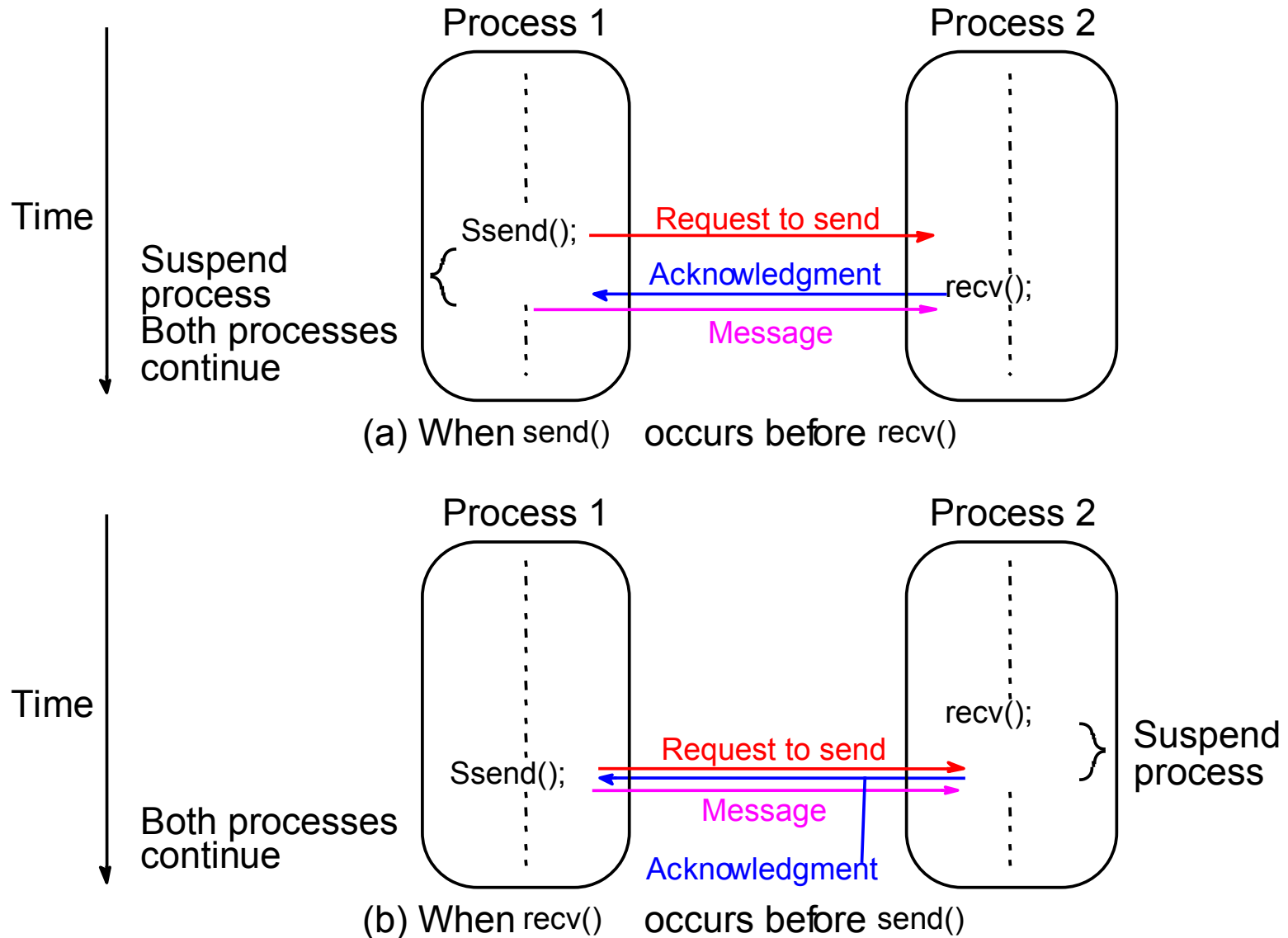
- Waits until the message it is expecting arrives.
In MPI, actually the regular `MPI_recv()` routine.

Synchronous Message Passing

Synchronous message-passing routines intrinsically perform two actions:

- They transfer data and
- They synchronize processes.

Synchronous Ssend() and recv() using 3-way protocol



Parameters of synchronous send (same as blocking send)

MPI_Ssend(buf, count, datatype, dest, tag, comm)

Address of
send buffer

Number of items
to send

Datatype of
each item

Rank of destination
process

Message tag

Communicator

Asynchronous Message Passing

- Routines that do not wait for actions to complete before returning. Usually require local storage for messages.
- More than one version depending upon the actual semantics for returning.
- In general, they do not synchronize processes but allow processes to move forward sooner.
- Must be used with care.

MPI Definitions of Blocking and Non-Blocking

- **Blocking** - return after their local actions complete, though the message transfer may not have been completed. Sometimes called **locally blocking**.
- **Non-blocking** - return immediately (*asynchronous*)

Non-blocking assumes that data storage used for transfer not modified by subsequent statements prior to being used for transfer, **and it is left to the programmer to ensure this**.

Blocking/non-blocking terms may have different interpretations in other systems.

MPI blocking routines

Block until local actions complete

- **Blocking send** - `MPI_send()` - blocks only until message is on its way. User can modify buffer after it returns.
- **Blocking receive** - `MPI_recv()` - blocks until message arrives

MPI Nonblocking Routines

- **Non-blocking send** - `MPI_Isend()` - will return “immediately” even before source location is safe to be altered.
- **Non-blocking receive** - `MPI_Irecv()` - will return even if no message to accept.

Nonblocking Routine Formats

`MPI_Isend(buf, count, datatype, dest, tag, comm, request)`

`MPI_Irecv(buf, count, datatype, source, tag, comm, request)`

Completion detected by `MPI_Wait()` and `MPI_Test()`.

`MPI_Wait()` waits until operation completed and returns then.

`MPI_Test()` returns with flag set indicating whether operation completed at that time.

Need to know whether particular operation completed.

Determined by accessing `request` parameter.

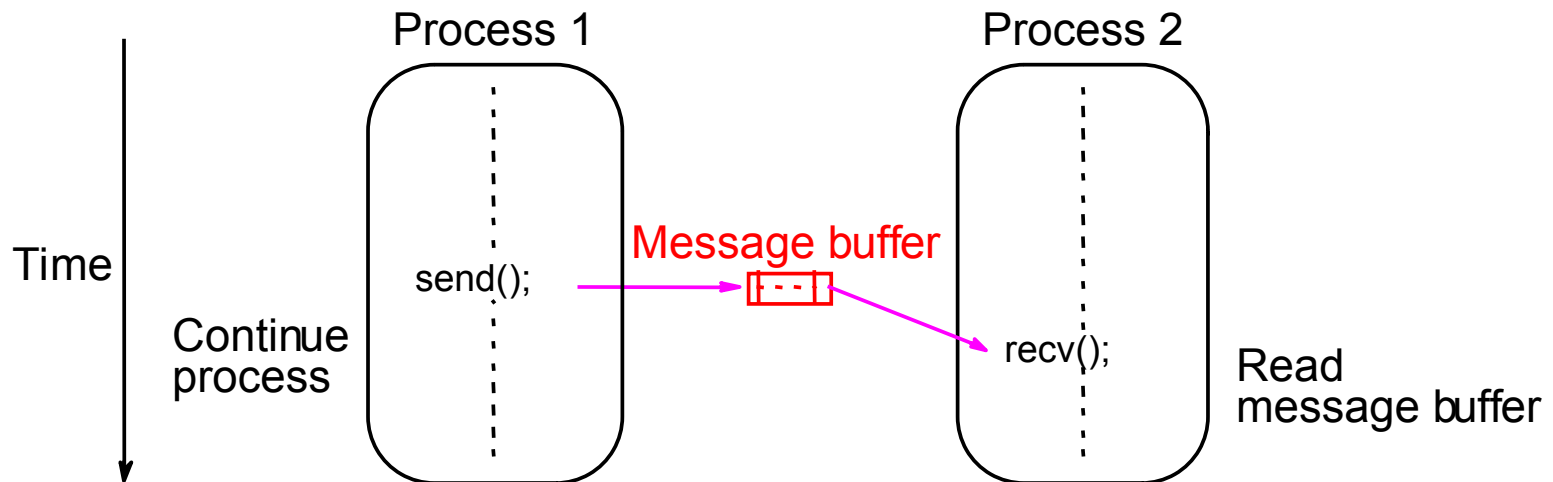
Example

To send an integer x from process 0 to process 1 and allow process 0 to continue:

```
MPI_Comm_rank(MPI_COMM_WORLD, &myrank); /* find rank */
if (myrank == 0) {
    int x;
    MPI_Isend(&x,1,MPI_INT, 1, msgtag, MPI_COMM_WORLD, req1);
    compute();
    MPI_Wait(req1, status);
} else if (myrank == 1) {
    int x;
    MPI_Recv(&x,1,MPI_INT,0,msgtag, MPI_COMM_WORLD, status);
}
```

How message-passing routines return before message transfer completed

Message buffer needed between source and destination to hold message:



Asynchronous (blocking) routines *changing to* synchronous routines

- Message buffers only of finite length
- A point could be reached when send routine held up because all available buffer space exhausted.
- Then, send routine will wait until storage becomes re-available - i.e. routine will behave as a synchronous routine.

That's it!