Collective Operations in MPI

CPSC 424/524
Lecture #9
October 24, 2018

MPI Communicators

- Communicators define communication domains
  - Virtual groups of processes (sometimes with topology)
  - Communicators facilitate independent streams of messages
  - Processes numbered consecutively in each communicator
- MPI has 2 types of communicators:
  - *[Intra]communicators*: msgs within groups of processes
  - *Intercommunicators*: msgs between disjoint groups of processes
- Creating Communicators (See "Using MPI" for more info)
  - MPI_Init initializes MPI_COMM_WORLD
  - MPI_Comm_create, MPI_Comm_split …
  - Topology-specific functions: MPI_Cart_create, MPI_Cart_sub

Collective Message-Passing Operations

- 1-to-many; many-to-1; many-to-many
  - Very convenient
  - May be more efficient in some cases (implementation dependent)
  - Not absolutely essential to use, but often convey programmer intent more accurately
  - Provide for at least “weak” synchronization

- Collectives include operations for:
  - Synchronization
  - Communication (broadcast, gather, scatter)
  - Reduction (cooperative computation)

Collective Operations: General Features

- Performed by all processes in one communicator
- Equivalent to multiple point-to-point calls, possibly with computation
- Locally blocking
- Synchronization
  - At least weakly synchronous
    - All processes must execute, but not necessarily at same time
  - *May* be strongly synchronous (implementation dependent)
  - Use `MPI_Barrier` for strong synchronization
    - No process can leave before all processes have entered
- Some collectives use a *root* process to originate/receive all data
- Data "segments" must exactly match in basic versions
  - Many variations for more generality
- No message tags are used (or needed)

Collective Operations

Principal collective operations:

- **`MPI_Barrier()`** - Synchronizes processes in a communicator by blocking each one until all call **`MPI_Barrier`**

- **`MPI_Bcast()`** - Broadcast from one root process to all processes
- **`MPI_Scatter()`** - Root decomposes buffer & sends segments to procs
- **`MPI_Gather()`** - Root assembles data segments from group of procs
- **`MPI_Allgather()`** - Like Gather, but all procs receive assembled data
- **`MPI_Alltoall()`** - Like simultaneous scatters from all procs to all

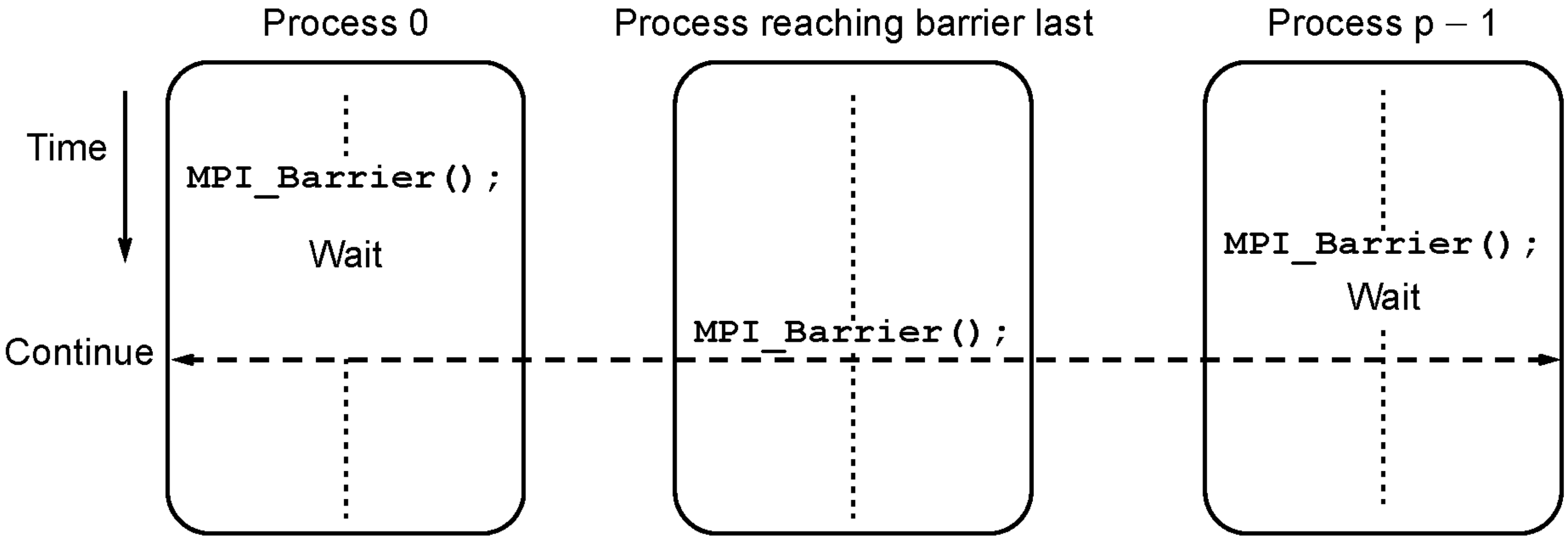
- **`MPI_Reduce()`** - Root receives values combining values frm all procs
- **`MPI_Allreduce()`** - Like Reduce, except all procs receive the results
- **`MPI_Reduce_scatter()`** - Combine values and scatter results
- **`MPI_Scan()`** - Processes receive partial (prefix) reductions (inclusive)
- **`MPI_Exscan()`** - Processes receive partial (prefix) reductions (exclusive)

MPI Barrier Operation

When any process makes a barrier call, it is blocked
until all processes in the communicator have made a barrier call

`MPI_Barrier(comm);`

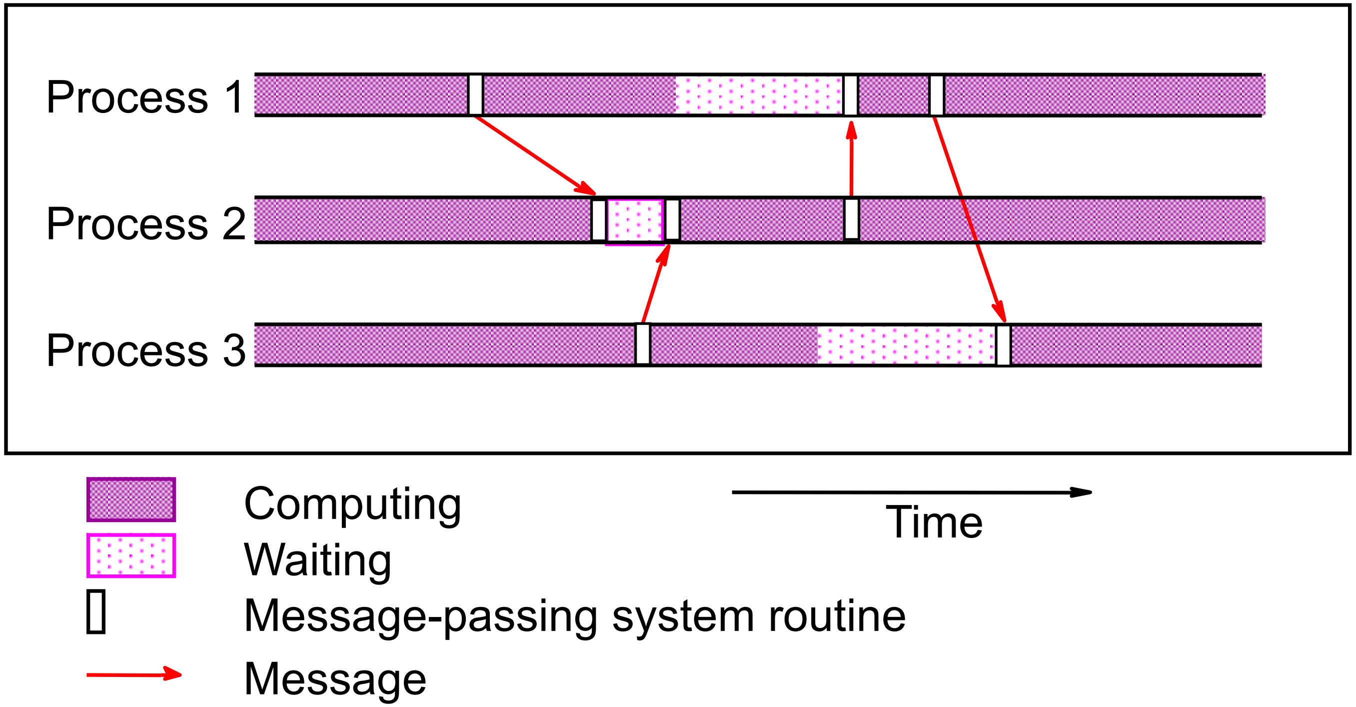
Uses:
- Program/algorithm correctness
- Performance measurement
- Debugging



Understanding Performance

Many approaches, including timeline diagrams, program timing, and a number of profiling and visualization tools.

Timeline Diagram:



Timing a Program

To measure execution time between point L1 and point L2 in serial code, might have construction such as:

```
L1: time(&t1);              //start timer
      .
      .
L2: time(&t2);              // stop timer
            .
elapsed_Time = difftime(t2, t1); // time=t2-t1

printf("Elapsed time=%5.2f secs", elapsed_Time);
```

- ***Know what you're timing!***!
- May need an extensive table of timings for different parts of a program
- Automatic profiling tools may be simpler and more effective
  - MPE (Multi-Processing Environment) is part of MPI and generates data that can be displayed graphically.

MPI Timing Routines

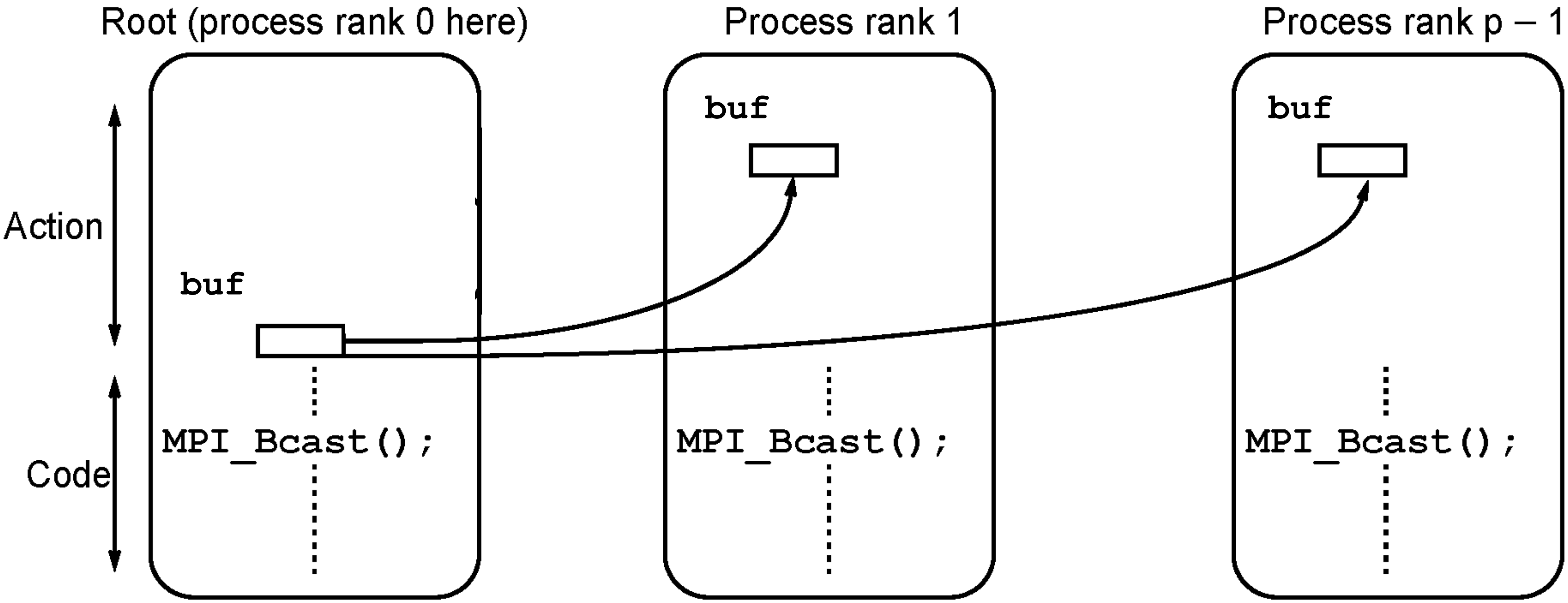
MPI provides the routine `MPI_Wtime()` that returns time (in seconds from an arbitrary reference point) as a double:

```
double start_time, end_time, exe_time;

       . . .

start_time = MPI_Wtime();

       . . .

end_time = MPI_Wtime();
exe_time = end_time - start_time;
```

MPI Broadcast Operation

Root sends same data buffer to all other processes in communicator

(MPI has no multicast---sending to a subset of processes)



MPI_Bcast Parameters

```
MPI_Bcast(*buf, count, datatype, root, comm)
```

Address of send buffer

Number of items to send

Datatype of each item

Rank root process (source of broadcast)

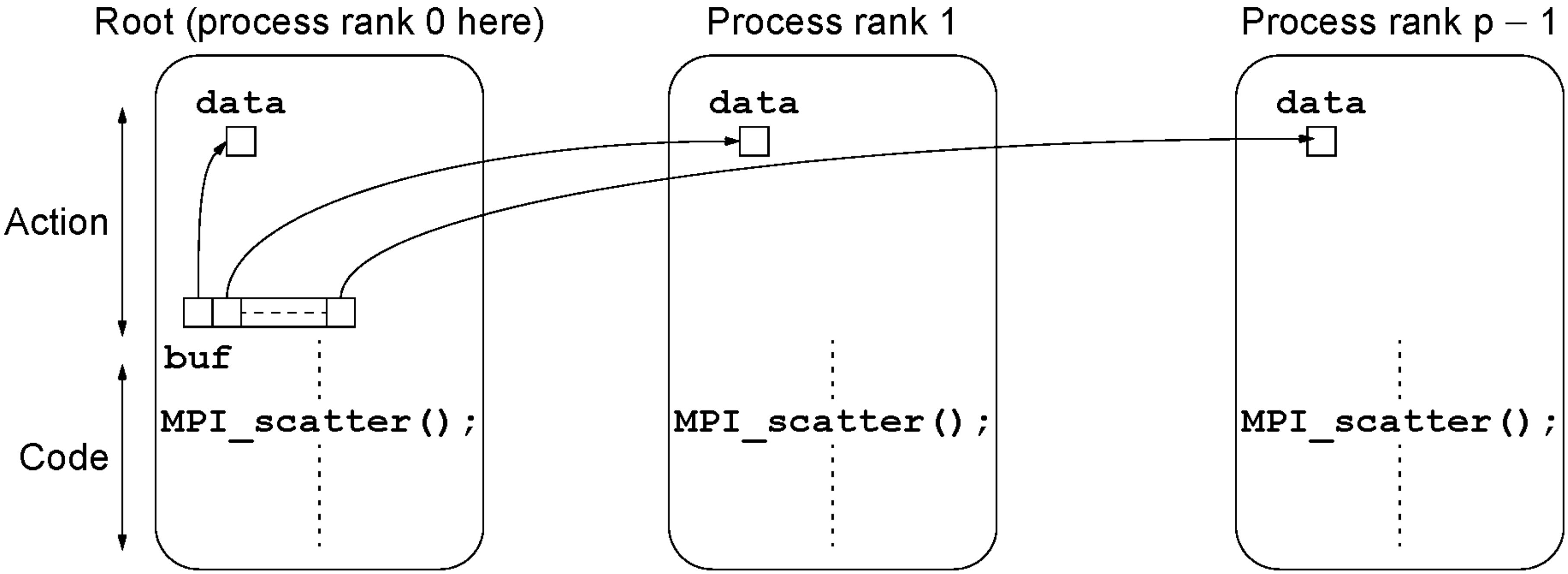
Communicator

Note: **`buf`** is a send buffer on root, but a recv buffer elsewhere

MPI Scatter Operation

<u>Root</u> sends (different) <u>equal-size segments</u> of data buffer to each of the processes (*including itself*)

The i^{th} segment goes to the i^{th} process (ordered by rank)

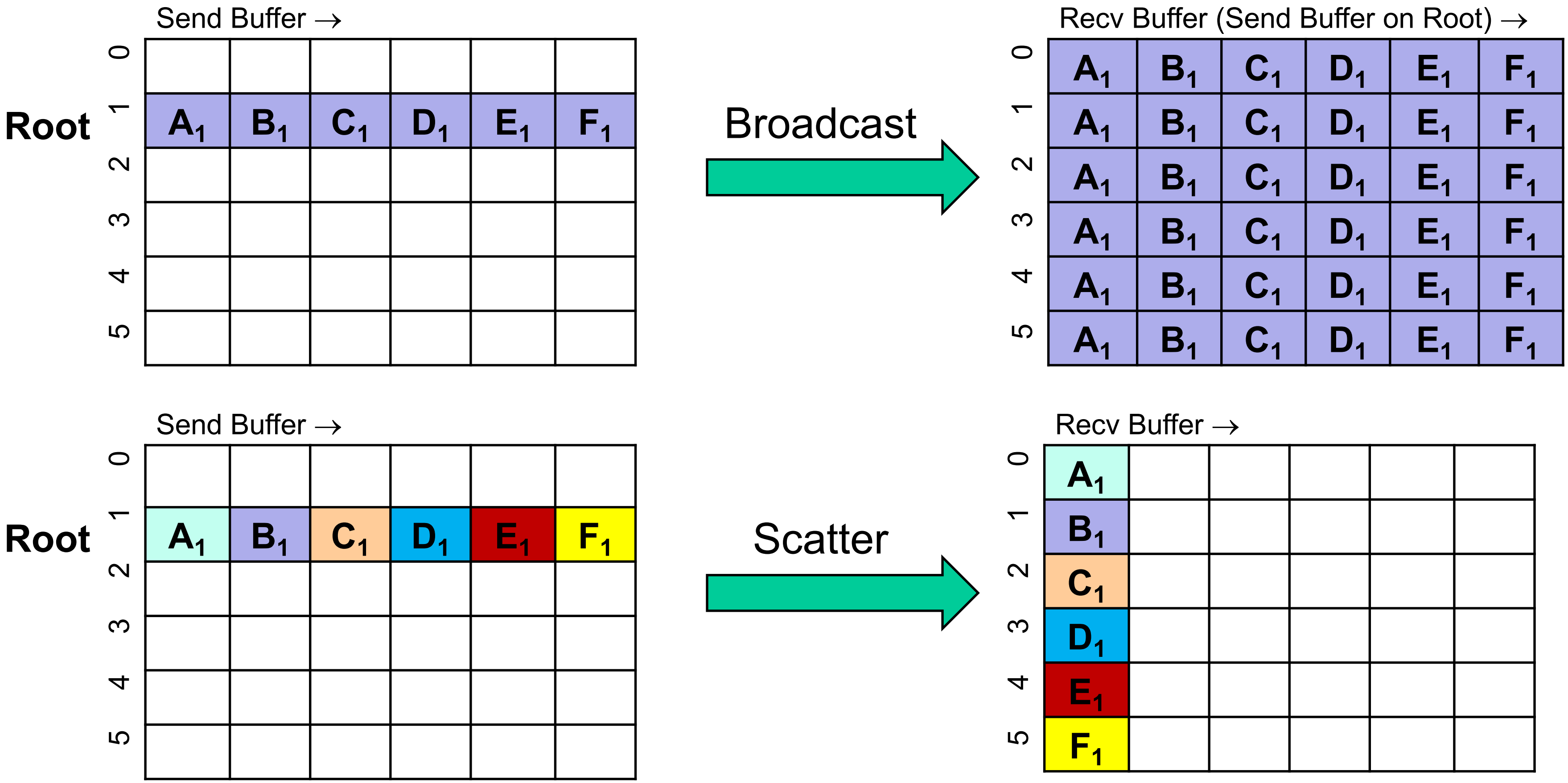


Broadcast vs. Scatter

Think of send buffers as having multiple equal-size segments: $\mathbf{A_p}$, $\mathbf{B_p}$, $\mathbf{C_p}$, $\mathbf{D_p}$, $\mathbf{E_p}$, $\mathbf{F_p}$

Buffer segments ordered left-to-right as **A**, **B**, **C**, …

Processes ordered top-to-bottom as **0**, **1**, **2**, …



Send Buffer →

Root

0						
1	A_1	B_1	C_1	D_1	E_1	F_1
2						
3						
4						
5						

Broadcast

Recv Buffer (Send Buffer on Root) →

0	A_1	B_1	C_1	D_1	E_1	F_1
1	A_1	B_1	C_1	D_1	E_1	F_1
2	A_1	B_1	C_1	D_1	E_1	F_1
3	A_1	B_1	C_1	D_1	E_1	F_1
4	A_1	B_1	C_1	D_1	E_1	F_1
5	A_1	B_1	C_1	D_1	E_1	F_1

Send Buffer →

Root

0						
1	A_1	B_1	C_1	D_1	E_1	F_1
2						
3						
4						
5						

Scatter

Recv Buffer →

0	A_1					
1	B_1					
2	C_1					
3	D_1					
4	E_1					
5	F_1					

MPI_Scatter Parameters

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

Address of send buffer

Number of items to send to each process

Datatype of items sent

Address of receive buffer

Number of items to receive

Datatype of items received

Rank of root process (source)

Communicator

Notes:

1. **`sbuf`**-related parameters only matter on the root process
2. Can use **`MPI_IN_PLACE`** for **`rbuf`** on root to avoid local data movement (when data is already in proper place in **`rbuf`**).

MPI Scatter Example

In the following code, the segment size for the data received by each process is `100` elements, so the total size of the send buffer is `100*<number of processes>` elements.

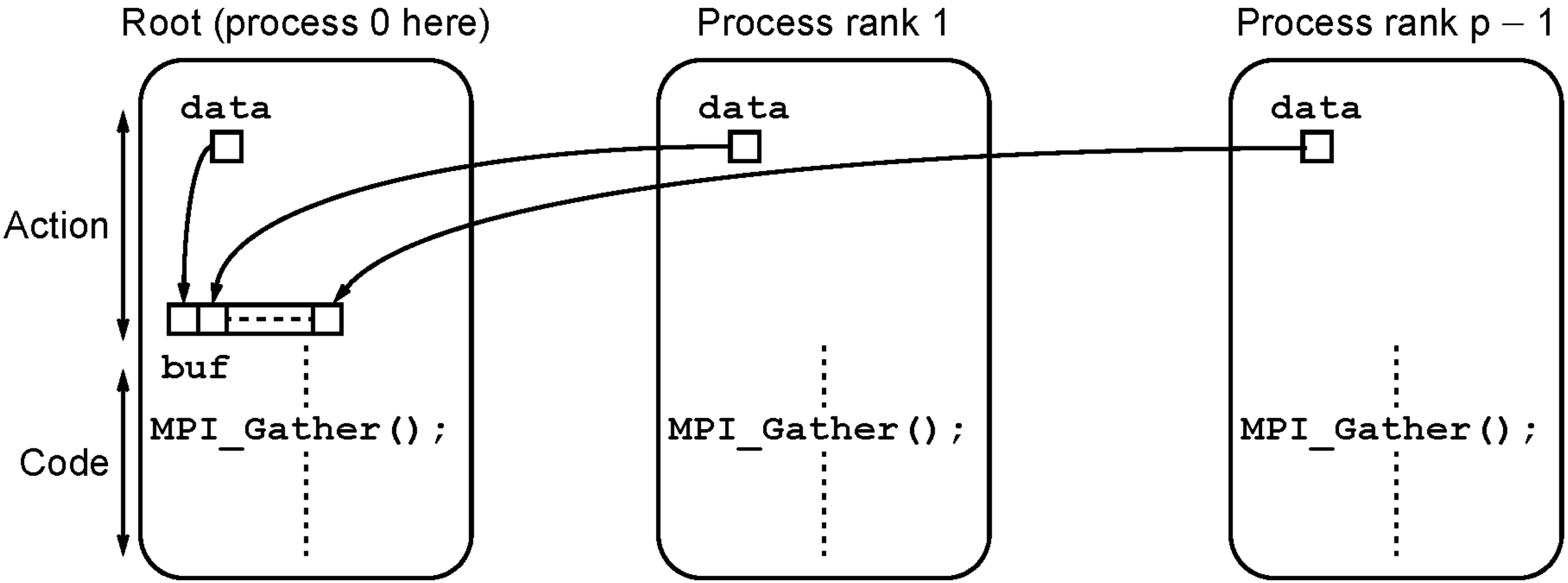
```
main (int argc, char *argv[]) {
  int size, myrank, *sendbuf=0, recvbuf[100], root=0;
  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  MPI_Comm_size(MPI_COMM_WORLD, &size);
  if (myrank==root) {
   sendbuf = (int *)malloc(size*100*sizeof(int));//root
      . . . // root initializes the sendbuf here
 }
  MPI_Scatter(sendbuf, 100, MPI_INT,
              recvbuf, 100, MPI_INT,
              root, MPI_COMM_WORLD);
      . . .
  MPI_Finalize();
}
```

MPI Gather Operation

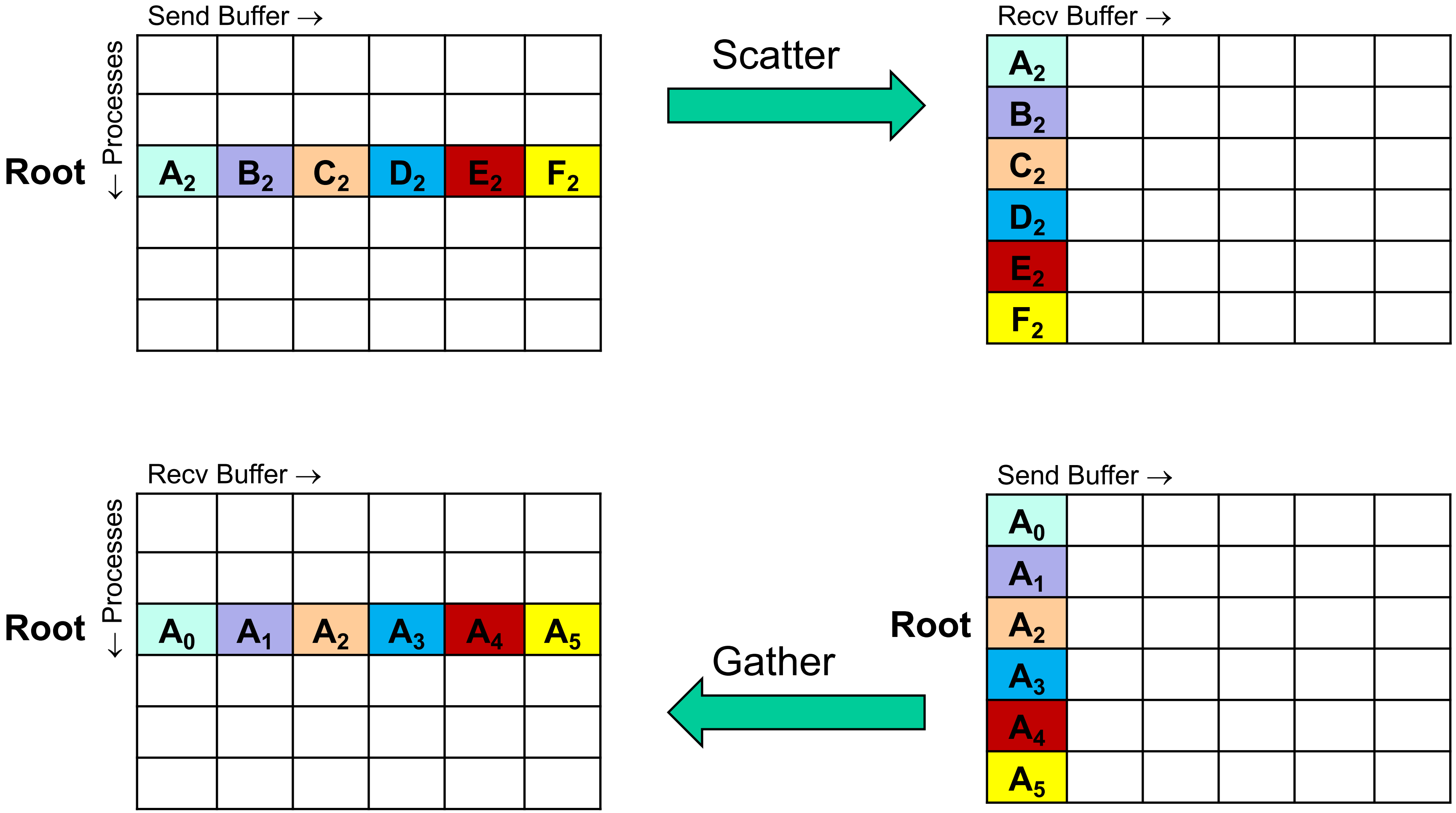
Root receives (different) equal-size segments of data buffer from each of the processes (*including itself*)

The i^{th} segment comes from i^{th} process (ordered by rank)

“Inverse” of scatter



Scatter vs. Gather



Send Buffer →

← Processes

Root

A_2	B_2	C_2	D_2	E_2	F_2

Scatter →

Recv Buffer →

A_2					
B_2					
C_2					
D_2					
E_2					
F_2					

Recv Buffer →

← Processes

Root

A_0	A_1	A_2	A_3	A_4	A_5

← Gather

Send Buffer →

Root

A_0					
A_1					
A_2					
A_3					
A_4					
A_5					

MPI_Gather Parameters

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

Address of send buffer

Number of items to send *to root process*

Datatype of items sent

Address of receive buffer

Number of items to receive

Datatype of items received

Rank of root process (destination)

Communicator

Notes:

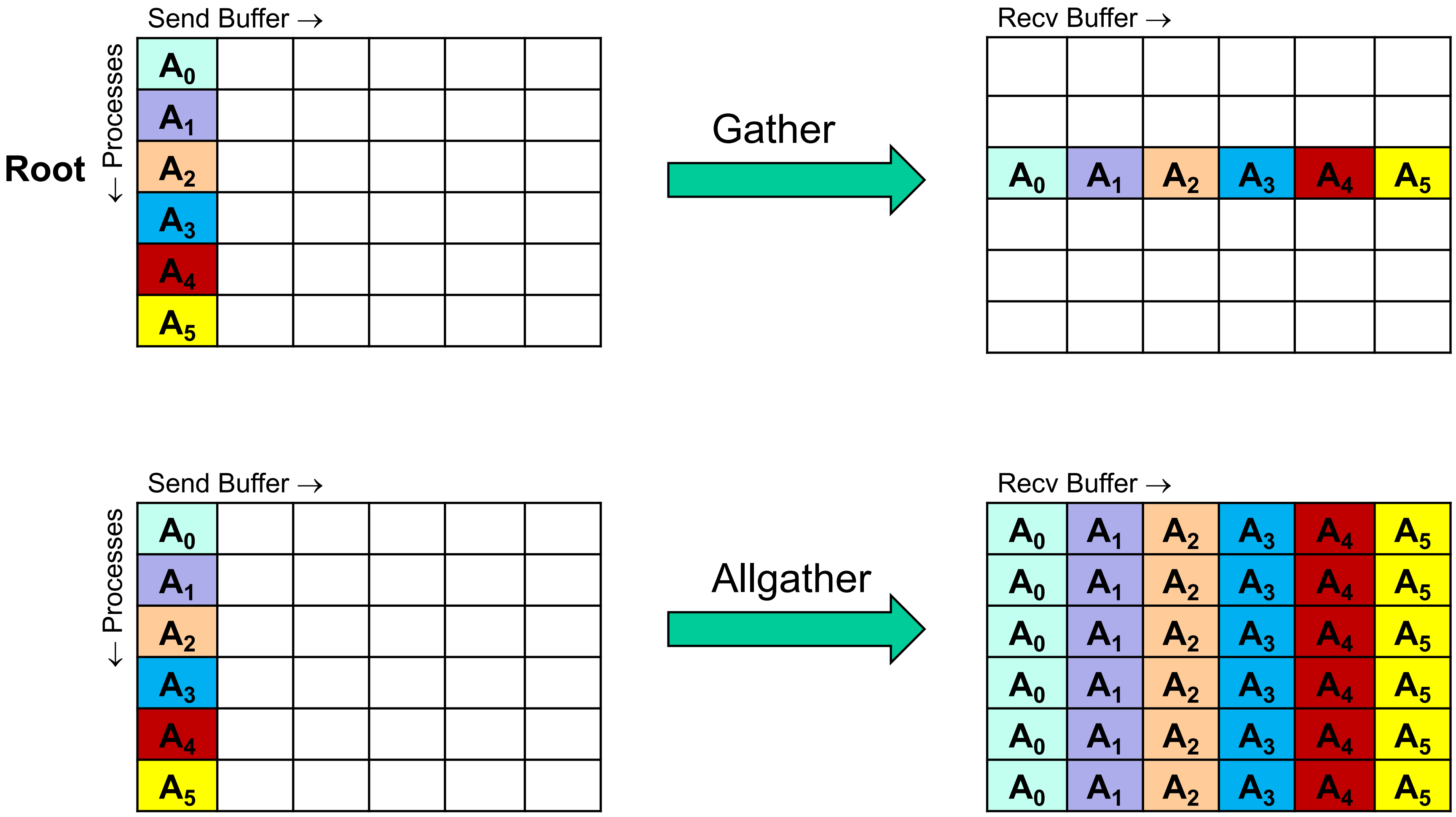
1. Receive-related parameters only matter on the root process
2. Can use **`MPI_IN_PLACE`** for **`sbuf`** on root if the local segment is already in the proper place in **`rbuf`**.

MPI Gather Example

In the following code, the segment size for the data sent by each process is `100` elements, so the recv buffer must be be able to hold at least `100*<number of processes>` elements.

```
main (int argc, char *argv[]) {
  int size, myrank, *recvbuf, sendbuf[100], root=0;
  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
  MPI_Comm_size(MPI_COMM_WORLD, &size);
  if (myrank==root)
    recvbuf = (int *)malloc(size*100*sizeof(int));//master only
      . . .
  MPI_Gather(sendbuf, 100, MPI_INT,
             recvbuf, 100, MPI_INT,
             root, MPI_COMM_WORLD);
      . . .
  MPI_Finalize();
}
```

Gather vs. Allgather



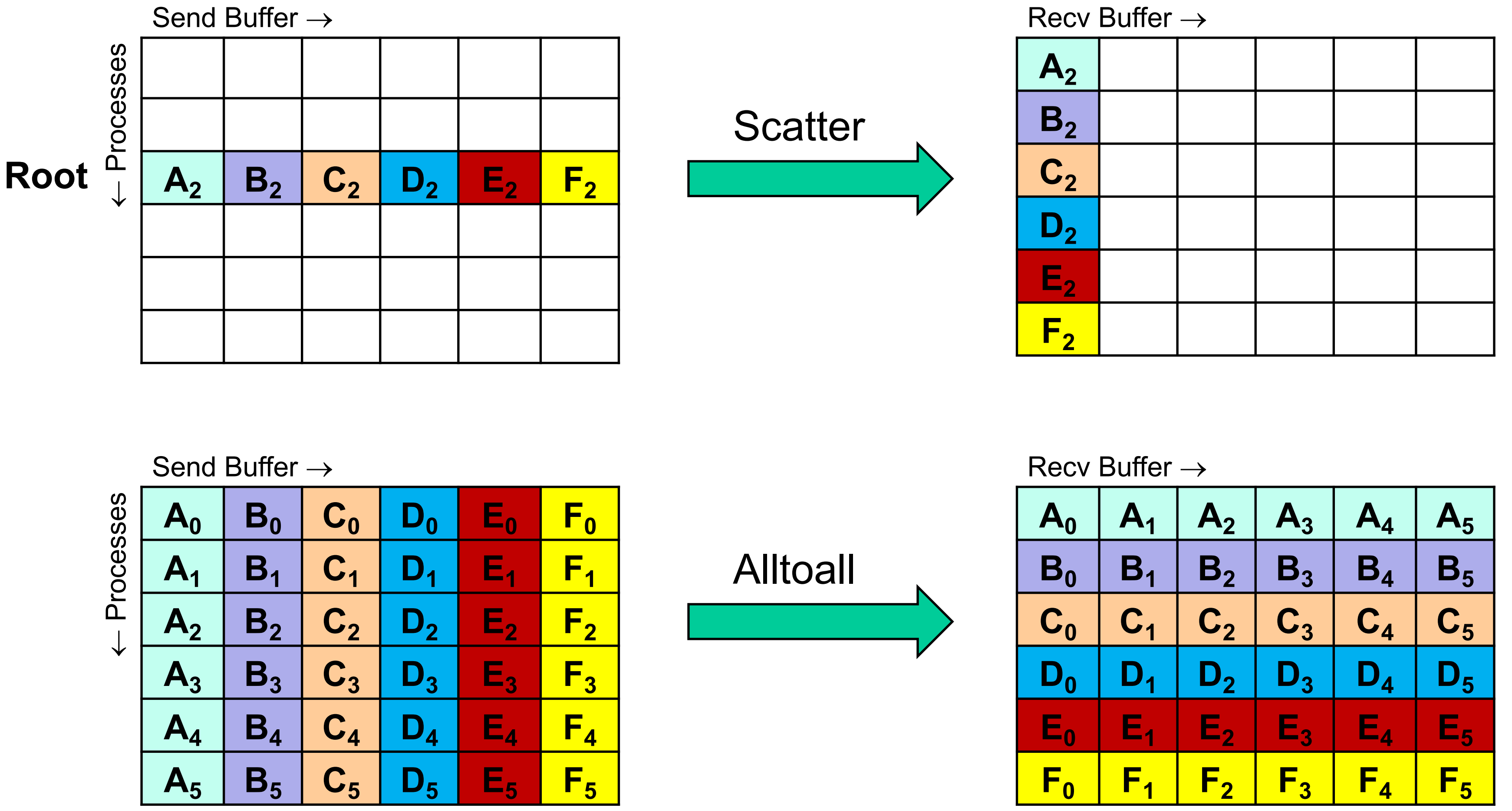
MPI Allgather Example

Allgather like gather, except all processes receive the data (no root)

In the following code, the segment size is `100` elements, so the size of the recv buffer is `100*<number of processes>` elements.

```
main (int argc, char *argv[]) {
  int size, myrank, *recvbuf, sendbuf[100];
  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
  MPI_Comm_size(MPI_COMM_WORLD, &size);
  recvbuf = (int *)malloc(size*100*sizeof(int));//all procs
      . . .
  MPI_Allgather(sendbuf, 100, MPI_INT,
                recvbuf, 100, MPI_INT,
                MPI_COMM_WORLD);
      . . .
  MPI_Finalize();
}
```

Scatter vs. Alltoall



Send Buffer →

← Processes

Root

A_2	B_2	C_2	D_2	E_2	F_2

Scatter

Recv Buffer →

A_2					
B_2					
C_2					
D_2					
E_2					
F_2					

Send Buffer →

← Processes

A_0	B_0	C_0	D_0	E_0	F_0
A_1	B_1	C_1	D_1	E_1	F_1
A_2	B_2	C_2	D_2	E_2	F_2
A_3	B_3	C_3	D_3	E_3	F_3
A_4	B_4	C_4	D_4	E_4	F_4
A_5	B_5	C_5	D_5	E_5	F_5

Alltoall

Recv Buffer →

A_0	A_1	A_2	A_3	A_4	A_5
B_0	B_1	B_2	B_3	B_4	B_5
C_0	C_1	C_2	C_3	C_4	C_5
D_0	D_1	D_2	D_3	D_4	D_5
E_0	E_1	E_2	E_3	E_4	E_5
F_0	F_1	F_2	F_3	F_4	F_5

Equivalent to multiple scatters (in rank order).

Effectively, this is like transposing a data matrix.

MPI Alltoall Example

Every process sends and receives data (no root)

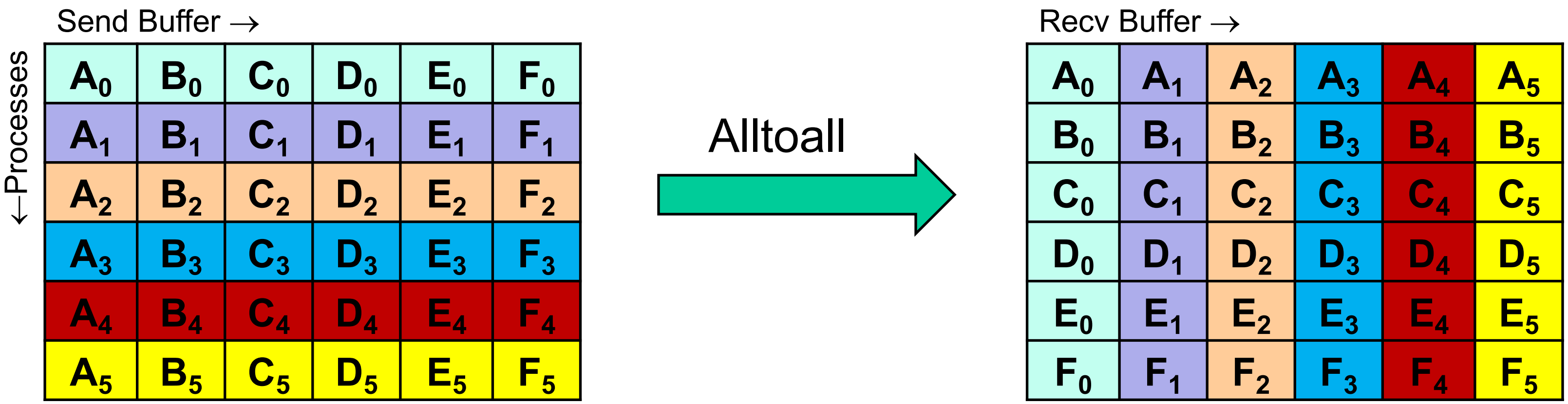
In the following code, the segment size is `100` elements, so the total size of each send or recv buffer is `100*<number of processes>` elements

```
main (int argc, char *argv[]) {
  int size, myrank, *recvbuf, *sendbuf;
  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
  MPI_Comm_size(MPI_COMM_WORLD, &size);
  sendbuf = (int *)malloc(size*100*sizeof(int));
  recvbuf = (int *)malloc(size*100*sizeof(int));
      . . .
  MPI_Alltoall(sendbuf, 100, MPI_INT,
               recvbuf, 100, MPI_INT,
               MPI_COMM_WORLD);
      . . .
  MPI_Finalize();
}
```

Generalized Gather/Scatter-like Operations

- There are generalized versions that permit segments to have different sizes and starting displacements in the send buffer:
  - `MPI_Scatterv`
  - `MPI_Gatherv`
  - `MPI_Allgatherv`
  - `MPI_Alltoallv`
  - `MPI_Alltoallw`
- For “v” operations, all data is of fixed type, but size of the data segments may vary, and the data segments may not be adjacent in the send buffer. Segment starts are specified as displacements (no. of elements) from the start of the send buffer.
- For “w” operations, types may also vary. (For that reason, data segment starting displacements must be measured in bytes, not elements.)

Alltoall and Alltoallv



Send Buffer →

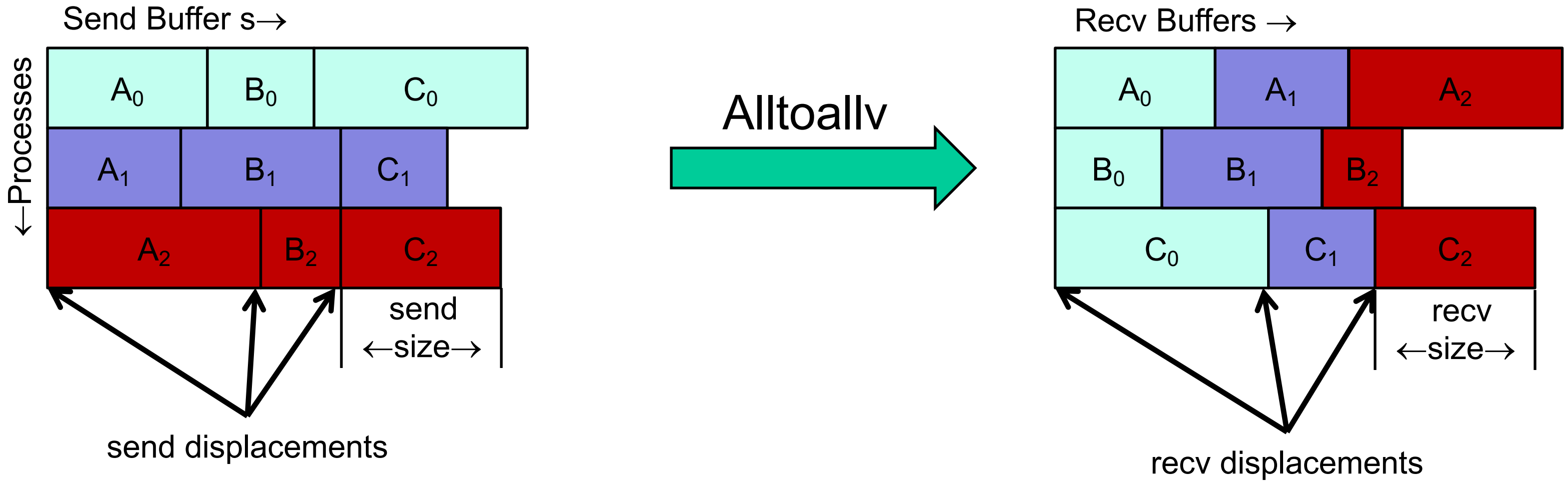
←Processes

A_0	B_0	C_0	D_0	E_0	F_0
A_1	B_1	C_1	D_1	E_1	F_1
A_2	B_2	C_2	D_2	E_2	F_2
A_3	B_3	C_3	D_3	E_3	F_3
A_4	B_4	C_4	D_4	E_4	F_4
A_5	B_5	C_5	D_5	E_5	F_5

Alltoall

Recv Buffer →

A_0	A_1	A_2	A_3	A_4	A_5
B_0	B_1	B_2	B_3	B_4	B_5
C_0	C_1	C_2	C_3	C_4	C_5
D_0	D_1	D_2	D_3	D_4	D_5
E_0	E_1	E_2	E_3	E_4	E_5
F_0	F_1	F_2	F_3	F_4	F_5



Send Buffer s→

←Processes

A_0	B_0	C_0
A_1	B_1	C_1
A_2	B_2	C_2

send ←size→

send displacements

Alltoallv

Recv Buffers →

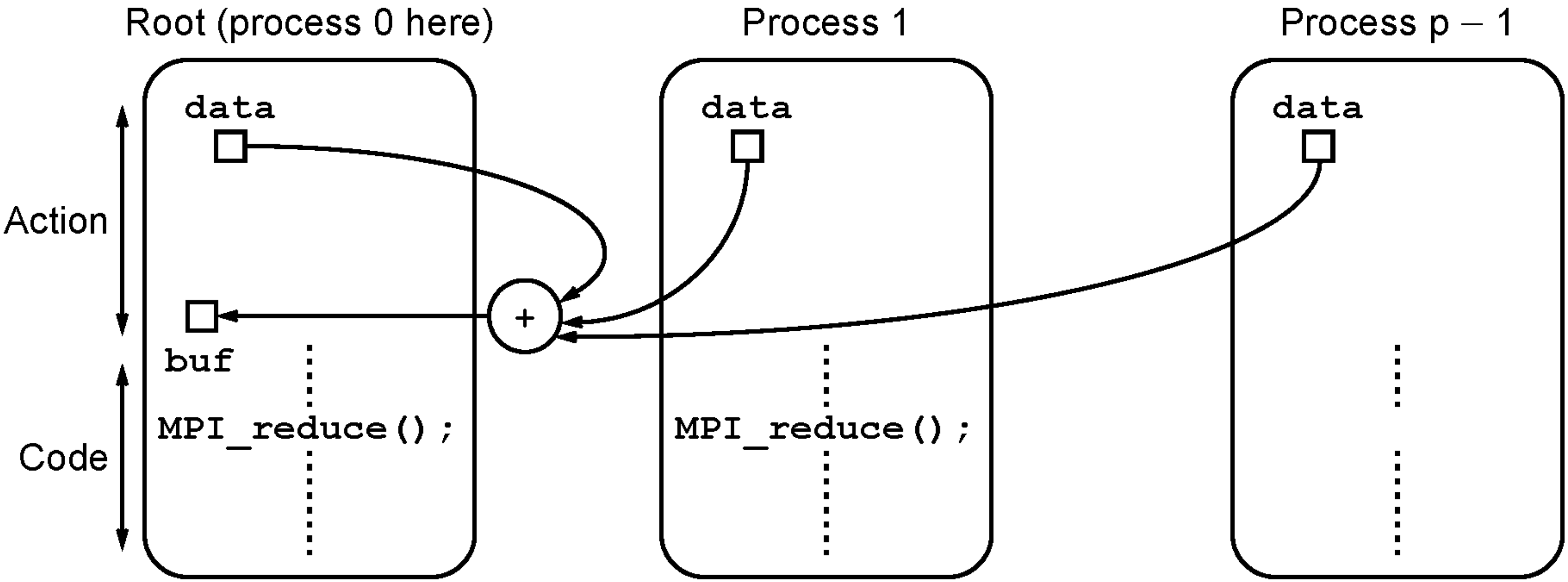
A_0	A_1	A_2
B_0	B_1	B_2
C_0	C_1	C_2

recv ←size→

recv displacements

Global Reduction Operations

- Combine gather-like operation with arithmetic/logical operation
- Common example: Adding up partial sums
- Very general: can define your own operation



MPI_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

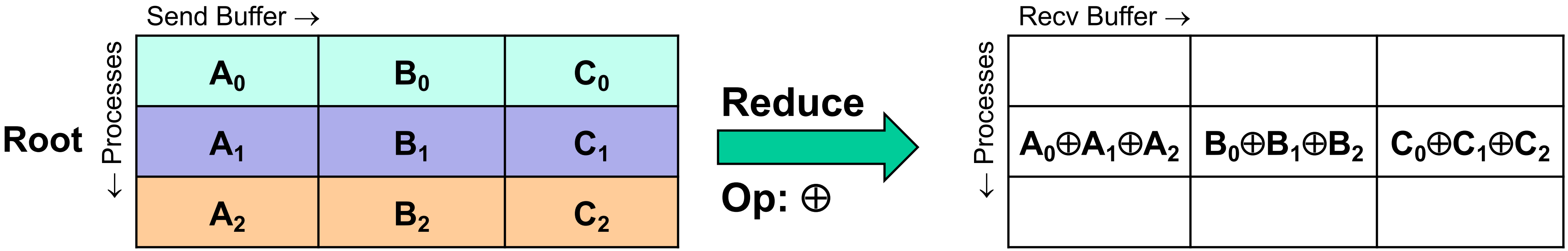
Operation

Rank of root process (destination)

Communicator

- Arguments similar to **`MPI_Gather`** except for **`op`** argument
- Predefined Reduction Operations:
  - Min/Max: **`MPI_MAX`**, **`MPI_MIN`**, **`MPI_MAXLOC`**, **`MPI_MINLOC`**
  - Arithmetic ops: **`MPI_SUM`**, **`MPI_PROD`**
  - Logical ops: **`MPI_LAND`**, **`MPI_LOR`**, **`MPI_LXOR`**
  - Bit-wise ops: **`MPI_BAND`**, **`MPI_BOR`**, **`MPI_BXOR`**
- Or you can define your own reduction operations

Reduce vs. Allreduce vs. Reduce_scatter



Send Buffer →

↓ Processes

	A_0	B_0	C_0
Root	A_1	B_1	C_1
	A_2	B_2	C_2

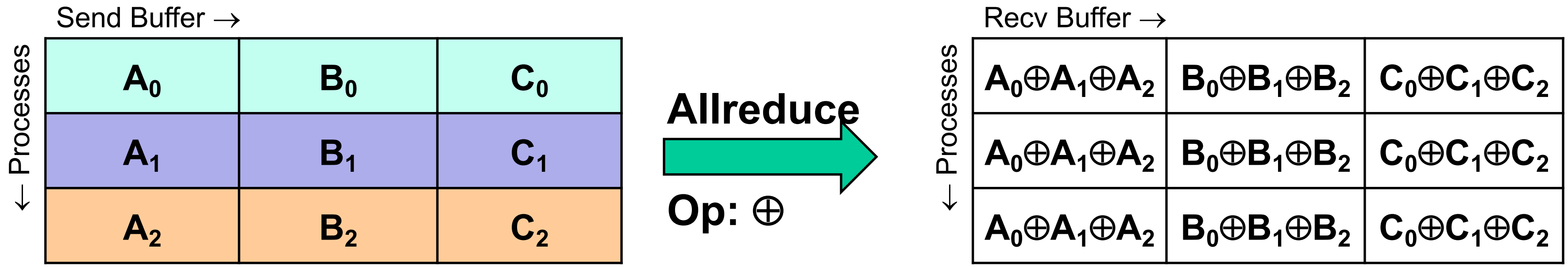
Reduce

Op: ⊕

Recv Buffer →

↓ Processes

$A_0 \oplus A_1 \oplus A_2$	$B_0 \oplus B_1 \oplus B_2$	$C_0 \oplus C_1 \oplus C_2$



Send Buffer →

↓ Processes

A_0	B_0	C_0
A_1	B_1	C_1
A_2	B_2	C_2

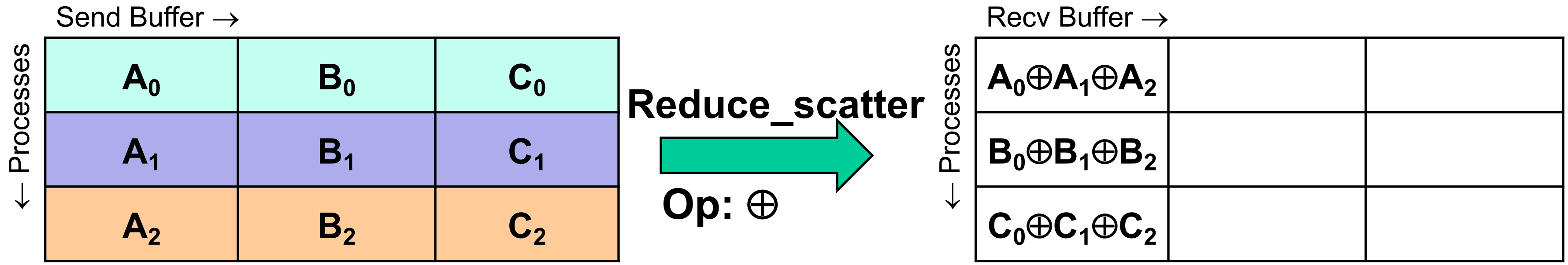
Allreduce

Op: ⊕

Recv Buffer →

↓ Processes

$A_0 \oplus A_1 \oplus A_2$	$B_0 \oplus B_1 \oplus B_2$	$C_0 \oplus C_1 \oplus C_2$
$A_0 \oplus A_1 \oplus A_2$	$B_0 \oplus B_1 \oplus B_2$	$C_0 \oplus C_1 \oplus C_2$
$A_0 \oplus A_1 \oplus A_2$	$B_0 \oplus B_1 \oplus B_2$	$C_0 \oplus C_1 \oplus C_2$



Send Buffer →

↓ Processes

A_0	B_0	C_0
A_1	B_1	C_1
A_2	B_2	C_2

Reduce_scatter

Op: ⊕

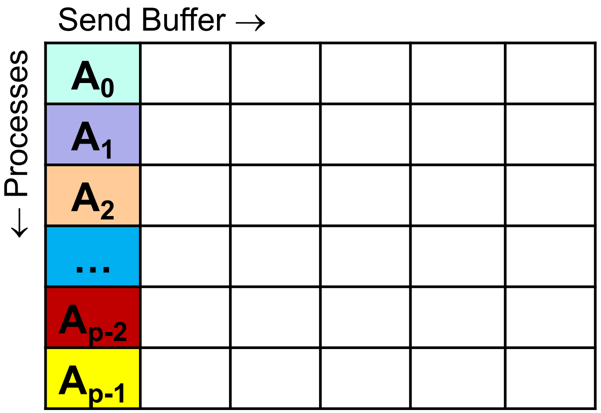
Recv Buffer →

↓ Processes

$A_0 \oplus A_1 \oplus A_2$		
$B_0 \oplus B_1 \oplus B_2$		
$C_0 \oplus C_1 \oplus C_2$		

Scans: Partial (Prefix) Reductions

Starting from:



Inclusive Scan (`MPI_SCAN`)

0	A_0
1	$A_0 \oplus A_1$
2	$A_0 \oplus A_1 \oplus A_2$
. . .	…
p-2	$A_0 \oplus A_1 \oplus \ldots \oplus A_{p-2}$
p-1	$A_0 \oplus A_1 \oplus \ldots \oplus A_{p-2} \oplus A_{p-1}$

Includes local data

Exclusive Scan (`MPI_EXSCAN`)

0	**"0"** ($\oplus$-Identity)
1	A_0
2	$A_0 \oplus A_1$
. . .	…
p-2	$A_0 \oplus A_1 \oplus \ldots \oplus A_{p-3}$
p-1	$A_0 \oplus A_1 \oplus \ldots \oplus A_{p-3} \oplus A_{p-2}$

Excludes local data

Additional Notes on Collective Operations

- Ordering of Collective Calls
  - MPI requires that collective routines on the same communicator be called by ALL processes in the communicator, in the same order
- Synchronization
  - No guarantees about the time at which different processes enter or exit most collective routines
  - Definitions of some "All" collectives may force them to be more strongly synchronized (Example: **`MPI_Allreduce`**)
  - **`MPI_Barrier`** only requires that all processes enter before any exit
- In-Place Operations
  - Not permitted to use same send and receive buffers
  - Use **`MPI_IN_PLACE`** instead
- **`MPI_Bcast`** & **`MPI_Recv`**
  - **`MPI_Recv`** cannot be used to receive data sent by **`MPI_Bcast`**