

Collective Communication

High-Performance Computing

Summer 2021 at GIST

Tae-Hyuk (Ted) Ahn

Department of Computer Science
Program of Bioinformatics and Computational Biology
Saint Louis University



**SAINT LOUIS
UNIVERSITY™**

— EST. 1818 —

MPI Collective Communications

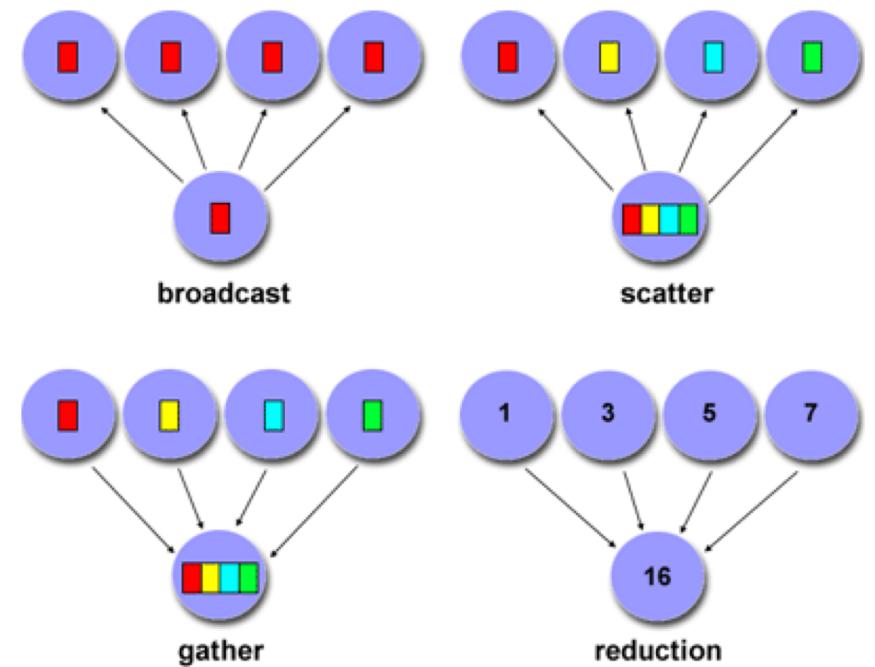
- **Point-to-Point (P2P) Communication** sends data from one point to another, one task sends while another receives.
 - **Blocking** (`MPI_Send()` and `MPI_Mrecv()`) send/routing will only return after it is safe to modify the buffer.
 - **Non-blocking** (`MPI_Isend()` and `MPI_Irecv()`) send/receive routines return immediately.
 - Improper use of blocking receive/send will result in **deadlock**, where two processors can't progress because each of them is waiting on the other.
- **Collective communication** is defined as communication between more than two (usually many) processors. A few forms:
 - One-to-many
 - Many-to-one
 - Many-to-many

Scope

- Collective communication routines must involve **all** processes within the scope of a communicator.
 - All processes are by default, members in the communicator MPI_COMM_WORLD.
 - Additional communicators can be defined by the programmer.
- Unexpected behavior, including program failure, can occur if even one task in the communicator doesn't participate.
- It is the programmer's responsibility to ensure that all processes within a communicator participate in any collective operations.

Types of Collective Operations

- **Synchronization** - processes wait until all members of the group have reached the synchronization point.
- **Data Movement** - broadcast, scatter/gather, all to all.
- **Collective computation** (reductions) - one member of the group collects data from the other members and performs an operation (min, max, add, multiply, etc.) on that data.



Barrier for Synchronization

- MPI has a special function that is dedicated to synchronizing processes:

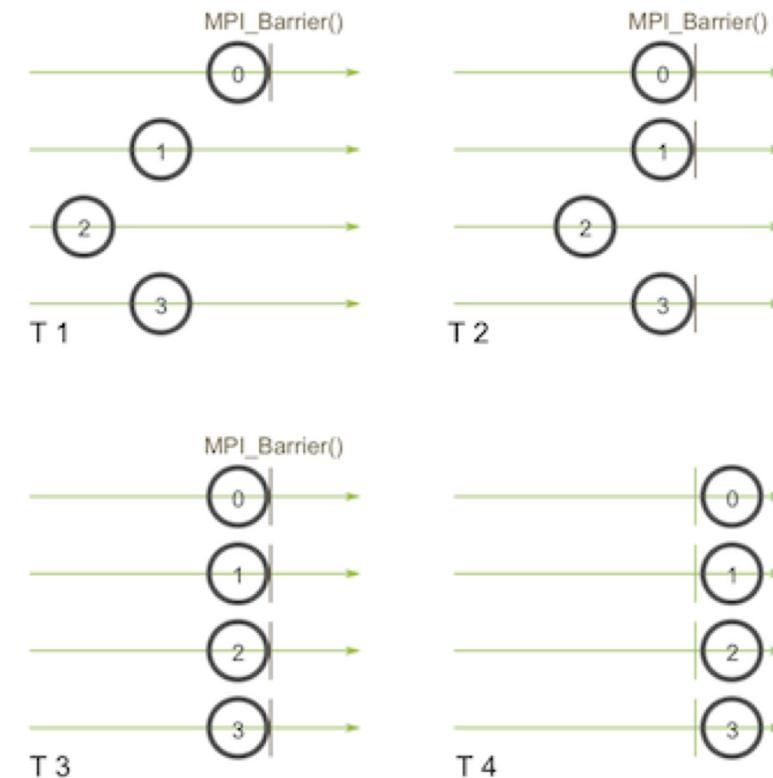
`MPI_Barrier(MPI_COMM comm)`

comm: the group of processes

- Blocks until all processes in the group call it.
- The name of the function is quite descriptive - the function forms a barrier, and no processes in the communicator can pass the barrier until all of them call the function.

Barrier for Synchronization

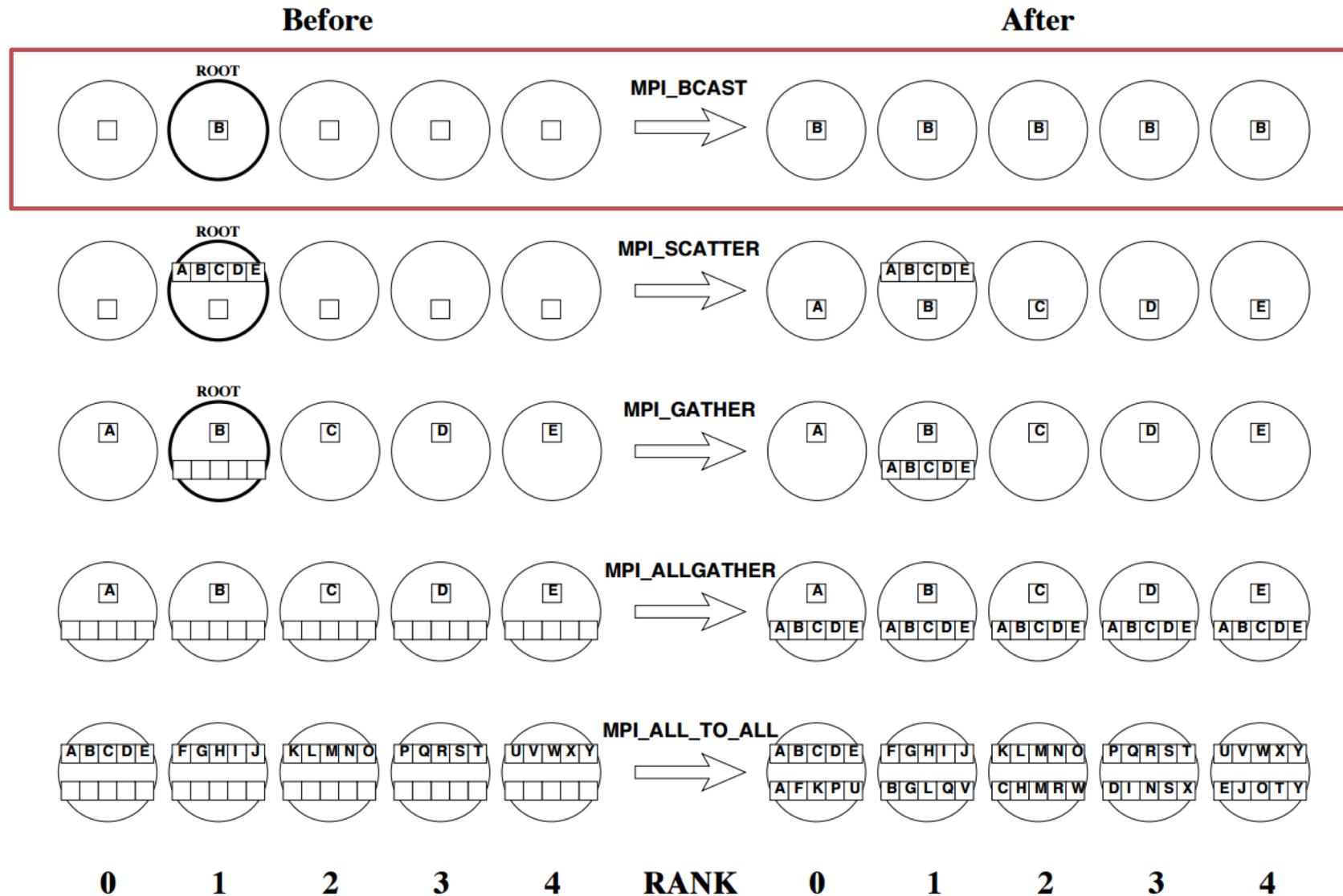
- Process zero first calls MPI_Barrier at the first time snapshot (T 1).
- While process zero is hung up at the barrier, process one and three eventually make it (T 2).
- When process two finally makes it to the barrier (T 3), all of the processes then begin execution again (T 4).



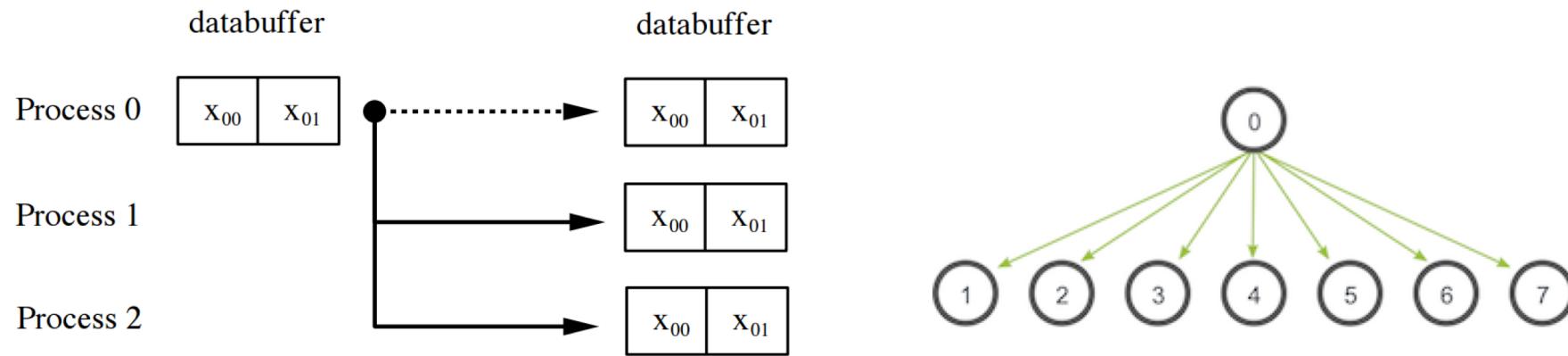
Barrier for Synchronization

- All collective operations in MPI are blocking, which means that it is safe to use all buffers passed to them after they return.
 - In particular, this means that all data was received when one of these functions returns. (However, it does not imply that all data was sent!) So `MPI_Barrier` is not necessary (or very helpful) before/after collective operations, if all buffers are valid already.
- Please also note, that `MPI_Barrier` does not magically wait for non-blocking calls.
 - If you use a non-blocking send/recv and both processes wait at an `MPI_Barrier` after the send/recv pair, it is not guaranteed that the processes sent/received all data after the `MPI_Barrier`. Use `MPI_Wait` (and friends) instead.

Data Movement



Broadcast



- A **broadcast** is one of the standard collective communication techniques.
- During a broadcast, one process sends the same data to all processes in a communicator.
- One of the main uses of broadcasting is to send out user input to a parallel program, or send out configuration parameters to all processes.

Broadcast: MPI_Bcast

- C Syntax: `MPI_Bcast(data, count, datatype, root, comm)`
- C++ Syntax: `MPI::Comm.Bcast(data, count, datatype, root)`
- No tag!!
- A broadcast has a specified root process and every process receives one copy of the message from the root.
- All processes must specify the same root (and communicator).
- The root argument is the rank of the root process.
- The buffer, count and datatype arguments are treated as in a point-to-point send on the root and as in a point-to-point receive elsewhere.

Lab: mpi_collective_bcast.cpp

```
# include <iostream>
# include <cstdlib> // has exit(), etc.
# include <ctime>
# include "mpi.h"    // MPI header file

using namespace std;

//*****80
int main (int argc, char **argv)
//*****80
{
    int nprocs, rank;
    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &nprocs);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);

    int data = 0;
    cout << "Before Bcast, data = " << data << " in rank = " << rank << endl;

    if (rank == 0) { // root
        data = 100;
    }

    MPI_Bcast(&data, 1, MPI_INT, 0, MPI_COMM_WORLD);      // MPI_Bcast
    cout << "After Bcast, data = " << data << " in rank = " << rank << endl;

    MPI_Finalize();
    return 0;
}
```

Collective vs. P2P?

- Broadcasting with MPI_Send and MPI_Recv

```
if (rank_id == 0) {  
    data = 100;  
    for (int i = 1; i < num_proc; i++) { // send our data to everyone  
        MPI_Send(&data, count, datatype, i, tag, MPI_COMM_WORLD);  
    }  
} else {  
    // If we are a receiver process, receive the data from the root  
    MPI_Recv(&data, count, datatype, 0, tag, MPI_COMM_WORLD status);  
}
```

- Broadcasting with MPI_Bcast

```
if (rank_id == 0) {  
    data = 100;  
}  
MPI_Bcast(&data, 1, MPI_INT, 0, MPI_COMM_WORLD);
```

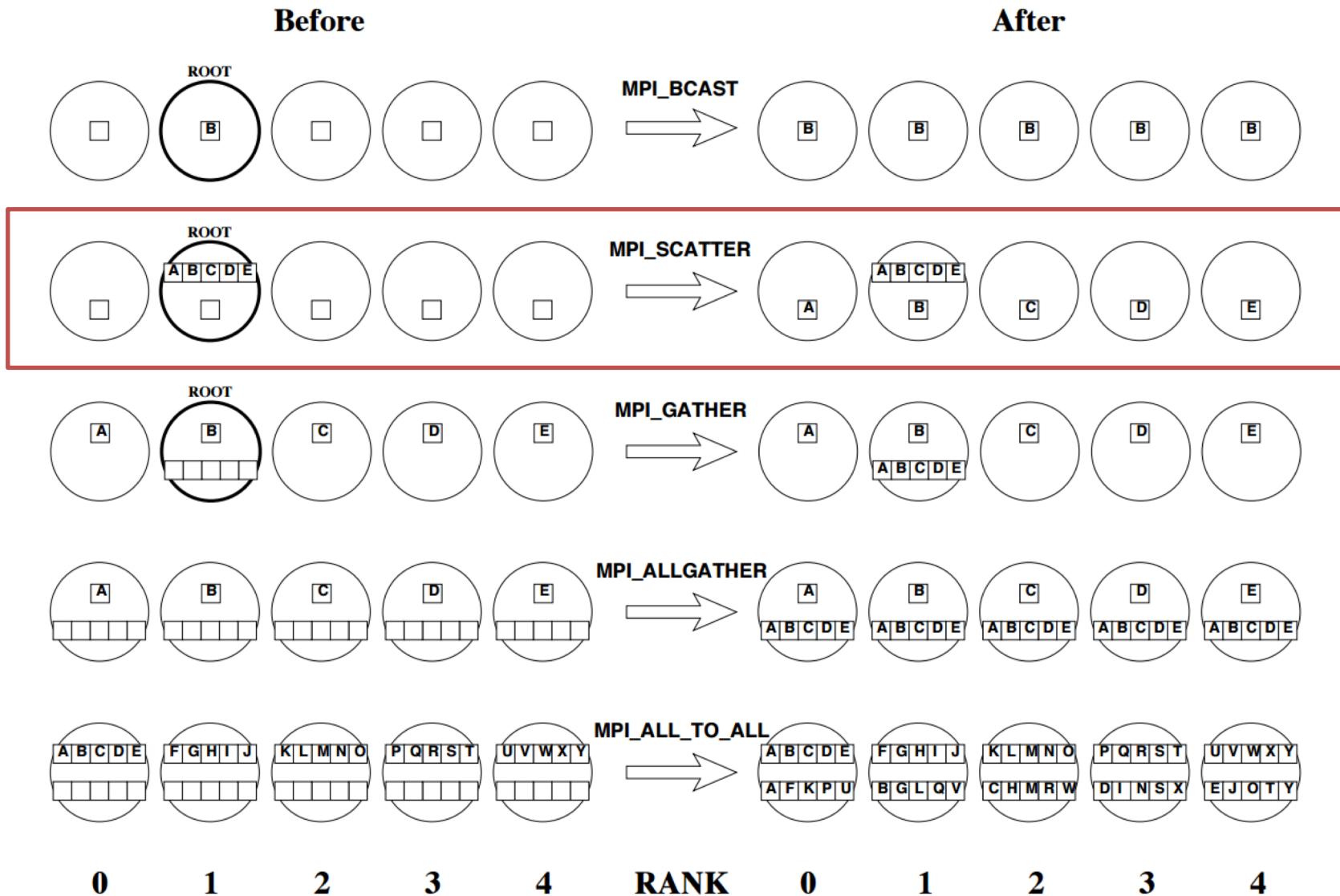
- Time comparison?

Collective vs. P2P?

- Time comparison example:

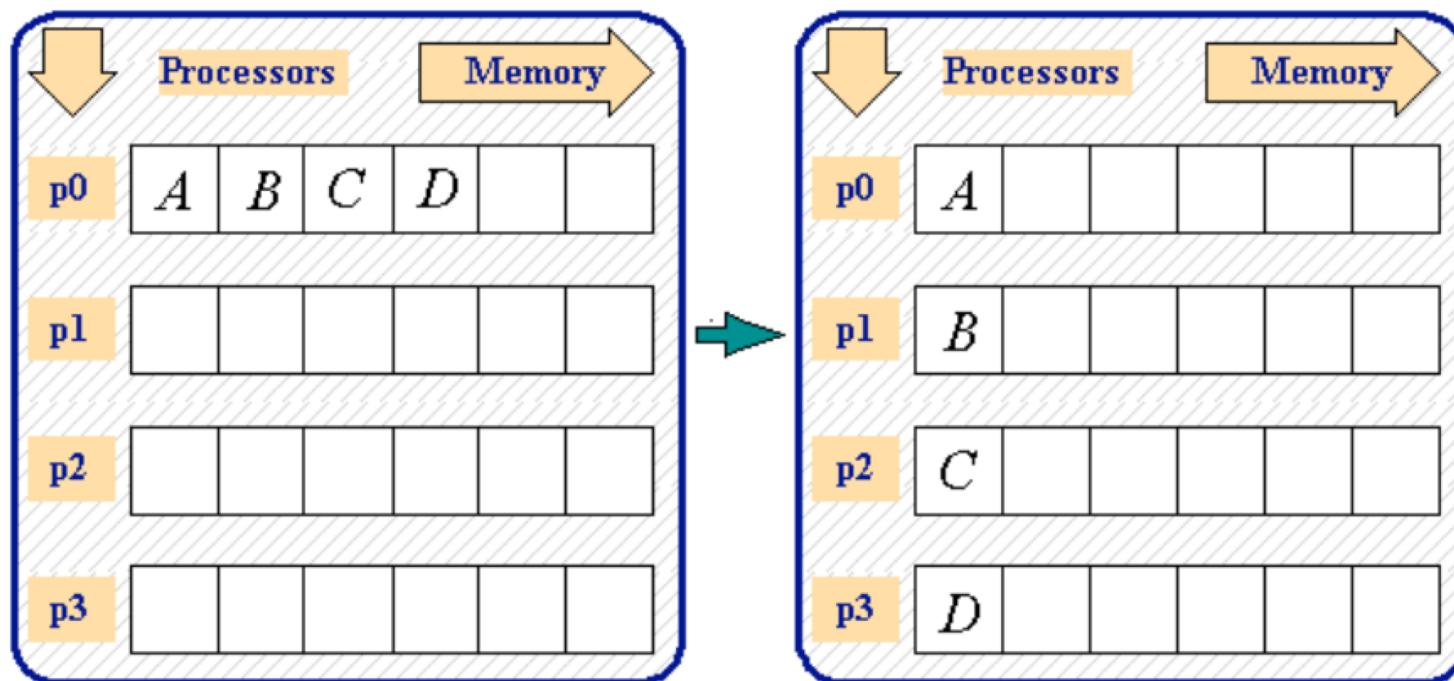
Processors	P2P broadcast	MPI_Bcast
2	0.0344	0.0344
4	0.1025	0.0817
8	0.2385	0.1084
16	0.5109	0.1296

Data Movement



Scatter: MPI_Scatter

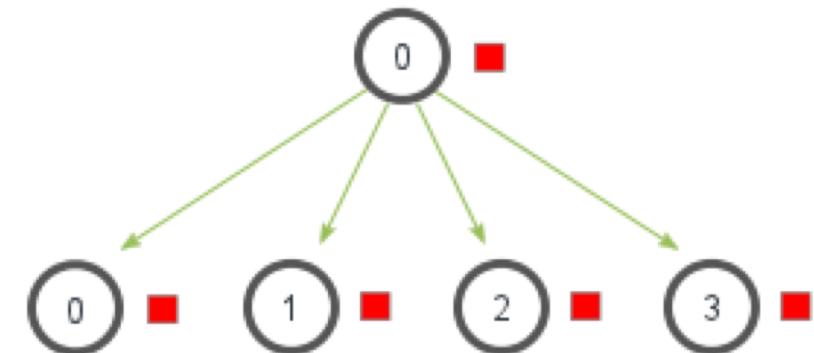
MPI_Scatter



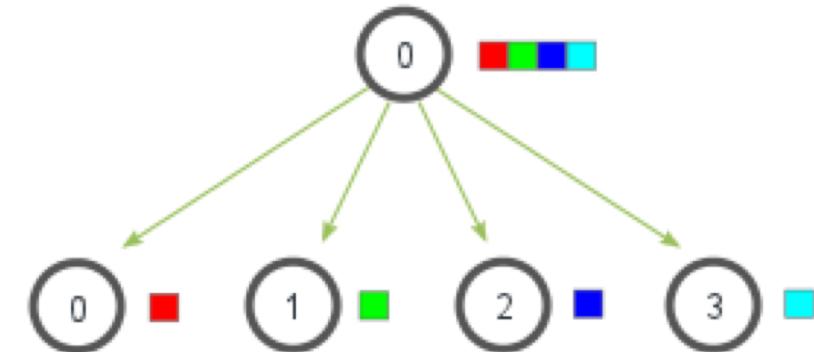
Scatter: MPI_Scatter

- MPI_Scatter involves a designated root process sending data to all processes in a communicator.
- MPI_Scatter sends *chunks of an array* to different processes.
- Check out the illustration below for further clarification.
- The main difference from MPI_Bcast is that the send and receive details are in general different and so must both be specified in the argument lists.

MPI_Bcast



MPI_Scatter



Scatter: MPI_Scatter

- C Syntax: `MPI_Scatter(sendbuffer, sendcount, sendtype, recvbuffer, recvcount, recvtype, root, comm)`
- C++ Syntax: `MPI::Comm.Scatter(sendbuffer, sendcount, sendtype, recvbuffer, recvcount, recvtype, root)`
- No tag!!
- Note that the sendcount (at the root) is the number of elements to send to each process, not to send in total. Therefore if `sendtype = recvtype`, `sendcount = recvcount`.
- The `sendbuf`, `sendcount`, `sendtype` arguments are significant only at the root. The `buffer`, `count` and `datatype` arguments are treated as in a point-to-point send on the root and as in a point-to-point receive elsewhere.

Lab: MPI_Scatter()

- Scattering with MPI_Scatter()
 - int **data_array**[8] = {};
 - int **recv_buffer**[8] = {};
 - data_array** is assigned to 100,...107 at **rank_id** = 0
 - scattering two elements to other processes that will be saved to **recv_buffer**.

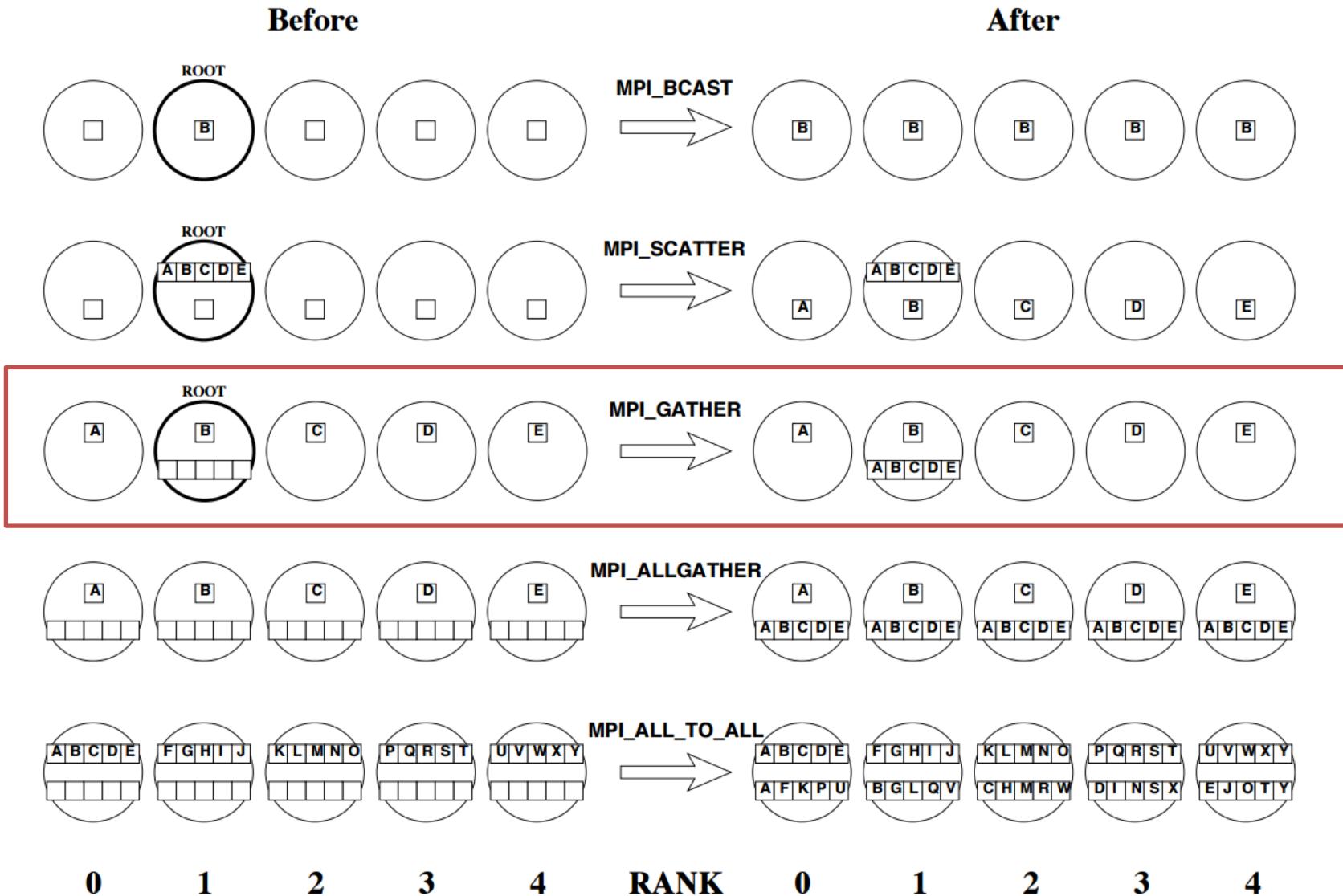
Lab: mpi_collective_scatter.cpp

```
1 # include <iostream>
2 # include <cstdlib> // has exit(), etc.
3 # include <ctime>
4 # include "mpi.h"    // MPI header file
5
6 using namespace std;
7
8 int main (int argc, char **argv)
9 {
10    int data_array[8] = {};
11    int recv_buffer[8] = {};
12    int nprocs, rank;
13
14    MPI_Init(&argc, &argv);
15    MPI_Comm_size(MPI_COMM_WORLD, &nprocs);
16    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
17
18    if (rank == 0) { // root
19        for (int i=0; i<8; i++) {
20            data_array[i] = 100+i;
21        }
22    }
23
24    int send_count = 2;
25    int recv_count = 2;
```

```
26
27    cout << "Before Scatter, data_array = ";
28    for (int j=0; j<8; j++) {cout << data_array[j] << " ";}
29    cout << " in rank = " << rank << endl;
30    cout << "Before Scatter, recv_buffer = ";
31    for (int j=0; j<8; j++) {cout << recv_buffer[j] << " ";}
32    cout << " in rank = " << rank << endl;
33
34    MPI_Scatter(&data_array, send_count, MPI_INT, &recv_buffer,
35    recv_count, MPI_INT, 0, MPI_COMM_WORLD);
36
37    cout << "After Scatter, data_array = ";
38    for (int j=0; j<8; j++) {cout << data_array[j] << " ";}
39    cout << " in rank = " << rank << endl;
40    cout << "After Scatter, recv_buffer = ";
41    for (int j=0; j<8; j++) {cout << recv_buffer[j] << " ";}
42    cout << " in rank = " << rank << endl;
43
44    MPI_Finalize();
45 }
```

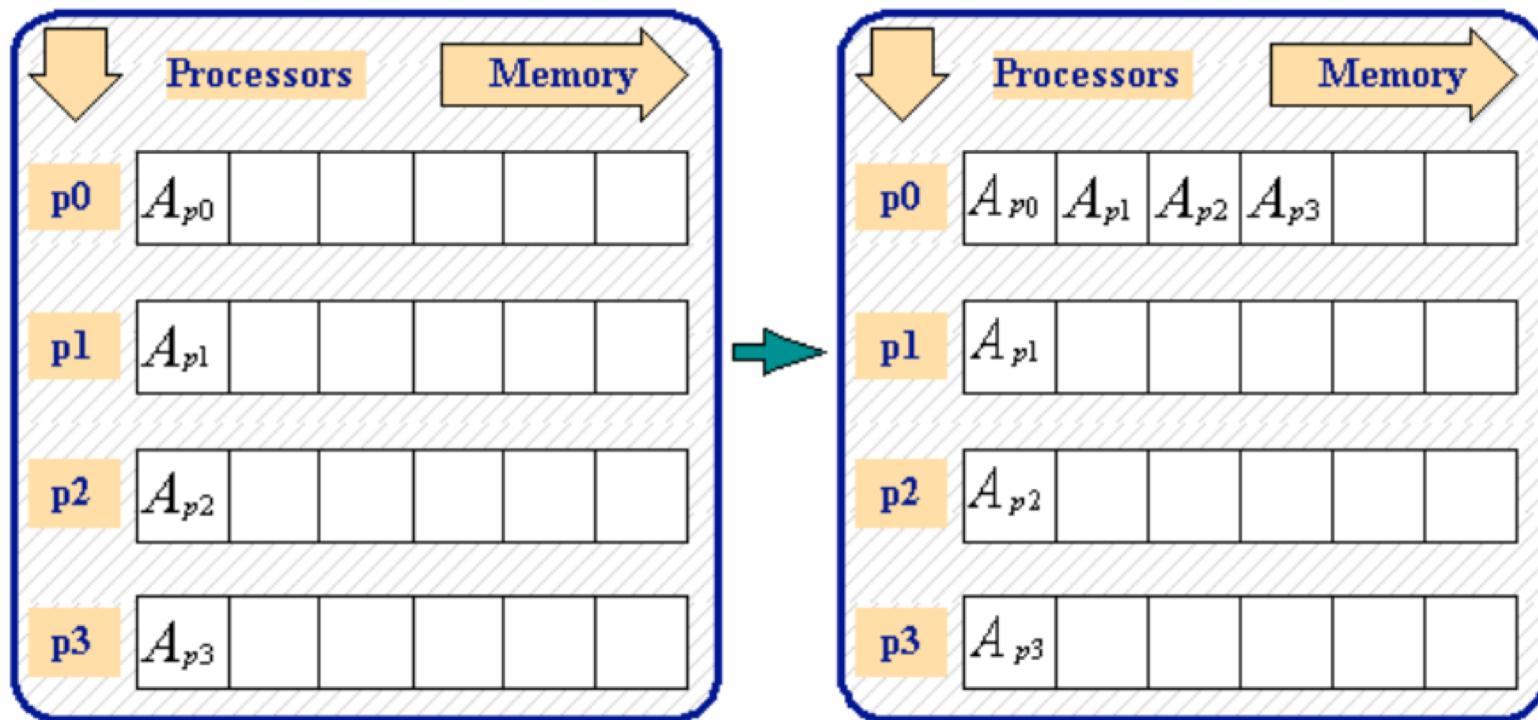
- Let us check the results together!

Data Movement



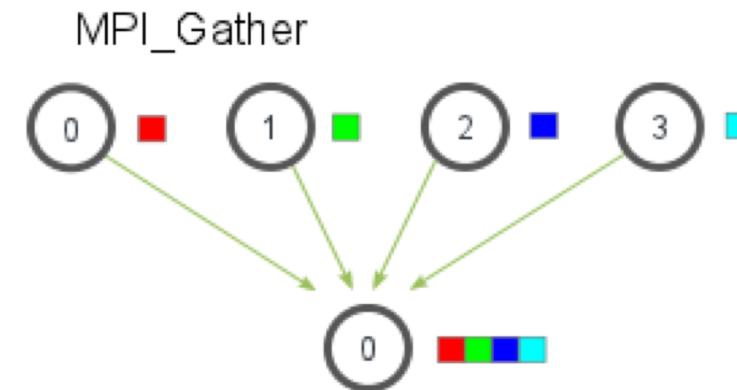
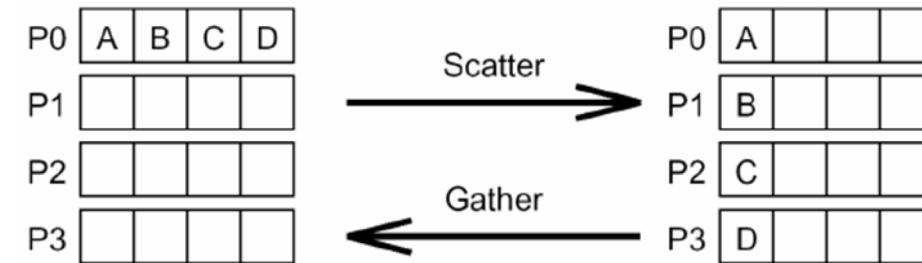
Gather: MPI_Gather

MPI_Gather



Gather: MPI_Gather

- MPI_Gather is the inverse of MPI_Scatter.
- MPI_Gather takes elements from many processes and gathers them to one single process.
- This routine is highly useful to many parallel algorithms, such as parallel sorting and searching.



Gather: MPI_Gather

- C Syntax: `MPI_Gather(sendbuffer, sendcount, sendtype, recvbuffer, recvcount, recvtype, root, comm)`
- C++ Syntax: `MPI::Comm.Gather(sendbuffer, sendcount, sendtype, recvbuffer, recvcount, recvtype, root)`
- No tag!!
- Similar to MPI_Scatter, MPI_Gather takes elements from each process and gathers them to the root process.
- The elements are ordered by the rank of the process from which they were received.

Lab: MPI_Gather()

- Gathering with MPI_Gather()

- `int data_array[8] = {};`
- `int recv_buffer[8] = {};`
- `data_array` is assigned to 100,101 at `rank_id = 0`,
102, 103 at `rank_id = 1`, ..., 106, 107 at `rank_id = 3`.
- Gathering all elements to root (`rank_id = 0`) and compute average of them.

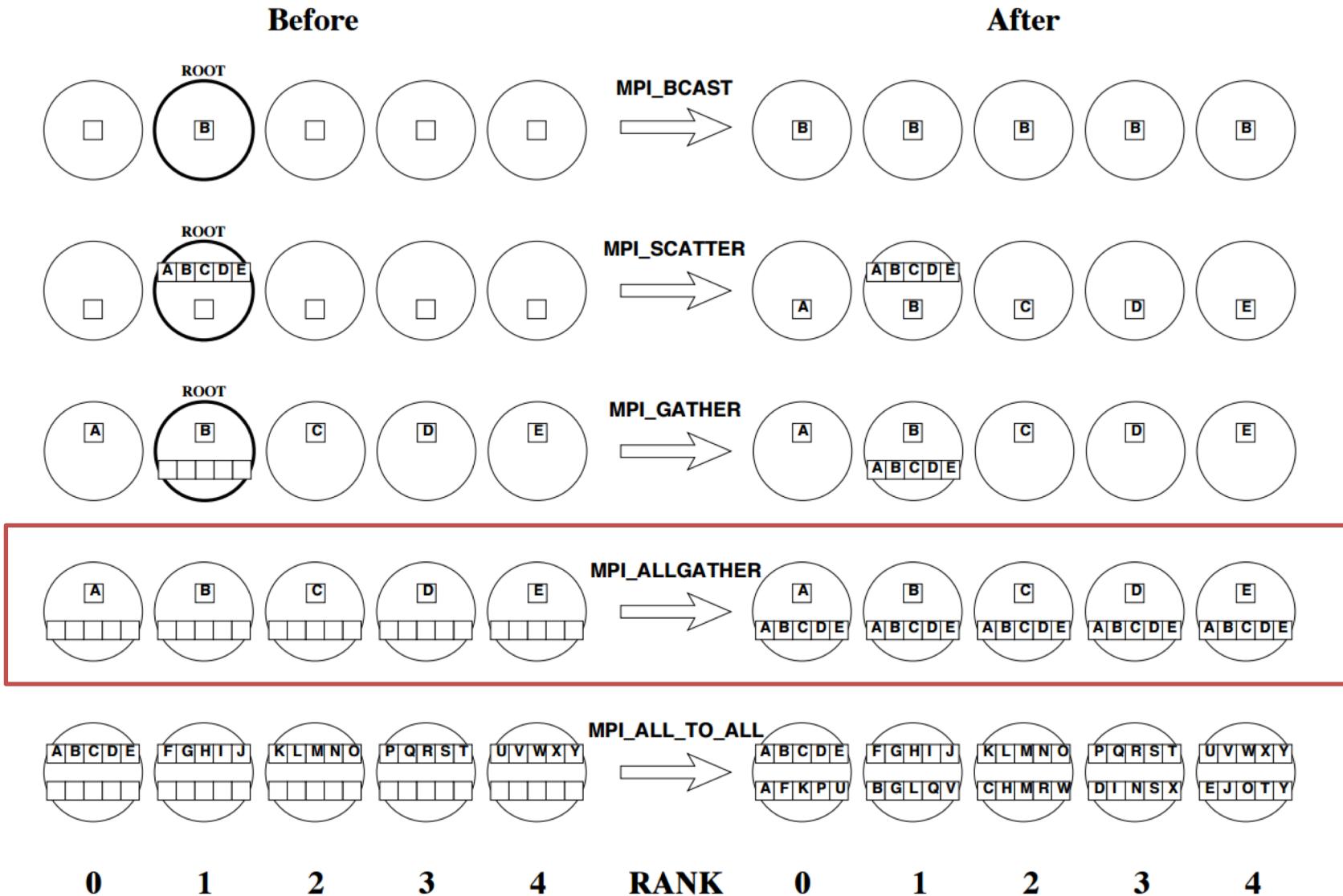
Lab: mpi_collective_gather.cpp

```
1 # include <iostream>
2 # include <cstdlib> // has exit(), etc.
3 # include <ctime>
4 # include "mpi.h"    // MPI header file
5
6 using namespace std;
7
8 int main (int argc, char **argv)
9 {
10    int data_array[8] = {};
11    int recv_buffer[8] = {};
12    int nprocs, rank;
13
14    MPI_Init(&argc, &argv);
15    MPI_Comm_size(MPI_COMM_WORLD, &nprocs);
16    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
17
18    if (rank == 0) { // root
19        for (int i=0; i<8; i++) {
20            data_array[i] = 100+i;
21        }
22    }
23
24    int send_count = 2;
25    int recv_count = 2;
```

```
26
27    cout << "Before Scatter, data_array = ";
28    for (int j=0; j<8; j++) {cout << data_array[j] << " ";}
29    cout << " in rank = " << rank << endl;
30    cout << "Before Scatter, recv_buffer = ";
31    for (int j=0; j<8; j++) {cout << recv_buffer[j] << " ";}
32    cout << " in rank = " << rank << endl;
33
34    MPI_Scatter(&data_array, send_count, MPI_INT, &recv_buffer,
35    recv_count, MPI_INT, 0, MPI_COMM_WORLD);
36
37    cout << "After Scatter, data_array = ";
38    for (int j=0; j<8; j++) {cout << data_array[j] << " ";}
39    cout << " in rank = " << rank << endl;
40    cout << "After Scatter, recv_buffer = ";
41    for (int j=0; j<8; j++) {cout << recv_buffer[j] << " ";}
42    cout << " in rank = " << rank << endl;
43
44    MPI_Finalize();
45 }
```

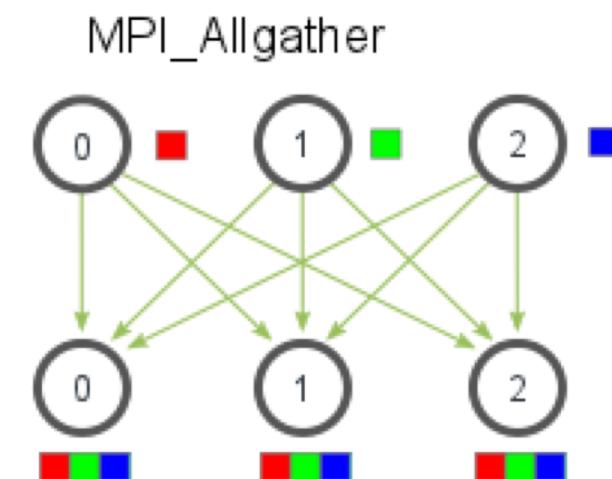
- Let us check the results together!

Data Movement



Gather: MPI_Allgather

- So far, we have covered two MPI routines that perform *many-to-one* or *one-to-many* communication patterns, which simply means that many processes send/receive to one process.
- Oftentimes it is useful to be able to send many elements to many processes (i.e. a *many-to-many* communication pattern).
- MPI_Allgather has this characteristic.
- Given a set of elements distributed across all processes, MPI_Allgather will gather all of the elements to all the processes.



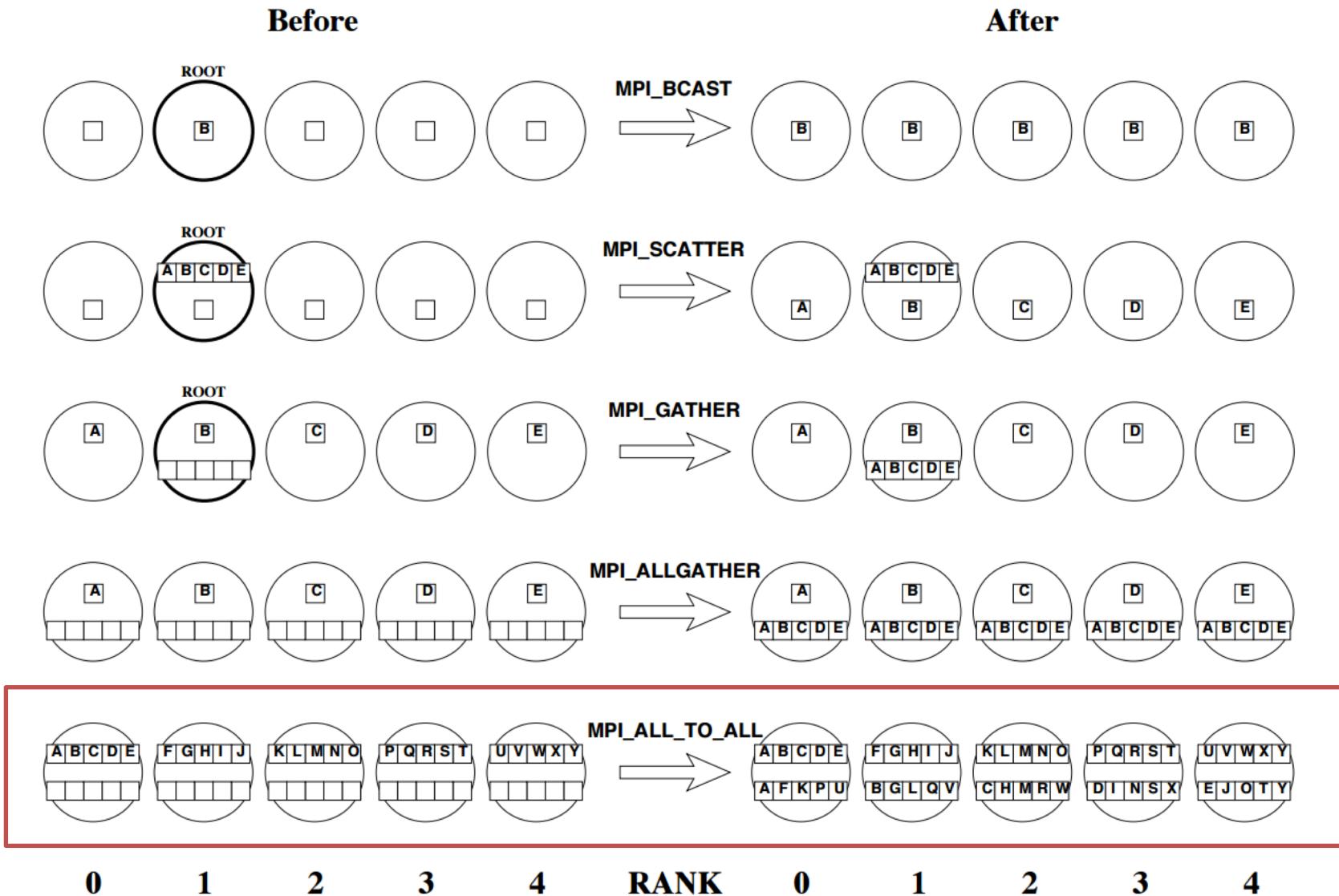
Gather: MPI_Allgather

- C Syntax: `MPI_Allgather(sendbuffer, sendcount, sendtype, recvbuffer, recvcount, recvtype, comm)`
- C++ Syntax: `MPI::Comm.Allgather(sendbuffer, sendcount, sendtype, recvbuffer, recvcount, recvtype)`
- Just like `MPI_Gather`, the elements from each process are gathered in order of their rank, except this time the elements are gathered to all processes.
- **No root, No tag!!**
- In the most basic sense, `MPI_Allgather` is an `MPI_Gather` followed by an `MPI_Bcast`.

Assignment: MPI_Allgather()

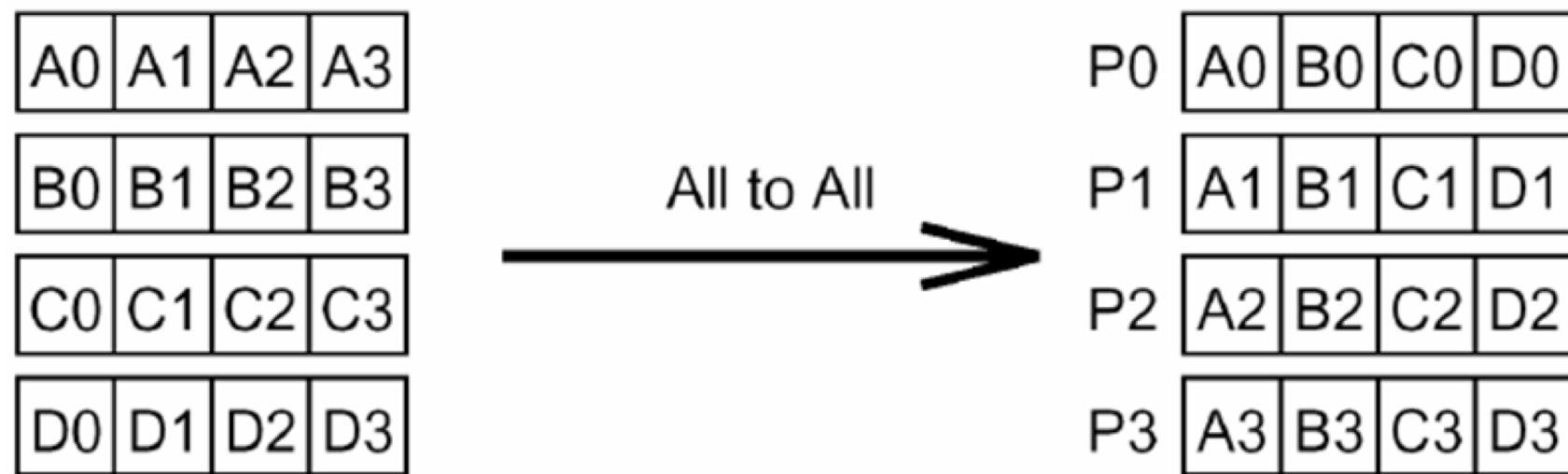
- Gathering with MPI_Allgather()
 - `int data_array[8] = {};`
 - `int recv_buffer[8] = {};`
 - `data_array` is assigned to 100,101 at `rank_id = 0`,
102, 103 at `rank_id = 1`, ..., 106, 107 at `rank_id = 3`.
 - Gathering all elements to all processes and compute average of them at `rank_id = 1`.

Data Movement



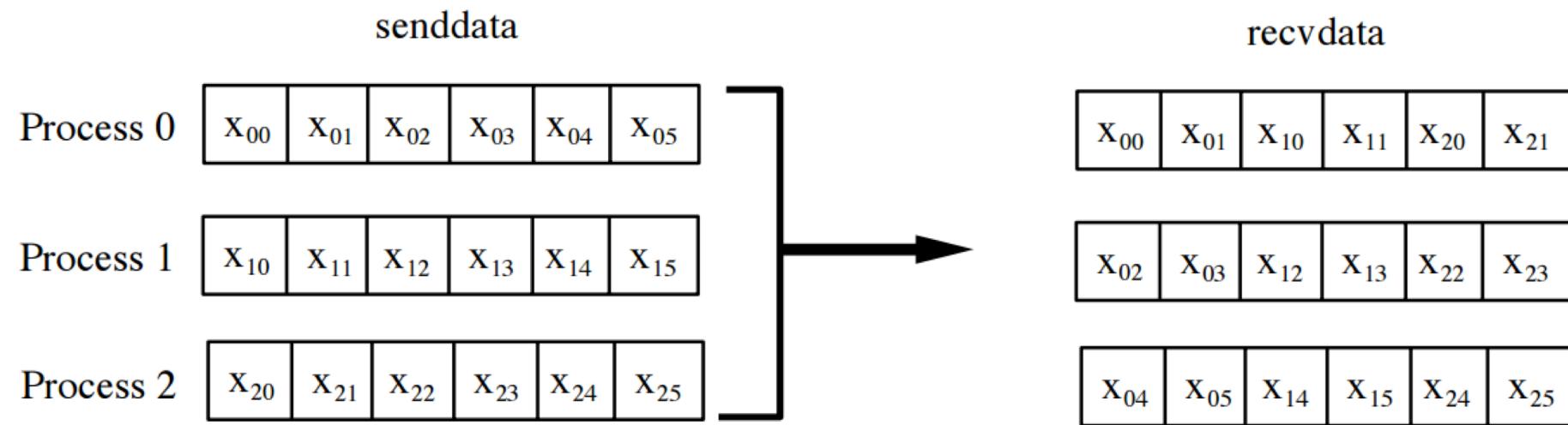
MPI_Alltoall

- Data movement operation.
- Each task in a group performs a scatter operation, sending a distinct message to all the tasks in the group in order by index.



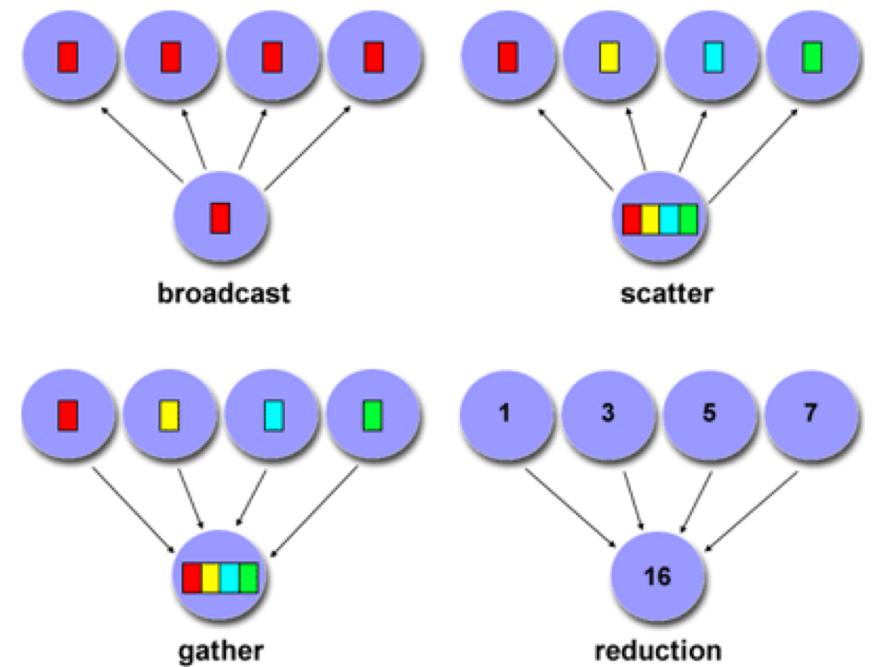
MPI_Alltoall

- C Syntax: MPI_Alltoall(sendbuffer, sendcount, sendtype, recvbuffer, recvcount, recvtype, comm)
- C++ Syntax: MPI::Comm.Alltoall(sendbuffer, sendcount, sendtype, recvbuffer, recvcount, recvtype)
- No root, No tag!!

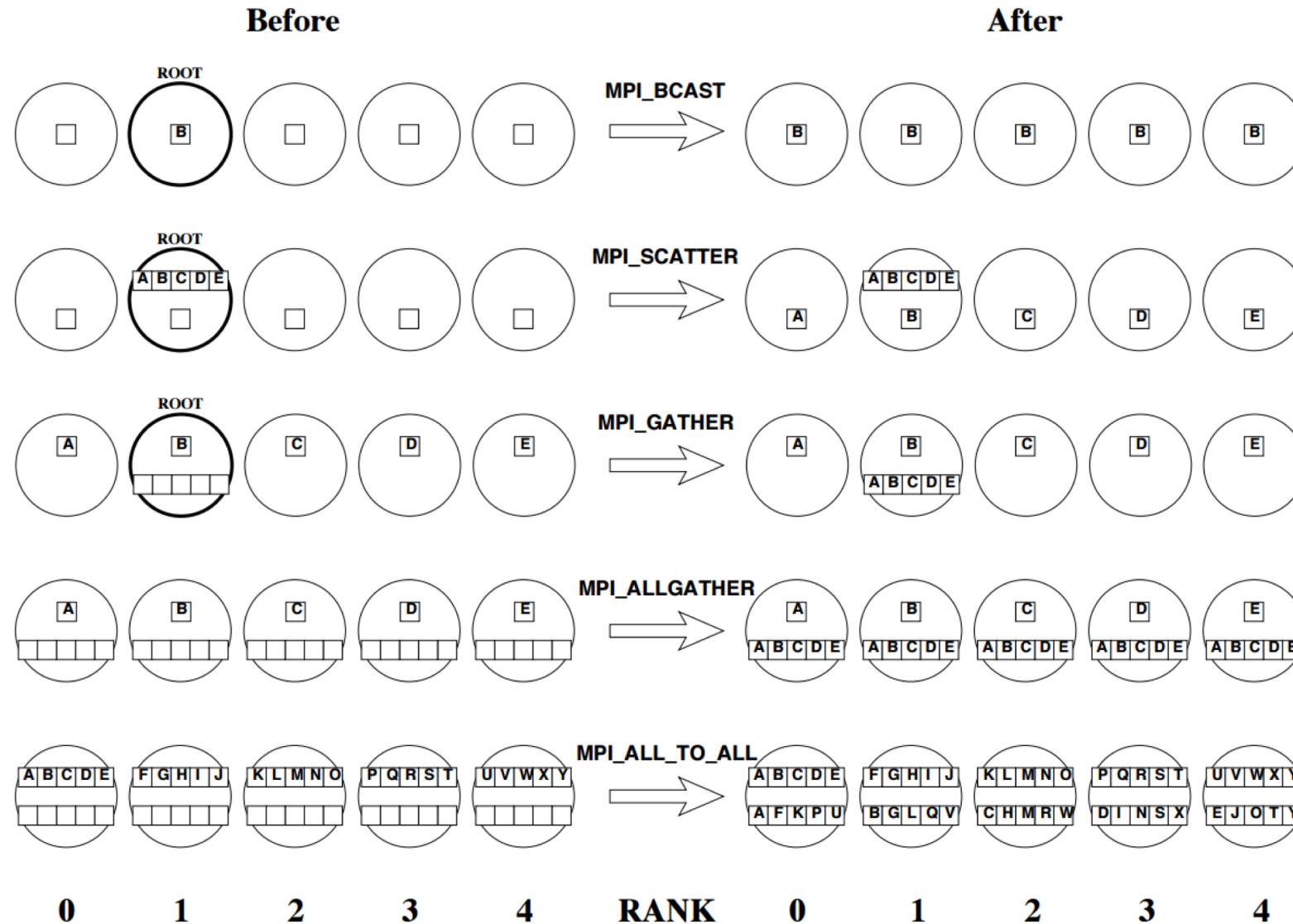


Types of Collective Operations

- **Synchronization** - processes wait until all members of the group have reached the synchronization point.
- **Data Movement** - broadcast, scatter/gather, all to all.
- **Collective computation** (reductions) - one member of the group collects data from the other members and performs an operation (min, max, add, multiply, etc.) on that data.

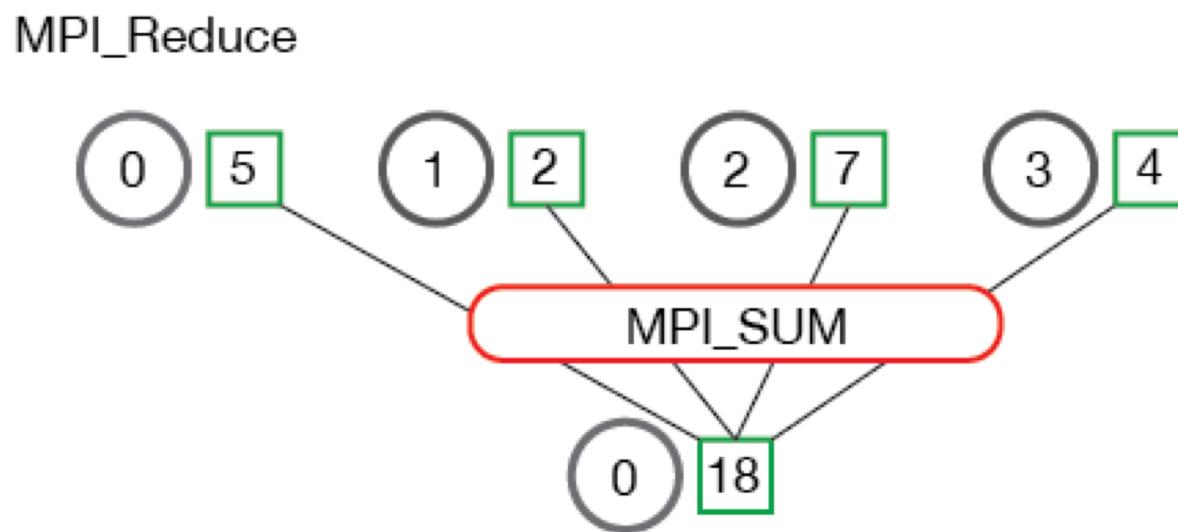


Recall: Data Movement

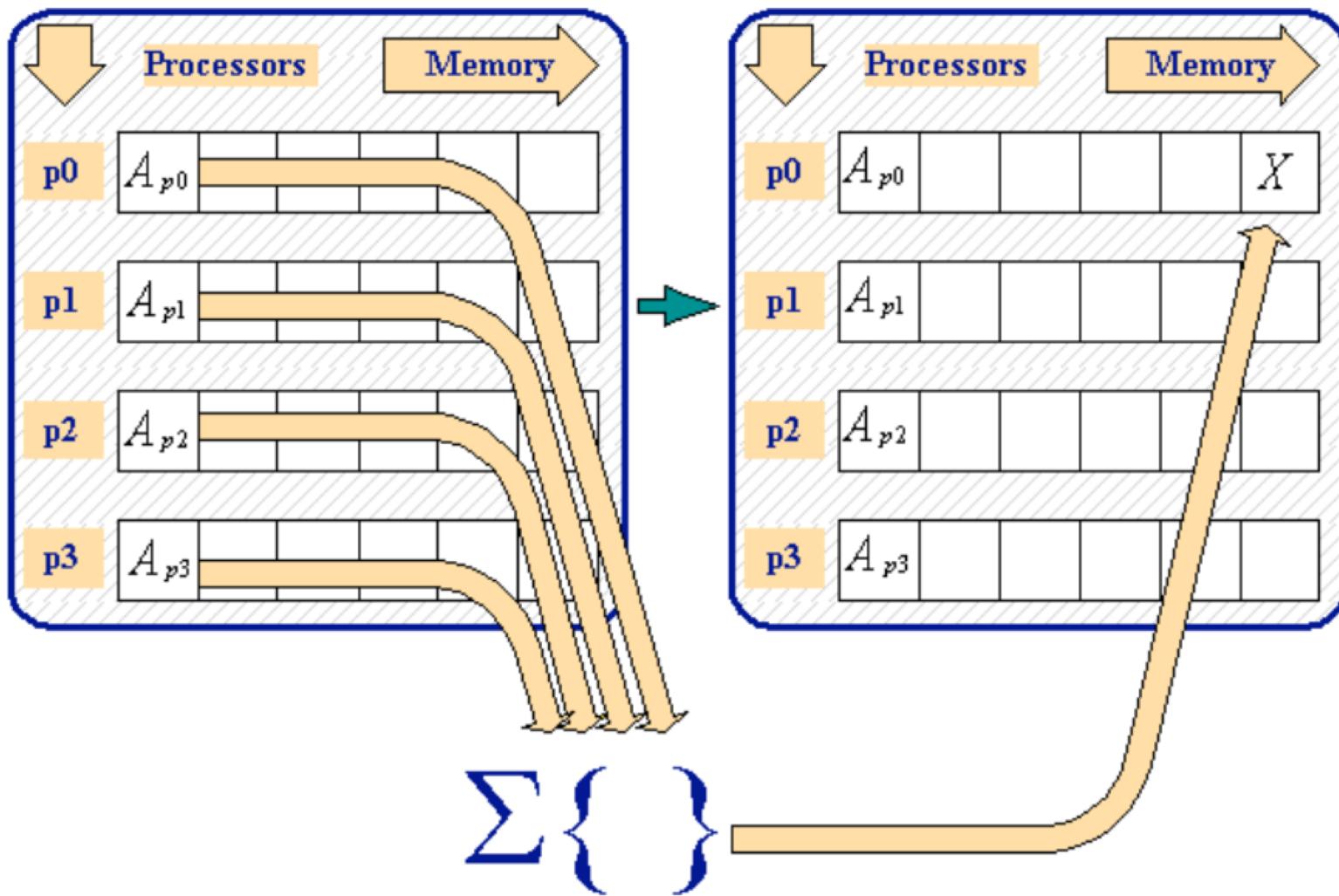


MPI_Reduce

- Collective computation operation.
- Applies a reduction operation on all tasks in the group and places the result in one task.



MPI_Reduce



MPI_Reduce

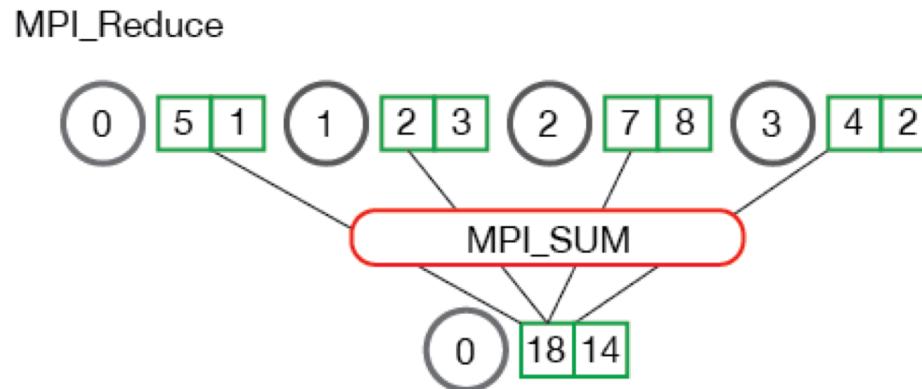
- C Syntax: `MPI_Reduce(sendbuffer, recvbuffer, count, datatype, operation, root, comm)`
- C++ Syntax: `MPI::Reduce (sendbuffer, recvbuffer, count, datatype, operation, root)`
- No tag!!
- All processes must specify the same root (and communicator).
- The sendbuffer parameter is an array of elements of type datatype that each process wants to reduce.
- The recvbuffer is only relevant on the process with a rank of root.

MPI Reduction Operations

- MPI_MAX - Returns the maximum element.
- MPI_MIN - Returns the minimum element.
- MPI_SUM - Sums the elements.
- MPI_PROD - Multiplies all elements.
- MPI_LAND - Performs a logical *and* across the elements.
- MPI_LOR - Performs a logical *or* across the elements.
- MPI_BAND - Performs a bitwise *and* across the bits of the elements.
- MPI_BOR - Performs a bitwise *or* across the bits of the elements.
- MPI_MAXLOC - Returns the maximum value and the rank of the process that owns it.
- MPI_MINLOC - Returns the minimum value and the rank of the process that owns it.

MPI_Reduce

- It is also useful to see what happens when processes contain multiple elements.
- The illustration below shows reduction of multiple numbers per process.



- The processes from the above illustration each have two elements.
- The resulting summation happens on a per-element basis. In other words, instead of summing all of the elements from all the arrays into one element, the i^{th} element from each array are summed into the i^{th} element in result array of process 0.

Lab: Sum of the N (1000) Integers using MPI P2P Block

```
#include <iostream>
# include <cstdlib>      // has exit(), etc.
# include <ctime>
#include <mpi.h>
#define MASTER 0

using namespace std;

int main(int argc, char **argv)
{
    int N = 1000;
    int nprocs, rank, tag=0;
    int tmp_sum = 0, partial_sum = 0, total_sum = 0;;
    int startval = 0, endval = 0;

    // MPI init
    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &nprocs);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);

    // Work
    startval = N*rank/nprocs + 1;
    endval   = N*(rank+1)/nprocs;
    //Need to modify based on number of processors
    for(int i=startval; i<=endval; i++)
    {
        partial_sum += i;
    }
}
```

```
// Collect partial sum from each worker
if (rank == MASTER)
{
    total_sum = partial_sum;           // master should have it's own partial sum
    for (int worker_rank = 1; worker_rank < nprocs; worker_rank++)
    {
        MPI_Status status;
        MPI_Recv(&partial_sum, 1, MPI_INT, worker_rank, tag, MPI_COMM_WORLD, &status);
        total_sum += partial_sum; // master should add each partial sum from workers
    }
    cout << "The sum from 1 to " << N << " is: " << total_sum << endl;
}
else
{
    MPI_Send(&partial_sum, 1, MPI_INT, MASTER, tag, MPI_COMM_WORLD);
}

MPI_Finalize();
}
```

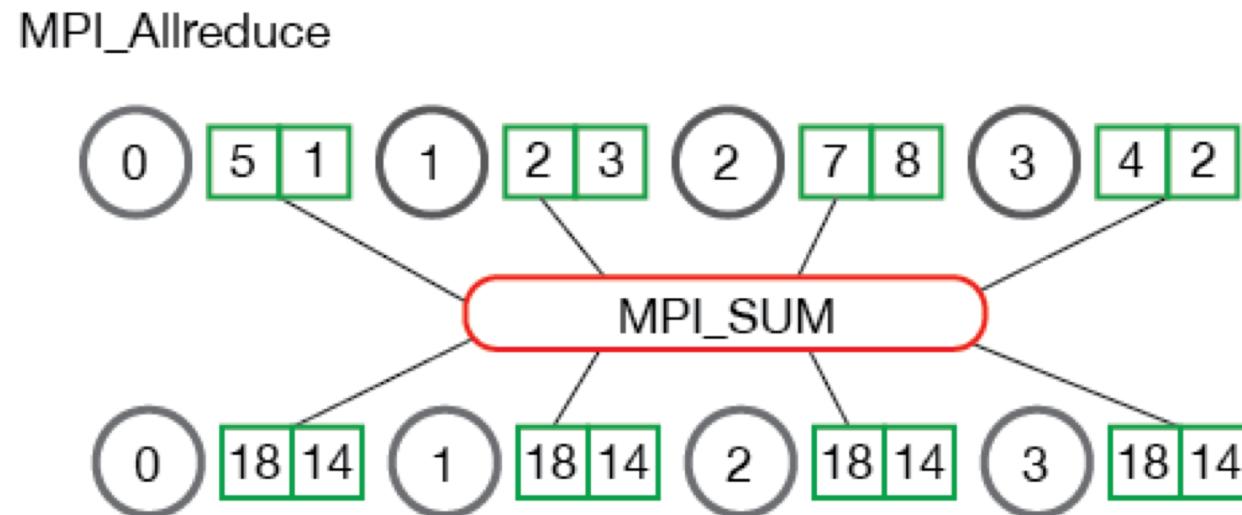
Lab: Sum of the N (100000000) using MPI_Reduce

```
1 #include <iostream>
2 #include <mpi.h>           // MPI header file
3 #define MASTER 0
4 #define N 100000000
5
6 using namespace std;
7
8 int main(int argc, char **argv) {
9     int nprocs, rank;
10    long long int start_val, end_val, total_sum = 0, partial_sum = 0;
11
12    // initialize for MPI
13    MPI_Init(&argc, &argv);
14    // get number of processes
15    MPI_Comm_size(MPI_COMM_WORLD, &nprocs);
16    // get the rank = this process's number (ranges from 0 to nprocs - 1)
17    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
18    // MPI status
19    MPI_Status status;
20
21    // each process do "partial sum" with
22    start_val = rank * (N/nprocs) + 1;
23    end_val = (rank+1) * (N/nprocs);
24    for (int val=start_val; val<=end_val; val++)
25        partial_sum += val;
26    cout << "rank " << rank << " calculated partial_sum " << partial_sum << endl;
27
```

```
// Your work
MPI_Finalize();
}
```

MPI_Allreduce

- Many parallel applications will require accessing the reduced results across all processes rather than the root process.
- In a similar complementary style of MPI_Allgather to MPI_Gather, MPI_Allreduce will reduce the values and distribute the results to all processes.



MPI_Allreduce

- C Syntax: `MPI_Allreduce(sendbuffer, recvbuffer, count, datatype, operation, comm)`
- C++ Syntax: `MPI:: COMM.Allreduce(sendbuffer, recvbuffer, count, datatype, operation)`
- No tag, **no root!!**
- `MPI_Allreduce` is identical to `MPI_Reduce` with the exception that it does not need a root process id (since the results are distributed to all processes).

Lab: Sum of the first N Integers using MPI_Allreduce and Print at Rank 1

```
1 #include <iostream>
2 #include <mpi.h>           // MPI header file
3 #define MASTER 0
4 #define N 100000000
5
6 using namespace std;
7
8 int main(int argc, char **argv) {
9     int nprocs, rank;
10    long long int start_val, end_val, total_sum = 0, partial_sum = 0;
11
12    // initialize for MPI
13    MPI_Init(&argc, &argv);
14    // get number of processes
15    MPI_Comm_size(MPI_COMM_WORLD, &nprocs);
16    // get the rank = this process's number (ranges from 0 to nprocs - 1)
17    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
18    // MPI status
19    MPI_Status status;
20
21    // each process do "partial sum" with
22    start_val = rank * (N/nprocs) + 1;
23    end_val = (rank+1) * (N/nprocs);
24    for (int val=start_val; val<=end_val; val++)
25        partial_sum += val;
26    cout << "rank " << rank << " calculated partial_sum " << partial_sum << endl;
27
```

```
// Your work
MPI_Finalize();
}
```

Collective vs. Point-to-Point Communications

- All the processes in the communicator must call the same collective function.
- For example, a program that attempts to match a call to `MPI_Reduce` on one process with a call to `MPI_Recv` on another process is erroneous, and, in all likelihood, the program will hang or crash.

Collective vs. Point-to-Point Communications

- The arguments passed by each process to an MPI collective communication must be “compatible.”
- For example, if one process passes in 0 as the `dest_process` and another passes in 1, then the outcome of a call to `MPI_Reduce` is erroneous, and, once again, the program is likely to hang or crash.

Collective vs. Point-to-Point Communications

- Point-to-point communications are matched on the basis of **tags and communicators**.
- Collective communications **don't use tags**.
- They're matched solely on the basis of the communicator and the order in which they're called.

Matrix-vector multiplication

$A = (a_{ij})$ is an $m \times n$ matrix

$\xrightarrow{\hspace{1cm}}$
 \mathbf{x} is a vector with n components

$\mathbf{y} = A\mathbf{x}$ is a vector with m components

$$y_i = a_{i0}x_0 + a_{i1}x_1 + a_{i2}x_2 + \cdots a_{i,n-1}x_{n-1}$$

i-th component of y

Dot product of the ith row of A with x.

Matrix-vector multiplication

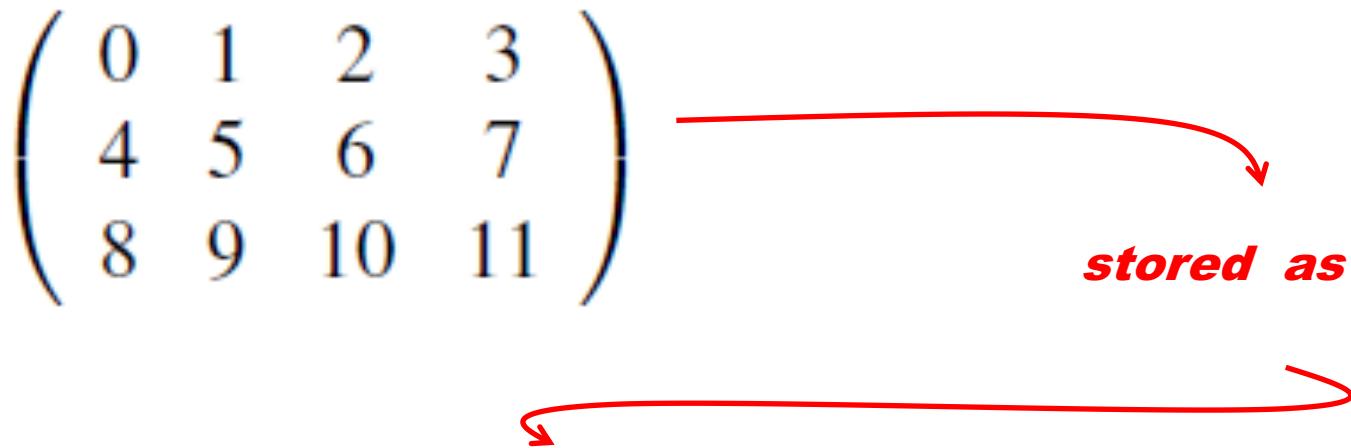
$$\begin{array}{|c|c|c|c|} \hline a_{00} & a_{01} & \cdots & a_{0,n-1} \\ \hline a_{10} & a_{11} & \cdots & a_{1,n-1} \\ \hline \vdots & \vdots & & \vdots \\ \hline a_{i0} & a_{i1} & \cdots & a_{i,n-1} \\ \hline \vdots & \vdots & & \vdots \\ \hline a_{m-1,0} & a_{m-1,1} & \cdots & a_{m-1,n-1} \\ \hline \end{array} \begin{matrix} x_0 \\ x_1 \\ \vdots \\ x_{n-1} \end{matrix} = \begin{array}{|c|} \hline y_0 \\ \hline y_1 \\ \hline \vdots \\ \hline y_i = a_{i0}x_0 + a_{i1}x_1 + \cdots + a_{i,n-1}x_{n-1} \\ \hline \vdots \\ \hline y_{m-1} \\ \hline \end{array}$$

Multiply a matrix by a vector

```
/* For each row of A */
for (i = 0; i < m; i++) {
    /* Form dot product of ith row with x */
    y[i] = 0.0;
    for (j = 0; j < n; j++)
        y[i] += A[i][j]*x[j];
}
```

Serial pseudo-code

C style arrays



0 1 2 3 4 5 6 7 8 9 10 11

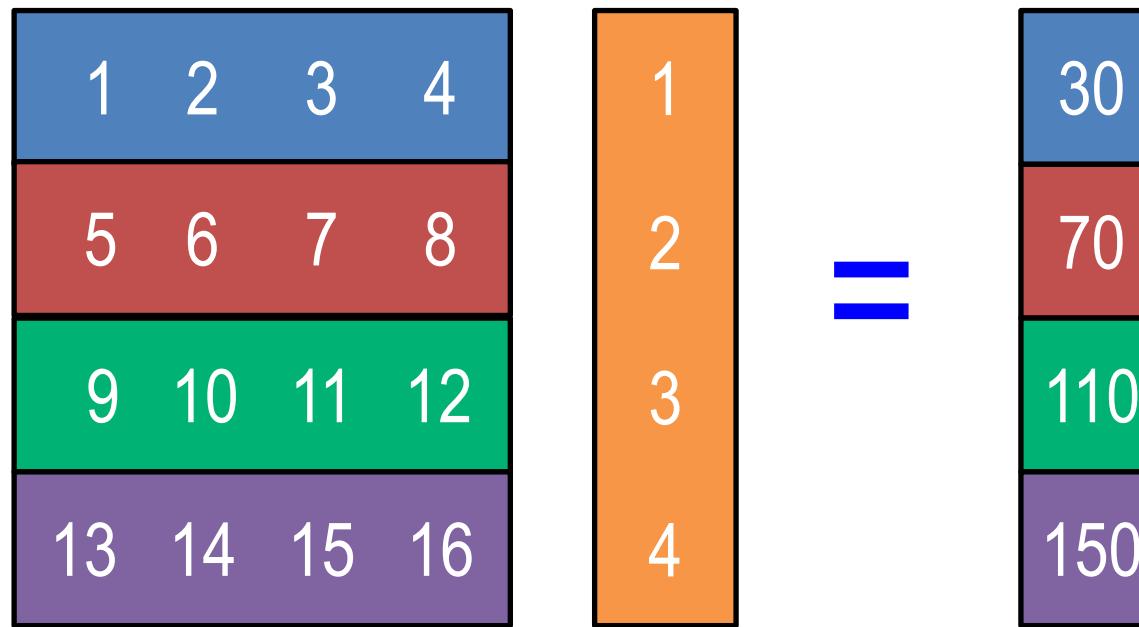
Serial matrix-vector multiplication

```
void Mat_vect_mult(
    double A [] /* in */,
    double x [] /* in */,
    double y [] /* out */,
    int m /* in */,
    int n /* in */) {
    int i, j;

    for (i = 0; i < m; i++) {
        y[i] = 0.0;
        for (j = 0; j < n; j++)
            y[i] += A[i*n+j]*x[j];
    }
} /* Mat_vect_mult */
```

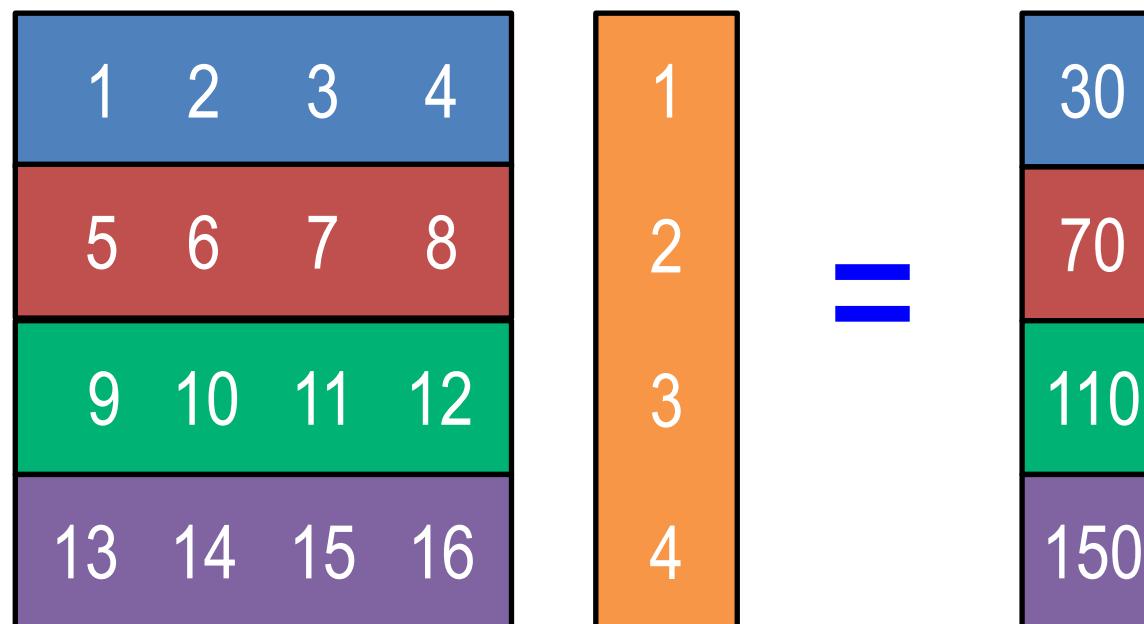
Matrix Vector Multiplication (Row Wise)

$$A \times B = X$$



Matrix Vector Multiplication (Row Wise)

- Scatter the rows of A to every process.
- Broadcast B to all the processes.
- Calculate each scalar using dot product of row A and B .
- Gather each scalar onto the root.



HW: Matrix Vector Multiplication (Row Wise)

```
#include <iostream>
#include <cstdlib>      // has exit(), etc.
#include <ctime>
#include <mpi.h>
#define MASTER 0

using namespace std;

int main(int argc, char **argv)
{
    // Just simply set N(4) X M(4) matrix and M(4) X 1 vector
    int N=4, M=4;
    int A[N][M], Apart[M], B[M], X[N];
    int Xpart = 0, root=0;
    int nproc, rank;

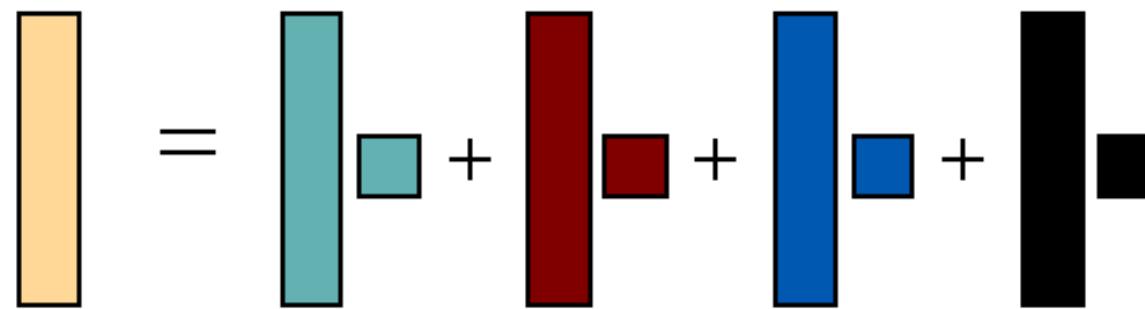
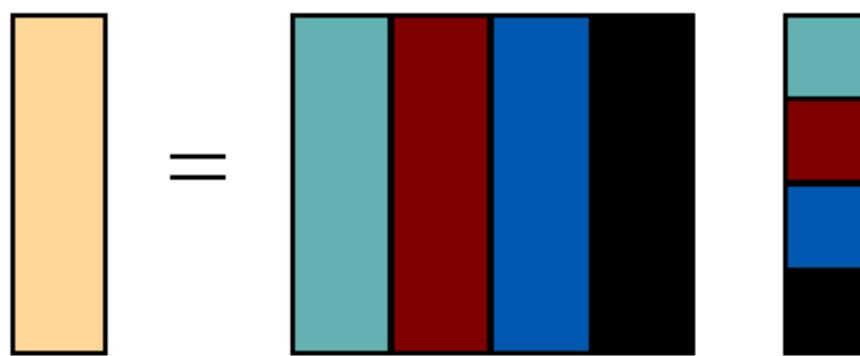
    // MPI init
    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &nprocs);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
```

```
// Initialize matrix
if (rank == MASTER) {
    ...
}

// MPI Scatter the A Matrix
...
// Broadcast the B vector
...
// Calculate Xpart
...
// MPI Gather
...
// Print results
if (rank == MASTER) {
    for(int i=0; i<N; i++) {
        cout << "X[" << i << "]=" << X[i] << endl;
    }
}

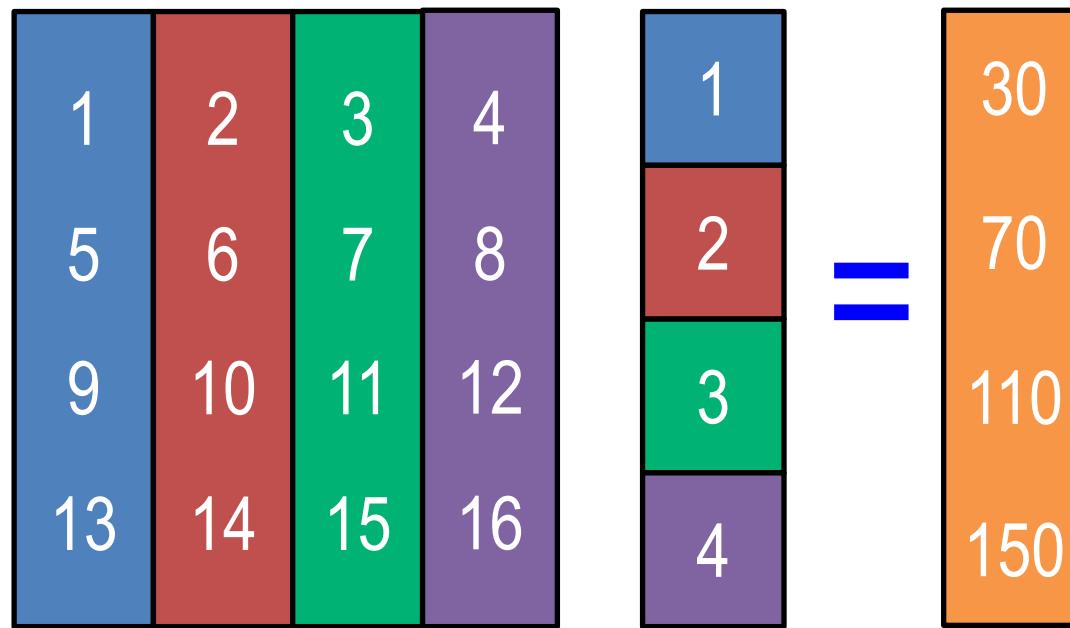
MPI_Finalize(); // MPI finalize
return 0; // Exit
}
```

Matrix-vector Multiplication (Column wise)

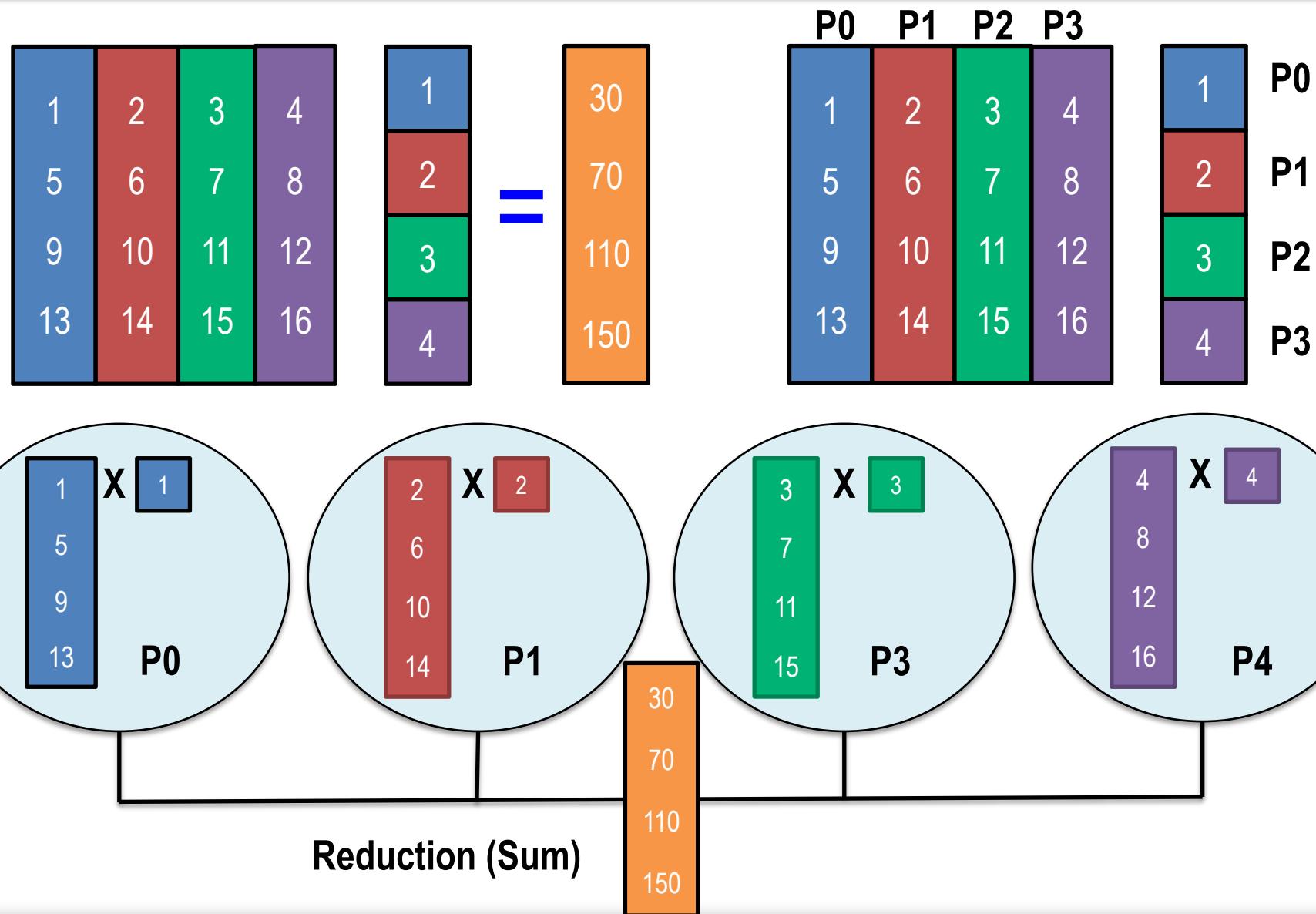


Schematic of parallel decomposition for vector-matrix multiplication, $\mathbf{A}=\mathbf{B}^*\mathbf{C}$. The vector **A** is depicted in yellow. The matrix **B** and vector **C** are depicted in multiple colors representing the portions, columns, and elements assigned to each processor, respectively.

Matrix Vector Multiplication (Column Wise)

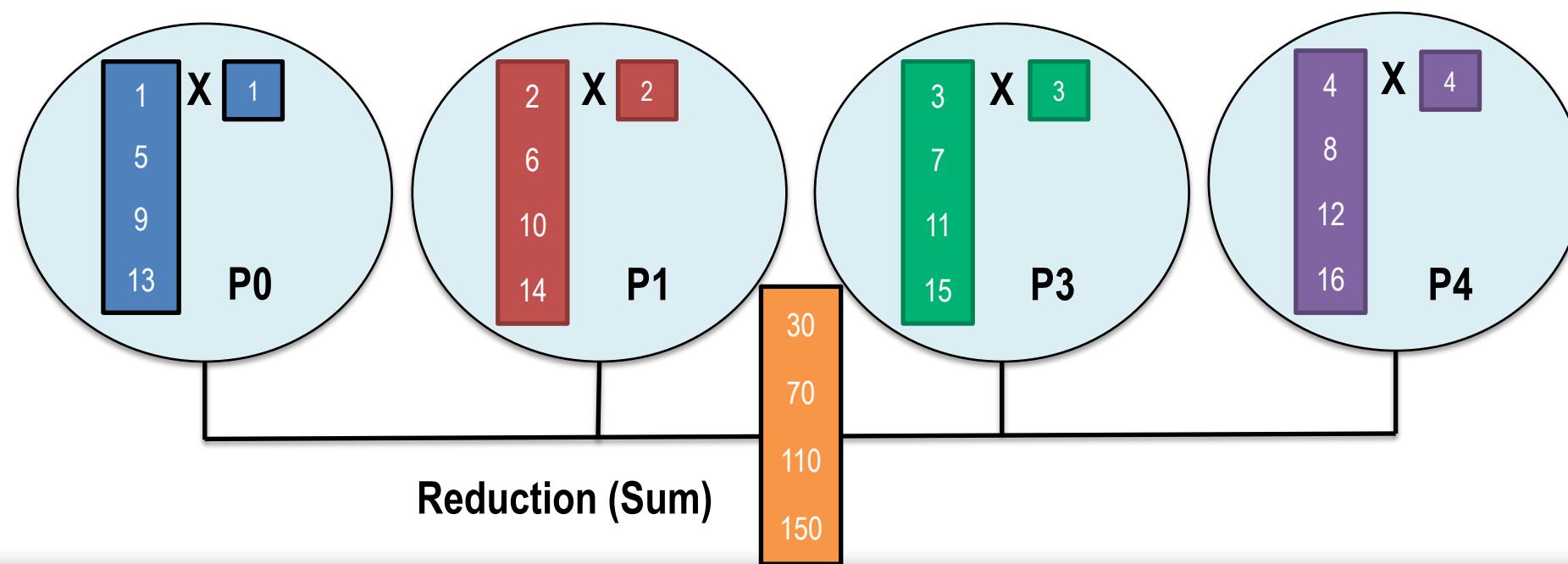


Matrix Vector Multiplication (Column Wise)



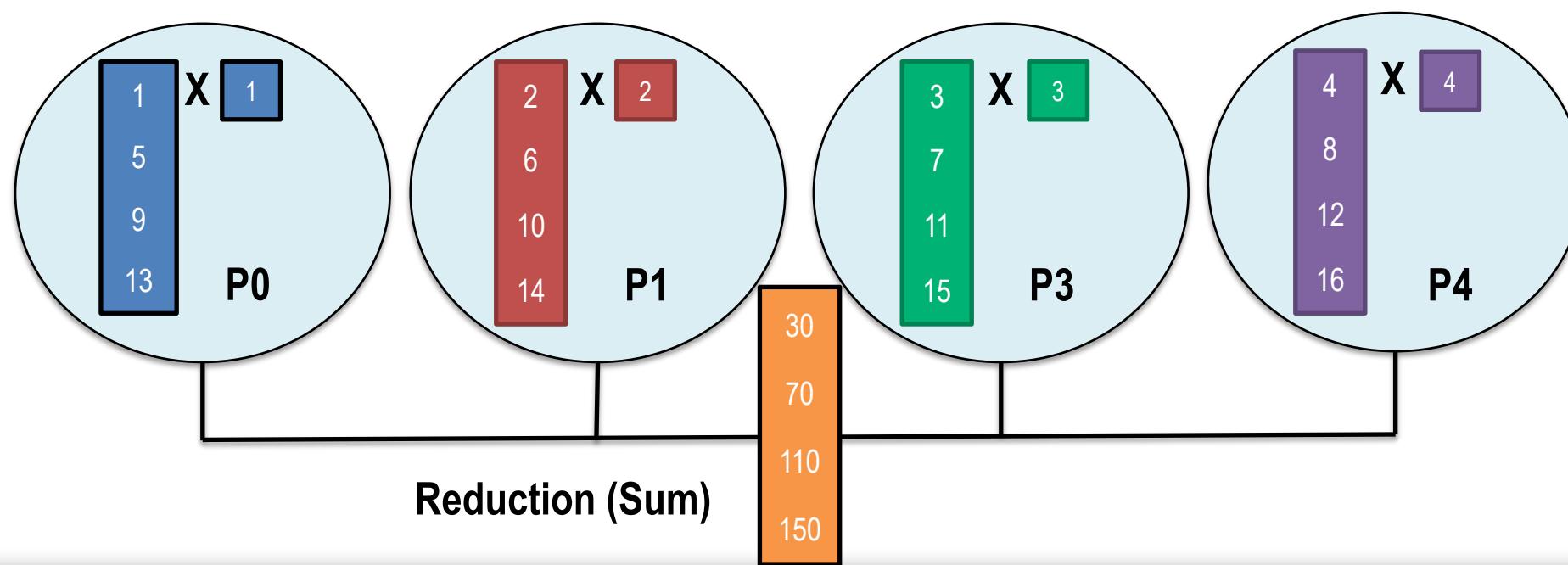
Matrix Vector Multiplication (Column Wise)

- Scatter the columns of A to every process.
- Scatter B to every process.
- Perform independent vector-scalar multiplication of row A and B element.
- Use “add” reduction operation to sum all vectors.



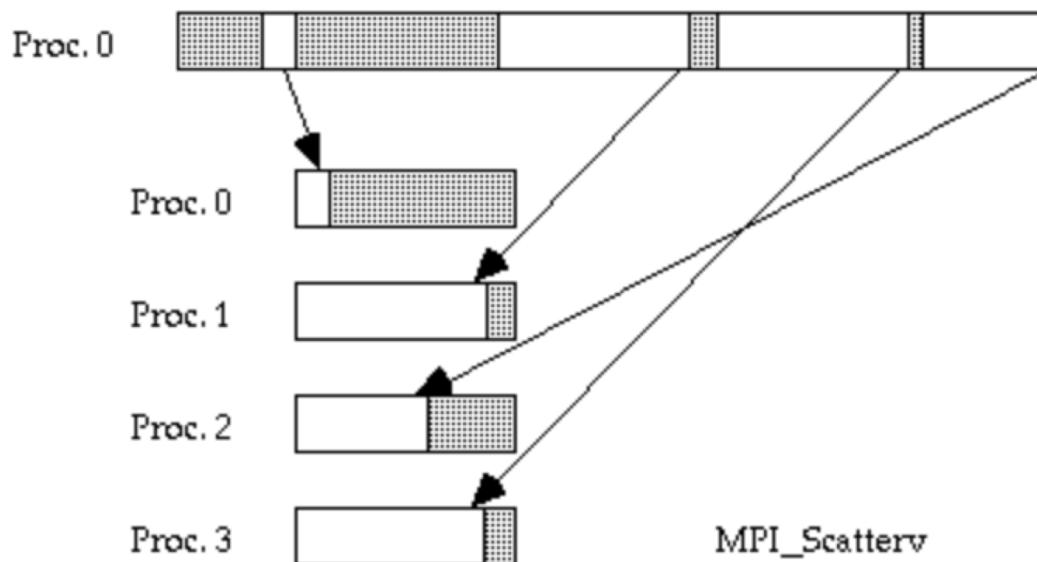
Matrix Vector Multiplication (Column Wise)

- Scatter the columns of A to every process. ← Can we do it?
- Scatter B to every process.
- Perform independent vector-scalar multiplication of row A and B element.
- Use “add” reduction operation to sum all vectors.



Scatterv

- Gaps are allowed between messages in source data
(but the individual messages must still be contiguous)
- Irregular message sizes are allowed
- Data can be distributed to processes in any order



<https://cvw.cac.cornell.edu/mpicc/scatterv>

MPI_Scatterv

Synopsis

```
int MPI_Scatterv(const void *sendbuf, const int *sendcounts, const int *displs,
                  MPI_Datatype sendtype, void *recvbuf, int recvcount,
                  MPI_Datatype recvtype,
                  int root, MPI_Comm comm)
```

Input Parameters

sendbuf

address of send buffer (choice, significant only at **root**)

sendcounts

integer array (of length group size) specifying the number of elements to send to each processor

displs

integer array (of length group size). Entry *i* specifies the displacement (relative to **sendbuf** from which to take the outgoing data to process *i*)

sendtype

data type of send buffer elements (handle)

recvcount

number of elements in receive buffer (integer)

recvtype

data type of receive buffer elements (handle)

root

rank of sending process (integer)

comm

communicator (handle)

Output Parameters

recvbuf

address of receive buffer (choice)

Lab: Scatterv example (mpi_collective_scatterv.cpp)

- Data has

```
char data[SIZE][SIZE] = {  
    {'a', 'b', 'c', 'd'},  
    {'e', 'f', 'g', 'h'},  
    {'i', 'j', 'k', 'l'},  
    {'m', 'n', 'o', 'p'}
```

- Set sendcounts and displacements as

```
sendcounts[0] = 1    displs[0] = 0  
sendcounts[1] = 2    displs[1] = 4  
sendcounts[2] = 3    displs[2] = 8  
sendcounts[3] = 4    displs[3] = 12
```

- After scatterv, it will scatter and print out as

```
0: a  
1: e f  
2: i j k  
3: mn o p
```

Lab: Scatterv example (mpi_collective_scatterv.cpp)

```
# include <iostream>
# include <cstdlib> // has exit(), etc.
# include <ctime>
# include <stdio.h>
# include "mpi.h"    // MPI header file

#define SIZE 4
using namespace std;

int main (int argc, char **argv)
{
    int nprocs, rank;

    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &nprocs);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);

    // the data to be distributed
    char data[SIZE][SIZE] = {
        {'a', 'b', 'c', 'd'},
        {'e', 'f', 'g', 'h'},
        {'i', 'j', 'k', 'l'},
        {'m', 'n', 'o', 'p'}
    };

    char rec_buf[100];
    int sendcounts[nprocs];
    int displs[nprocs];

    // calculate send counts and displacements
    // Your code

    // print calculated send counts and displacements for each process
    if (rank == 0) {
        for (int i = 0; i < nprocs; i++) {
            printf("sendcounts[%d] = %d\tdispls[%d] = %d\n", i, sendcounts[i], i, displs[i]);
        }
    }

    // MPI Scatterv
    // Your code

    printf("%d: ", rank);
    for (int i = 0; i < sendcounts[rank]; i++) {
        printf("%c\t", rec_buf[i]);
    }
    printf("\n");

    MPI_Finalize();
    return 0;
}
```