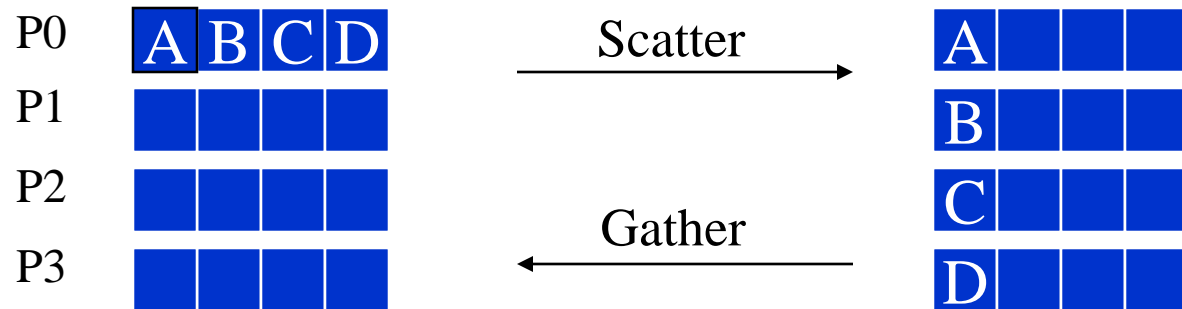
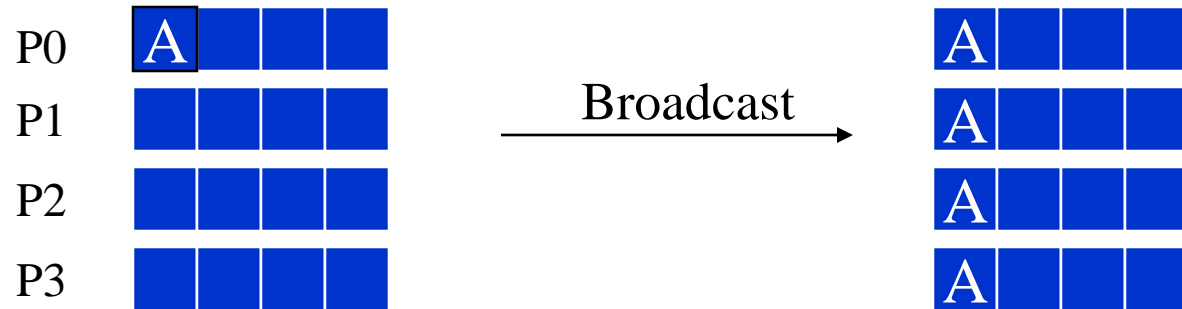
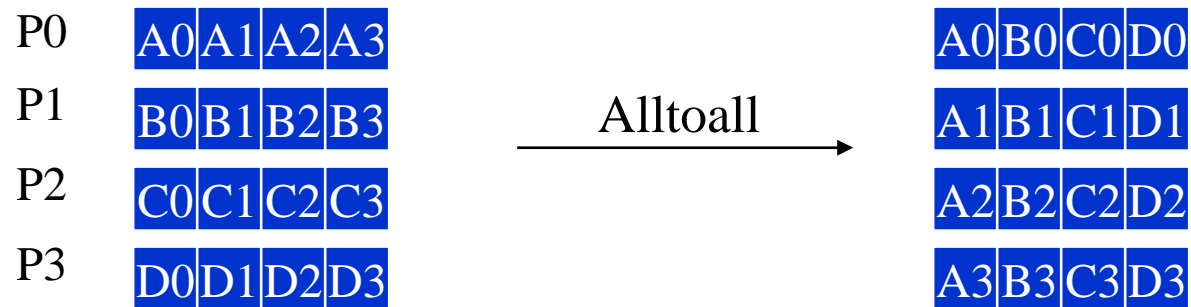
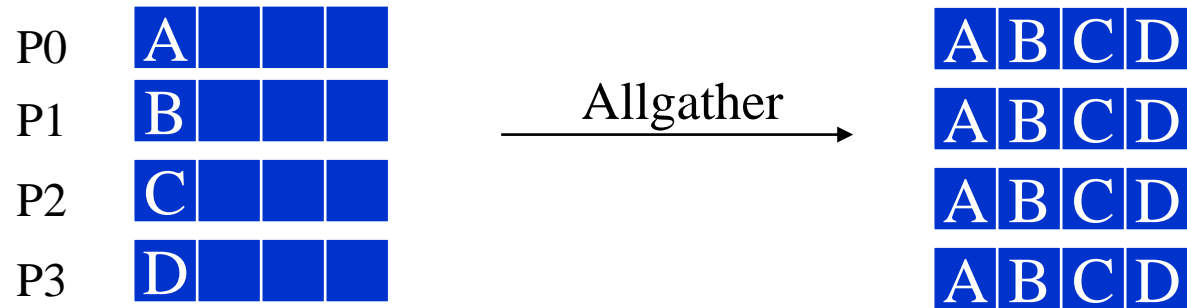


# Exercises 6: MPI Collective Communication

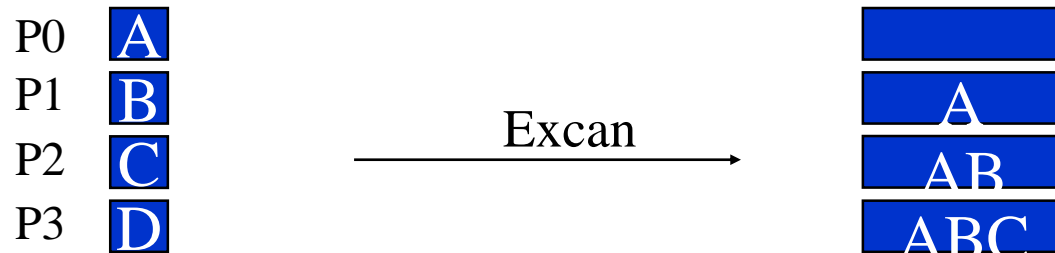
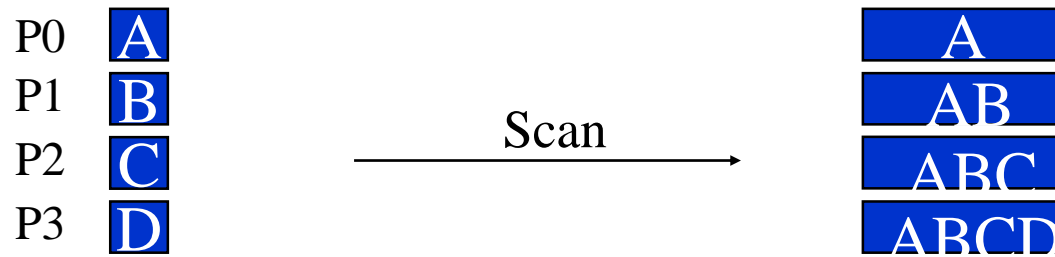
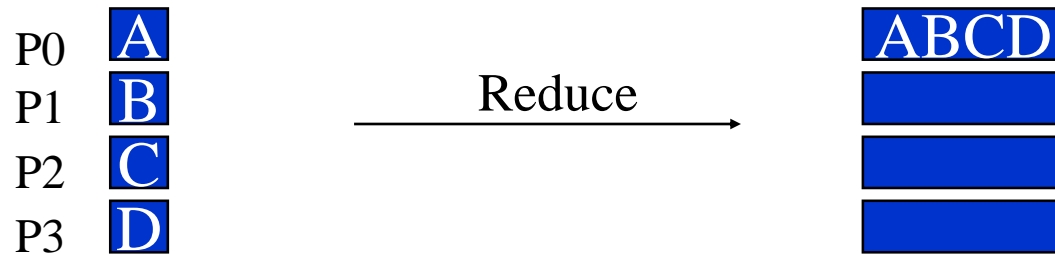
# Collective Data Movement



# More Collective Data Movement

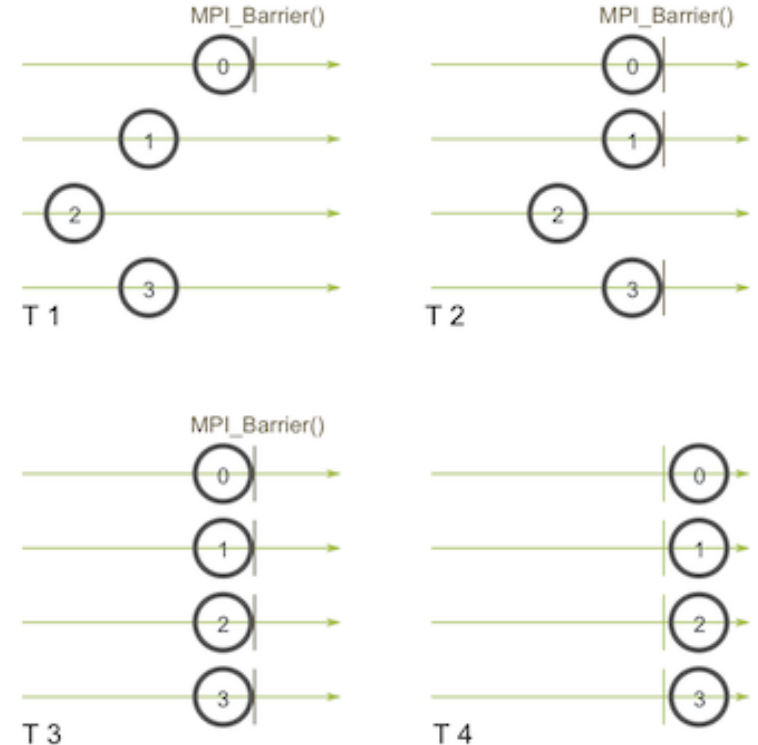


# Collective Computation



# Collective communication and synchronization points

- collective communication implies a *synchronization point* among processes
  - all processes must reach a point in their code before they can all begin executing again
- MPI has a special function that is dedicated to synchronizing processes:  
`MPI_Barrier(MPI_Comm communicator)`
- function forms a barrier, and no processes in the communicator can pass the barrier until all of them call the function



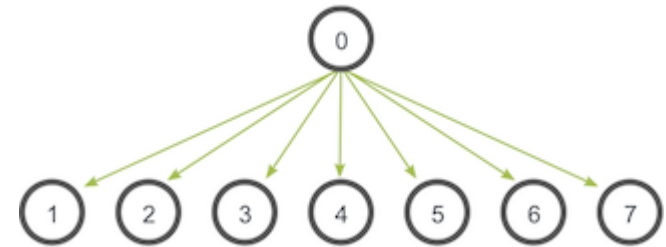
# MPI\_Barrier Implementation

- How would you implement an MPI\_Barrier?
- One way: recall the ring program from last time
  - Process 0 sends to 1, 1 sends to 2, ... $p-1$  sends back to 0
  - This type of program is one of the simplest methods to implement a barrier since a token can't be passed around completely until all processes work together.

# Broadcasting with MPI\_Bcast

- One process sends the same data to all processes in a communicator.
- Main uses of broadcasting: send out user input to a parallel program, or send out configuration parameters to all processes

```
MPI_Bcast(  
    void* data,  
    int count,  
    MPI_Datatype datatype,  
    int root,  
    MPI_Comm communicator)
```



In this example, process zero is the *root* process, and it has the initial copy of data. All of the other processes receive the copy of data.

- Although the root process and receiver processes do different jobs, they all call the same MPI\_Bcast function.
- When the root process calls MPI\_Bcast, the data variable will be sent to all other processes.
- When all of the receiver processes call MPI\_Bcast, the data variable will be filled in with the data from the root process.

# Task 1: Broadcast with MPI\_Send and MPI\_Recv

- We can implement the functionality of MPI\_Bcast just using MPI\_Send and MPI\_Recv.
  - Root process sends the data to everyone else while the others receive from the root process
- We can write such a wrapper function right now.
  - Look at my\_bcast.c
  - Fill in the appropriate MPI\_Send and MPI\_Recv calls
- Reminders:
  - Type “module load openmpi” to load the MPI compiler
  - To compile, use “mpicc” instead of gcc
  - To run, use (e.g.) “mpirun -n 4 ./my\_program”



# Solution

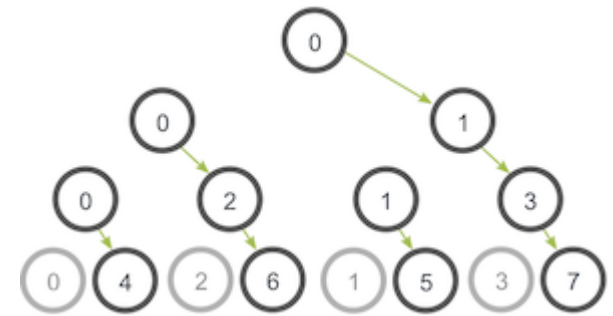
# my\_bcast.c

```
void my_bcast(void* data, int count, MPI_Datatype datatype, int root, MPI_Comm communicator) {
    int world_rank, world_size;
    MPI_Comm_rank(communicator, &world_rank);
    MPI_Comm_size(communicator, &world_size);

    if (world_rank == root) {
        // If we are the root process, send our data to everyone
        int i;
        for (i = 0; i < world_size; i++) {
            if (i != world_rank) {
                MPI_Send(data, count, datatype, i, 0, communicator);
            }
        }
    } else {
        // If we are a receiver process, receive the data from the root
        MPI_Recv(data, count, datatype, root, 0, communicator, MPI_STATUS_IGNORE);
    }
}
```

# Efficiency

- our function is actually very inefficient
  - Imagine that each process has only one outgoing/incoming network link.
  - Our function is only using one network link from process zero to send all the data.
  - A smarter implementation is a tree-based communication algorithm that can use more of the available network links at once.
- 
- Ex: Process zero starts off with the data and sends it to process one.
  - Similar to our previous example, process zero also sends the data to process two in the second stage.
  - The difference with this example is that process one is now helping out the root process by forwarding the data to process three.
    - During the second stage, two network connections are being utilized at a time. The network utilization doubles at every subsequent stage of the tree communication until all processes have received the data.



# Comparing MPI\_Bcast with Send/Recv version

- MPI\_Bcast implementation utilizes such a tree broadcast algorithm for good network utilization
- We will run `compare_bcast.c` to compare timing
- Note on timing in MPI:
  - MPI\_Wtime takes no arguments, and it simply returns a floating-point number of seconds since a set time in the past.
  - Similar to C's time function, you can call multiple MPI\_Wtime functions throughout your program and subtract their differences to obtain timing of code segments.

```
for (i = 0; i < num_trials; i++) {  
  
    // Time my_bcast  
    // Synchronize before starting timing  
    MPI_Barrier(MPI_COMM_WORLD);  
    total_my_bcast_time -= MPI_Wtime();  
    my_bcast(data, num_elements, MPI_INT, 0, MPI_COMM_WORLD);  
  
    // Synchronize again before obtaining final time  
    MPI_Barrier(MPI_COMM_WORLD);  
    total_my_bcast_time += MPI_Wtime();  
  
    // Time MPI_Bcast  
    MPI_Barrier(MPI_COMM_WORLD);  
    total_mpi_bcast_time -= MPI_Wtime();  
    MPI_Bcast(data, num_elements, MPI_INT, 0, MPI_COMM_WORLD);  
    MPI_Barrier(MPI_COMM_WORLD);  
    total_mpi_bcast_time += MPI_Wtime();  
}
```

# Task 2: Comparing Bcasts

Start an interactive session with n cores:

```
srun -n 8 -p express3 --pty /bin/bash -i
```

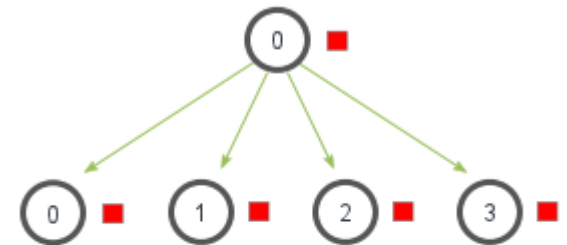
- Run `compare_bcast` with the following parameters:
  - `num_elements = 500`, `num_trials = 1000`
  - `num_elements = 500000`, `num_trials = 1000`
- For each set of parameters, try with `-n 2` and `-n 8` processes (can be on the same node)
  - So you should have a total of 4 runs

# MPI\_Scatter

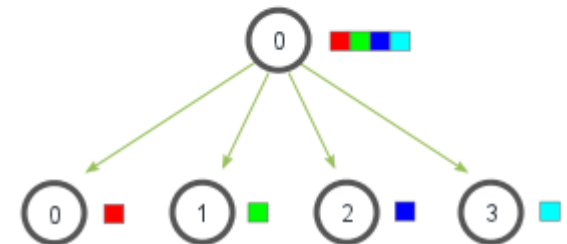
- similar to MPI\_Bcast
  - MPI\_Scatter involves a designated root process sending data to all processes in a communicator.
- The primary difference between MPI\_Bcast and MPI\_Scatter:  
**MPI\_Bcast sends the same piece of data to all processes while MPI\_Scatter sends chunks of an array to different processes.**

```
MPI_Scatter(  
    void* send_data,  
    int send_count,  
    MPI_Datatype send_datatype,  
    void* recv_data,  
    int recv_count,  
    MPI_Datatype recv_datatype,  
    int root,  
    MPI_Comm communicator)
```

MPI\_Bcast



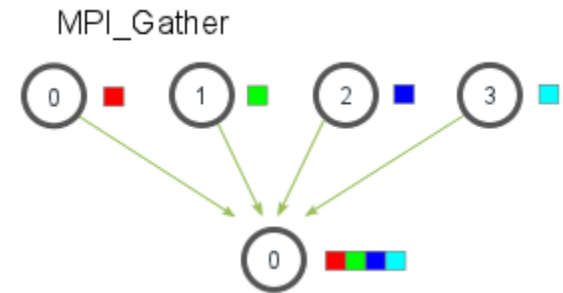
MPI\_Scatter



# MPI\_Gather

- MPI\_Gather is the inverse of MPI\_Scatter.
- Instead of spreading elements from one process to many processes, MPI\_Gather takes elements from many processes and gathers them to one single process.
- Highly useful to many parallel algorithms, such as parallel sorting and searching.
- 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.

```
MPI_Gather(  
    void* send_data,  
    int send_count,  
    MPI_Datatype send_datatype,  
    void* recv_data,  
    int recv_count,  
    MPI_Datatype recv_datatype,  
    int root,  
    MPI_Comm communicator)
```





# Task 3: Averaging Numbers with Collectives

- Example program `avg.c` meant to compute the average across all numbers in an array
- demonstrates how one can use MPI to divide work across processes, perform computation on subsets of data, and then aggregate the smaller pieces into the final answer
- Does the following:
  1. Generate a random array of numbers on the root process (process 0).
  2. Scatter the numbers to all processes, giving each process an equal amount of numbers.
  3. Each process computes the average of their subset of the numbers.
  4. Gather all averages to the root process. The root process then computes the average of these numbers to get the final average.

**Task:** Add the appropriate `MPI_Scatter` and `MPI_Gather` calls to the program

# Solution

```
// Scatter the random numbers to all processes
MPI_Scatter(rand_nums, num_elements_per_proc, MPI_FLOAT, sub_rand_nums,
            num_elements_per_proc, MPI_FLOAT, 0, MPI_COMM_WORLD);

// Compute the average of your subset
float sub_avg = compute_avg(sub_rand_nums, elements_per_proc);

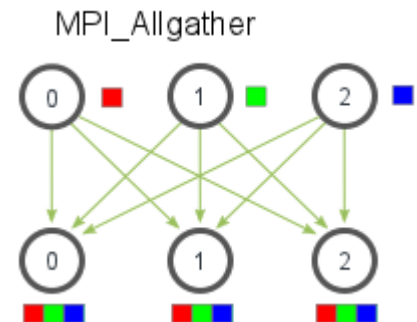
// Gather all partial averages down to the root process
float *sub_avgs = NULL;
if (world_rank == 0) { sub_avgs = malloc(sizeof(float) * world_size);}

MPI_Gather(&sub_avg, 1, MPI_FLOAT, sub_avgs, 1, MPI_FLOAT, 0, MPI_COMM_WORLD);
```

# Allgather: many to many collectives

- So far, we have covered two MPI routines that perform many-to-one or one-to-many communication patterns
- Often it is useful to be able to send many elements to many processes (i.e. a many-to-many communication pattern)
- Given a set of elements distributed across all processes, MPI\_Allgather will gather all of the elements to all the processes.
- In the most basic sense, MPI\_Allgather is an MPI\_Gather followed by an MPI\_Bcast.

```
MPI_Allgather(  
    void* send_data,  
    int send_count,  
    MPI_Datatype send_datatype,  
    void* recv_data,  
    int recv_count,  
    MPI_Datatype recv_datatype,  
    MPI_Comm communicator)
```



# Reductions

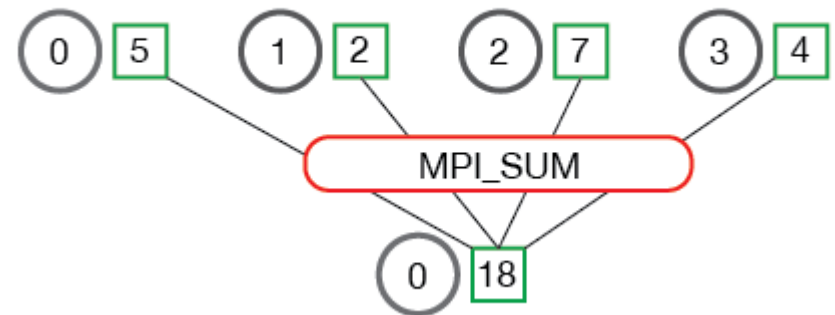
- "Reduce" is a classic concept from functional programming.
- Data reduction involves reducing a set of numbers into a smaller set of numbers via a function.
- For example, let's say we have a list of numbers [1, 2, 3, 4, 5].
  - Reducing this list of numbers with the sum function would produce  $\text{sum}([1, 2, 3, 4, 5]) = 15$ .
  - Similarly, the multiplication reduction would yield  $\text{multiply}([1, 2, 3, 4, 5]) = 120$ .

# MPI\_Reduce

- Similar to MPI\_Gather, MPI\_Reduce takes an array of input elements on each process and returns an array of output elements to the root process.
- The output elements contain the reduced result

```
MPI_Reduce(  
    void* send_data,  
    void* recv_data,  
    int count,  
    MPI_Datatype datatype,  
    MPI_Op op,  
    int root,  
    MPI_Comm communicator)
```

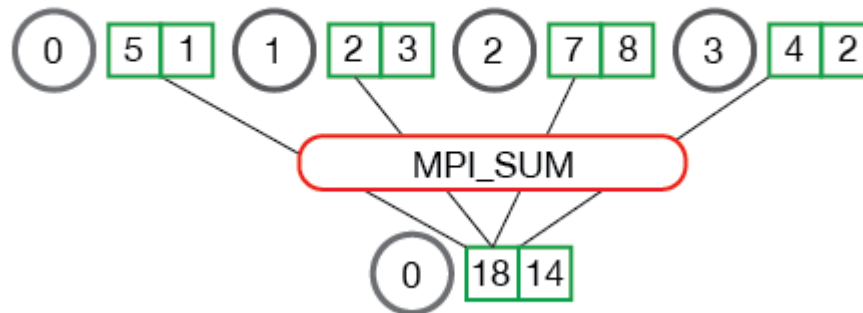
MPI\_Reduce



# Reduction on Arrays

- Ex: what if processes each have 2 elements?
- The resulting reduction happens on a per-element basis
  - $i^{\text{th}}$  element from each array are summed into the  $i^{\text{th}}$  element in result array of process 0.

MPI\_Reduce



# Included Reduction Operations

- MPI contains a set of common reduction operations that can be used.
  - 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.
- Custom reduction operations can also be defined



# Task 4: Computing Average with MPI\_Reduce

- Before we computed averages using MPI\_Scatter and MPI\_Gather
- Using MPI\_Reduce simplifies the code
- See reduce\_avg.c
- Add the appropriate MPI\_Reduce call

# Solution

```
// Sum the numbers locally
float local_sum = 0;
int i;
for (i = 0; i < num_elements_per_proc; i++) {
    local_sum += rand_nums[i];
}

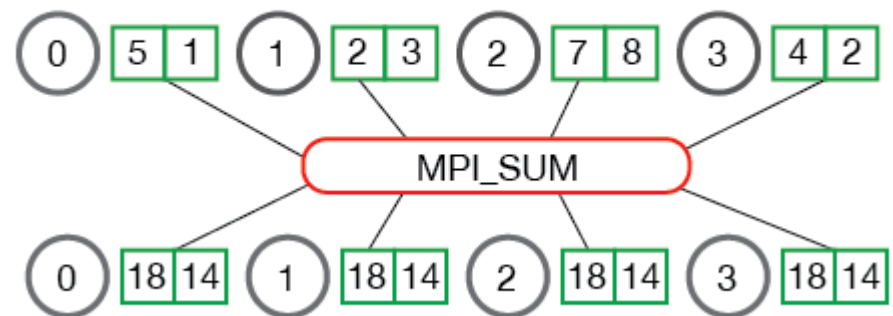
// Reduce all of the local sums into the global sum
float global_sum;
MPI_Reduce(&local_sum, &global_sum, 1, MPI_FLOAT, MPI_SUM, 0,
          MPI_COMM_WORLD);
```

# 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 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).

```
MPI_Allreduce(  
    void* send_data,  
    void* recv_data,  
    int count,  
    MPI_Datatype datatype,  
    MPI_Op op,  
    MPI_Comm communicator)
```

MPI\_Allreduce



$$\text{MPI\_Allreduce} = \text{MPI\_Reduce} + \text{MPI\_Bcast}$$

# Nonblocking Collectives

- Recall from lecture: Since MPI version 3, nonblocking collectives are included
  - Function returns immediately; doesn't wait for collective communication to finish
  - Can allow overlapping of communication with other useful work
- Nonblocking call has the same syntax as its blocking counterpart, with two differences
  - The letter “I” appears in the name of the call, immediately following the first underscore: e.g., `MPI_Ibcast`.
  - Final argument is a *handle* to a *request object* that holds detailed information about the transaction. The *request handle* can be used for subsequent `Wait` and `Test` calls.
- How to check that communication is finished before moving on?
  - `MPI_Wait(&request, &status);`
  - `MPI_Test(&request, &flag, &status);`

# Syntax Example

```
int MPI_Ibcast(    void *buffer,  
                  int count,  
                  MPI_Datatype datatype,  
                  int root,  
                  MPI_Comm comm,  
                  MPI_Request *request)
```

```
int MPI_Wait(      MPI_Request *request,  
                  MPI_Status *status)
```

```
int MPI_Test(      MPI_Request *request,  
                  int *flag,  
                  MPI_Status *status)
```

# Task 5: Find the Error

---

- The file `bcastnonblocking.c` attempts to implement a HelloWorld program using a nonblocking broadcast call.
  - Try to compile and run this program (use, e.g., 4 processes, and try running multiple times)
  - There is an error!
- 
- Task: Find the error and fix it!

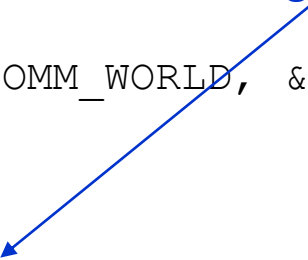
# Solution



# Nonblocking Bcast

```
carson@r0d0:[~/workingfiles/ex5]: srun -n 4  
./bcastnonblocking  
Message from process 1 : @  
@  
Message from process 3 : @  
@  
Message from process 2 : @  
@  
Message from process 0 : What will happen?
```

We don't check to make  
sure bcast is finished  
before overwriting  
message! Causes error!



```
//Call nonblocking broadcast  
MPI_Ibcast(message, 13, MPI_CHAR, root, MPI_COMM_WORLD, &request);  
if (rank == root)  
{  
    strcpy(message, "What will happen?");  
}  
printf( "Message from process %d : %s\n", rank, message);
```

# Nonblocking Bcast

- Need to call MPI\_Wait to make sure the communication is complete before we overwrite the buffer!
- See sols/bcastnonblocking\_sol.c

```
//Nonblocking broadcast
MPI_Ibcast(message, 13, MPI_CHAR, root, MPI_COMM_WORLD, &request);

//Call MPI_Wait to wait for broadcast to complete
MPI_Wait (&request, &status);

if (rank == root)
{
    strcpy(message, "What will happen?");
}
```