CS3006 Parallel and Distributed Computing

FALL 2022

NATIONAL UNIVERSITY OF COMPUTER AND EMERGING SCIENCES

Chapter 6. Programming Using the Message Passing Paradigm

Recall…

- A message-passing platform consists of ***p*** processing nodes, each with its own exclusive address space.
  - Interactions between processes running on different nodes must be accomplished using **messages** (data, work, and to synchronize actions among the processes), hence the name ***message passing***.
  - The basic operations in this programming paradigm are ***send*** and ***receive.***
  - The ***message-passing programming paradigm*** is one of the oldest and most widely used approaches for programming parallel computers.

Principles of Message-Passing Programming

- The two key attributes to characterize the message-passing programming paradigm.

1. The first is that it assumes a **partitioned address space**
2. Second is that it supports only **explicit parallelization**.

- There are two immediate implications of a partitioned address space.

1. First, each data element must belong to one of the partitions of the space; hence, **data must be explicitly partitioned and placed**.
2. The second implication is that all interactions (read-only or read/write) require cooperation of two processes - the process that has the data and the process that wants to access the data

Principles of Message-Passing Programming

- Message-passing programs are often written using the ***asynchronous*** or ***loosely synchronous*** paradigms
  - In the *asynchronous paradigm*, all concurrent tasks execute asynchronously.
  - However, such programs can be harder to reason about, and can have nondeterministic behavior due to race conditions .
  - In the *loosely synchronous model*, tasks or subsets of tasks synchronize to perform **interactions**. Between these interactions, tasks execute completely asynchronously.
- Most message-passing programs are written using the ***single program multiple data*** (SPMD) model.

The Building Blocks: **Send** and **Receive** Operations

- In their simplest form, the prototypes of these operations are defined as follows:

```
send(void *sendbuf, int nelems, int dest)

receive(void *recvbuf, int nelems, int source)
```

```
P0                      P1

a = 100;                receive(&a, 1, 0)
send(&a, 1, 1);         printf("%d\n", a);
a=0;
```

Modes of Communication

- Point-to-point communication
  - **Blocking** – returns from call when task completes
  - **Nonblocking** – returns from call without waiting for task to complete
- Collective communication

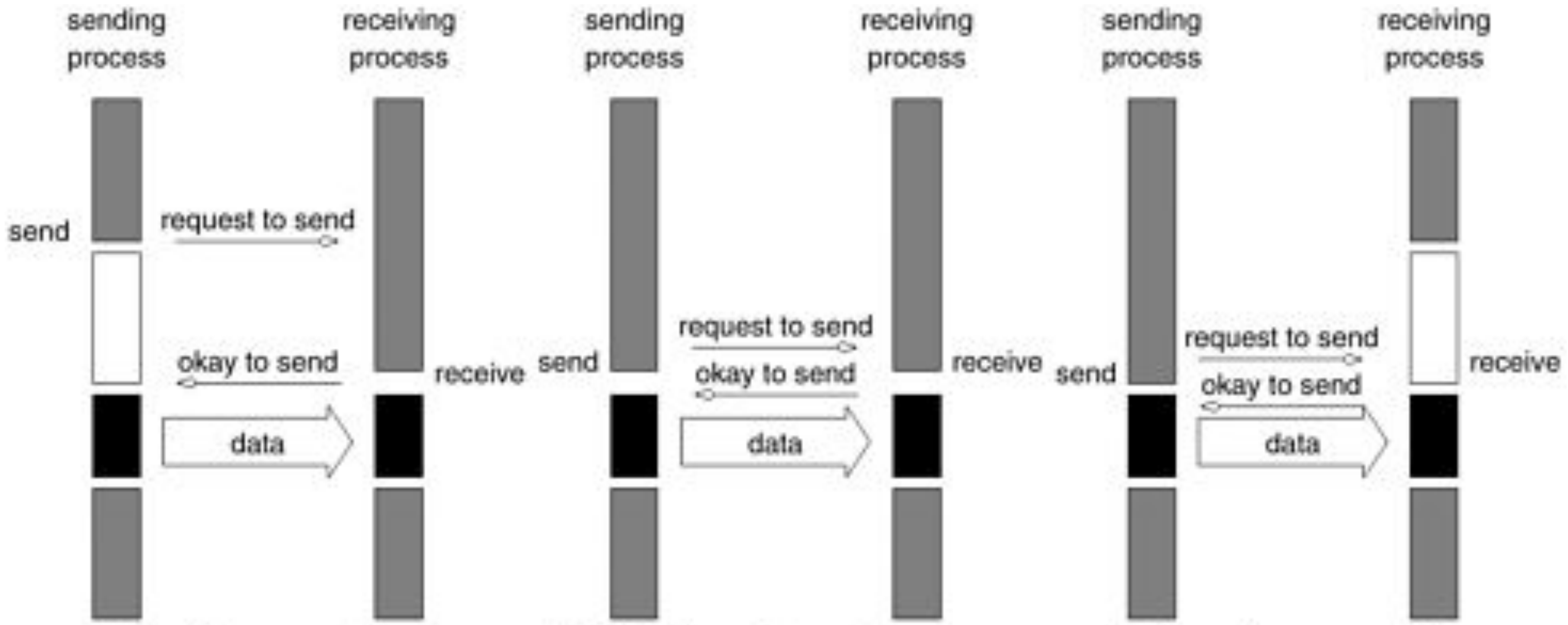
- The semantics of the send operation require that the value received by process **P1** must be **100** as opposed to **0**.

- This motivates the design of the **send** and **receive protocols**.

Blocking Message Passing Operations

- A simple method for forcing send/receive semantics is for the send operation to return only when it is safe to do so.

- In the **non-buffered blocking send**, the operation does not return until the matching receive has been encountered at the receiving process.
  - **Idling** and **deadlocks** are major issues with non-buffered blocking sends.



(a) Sender comes first; idling at sender

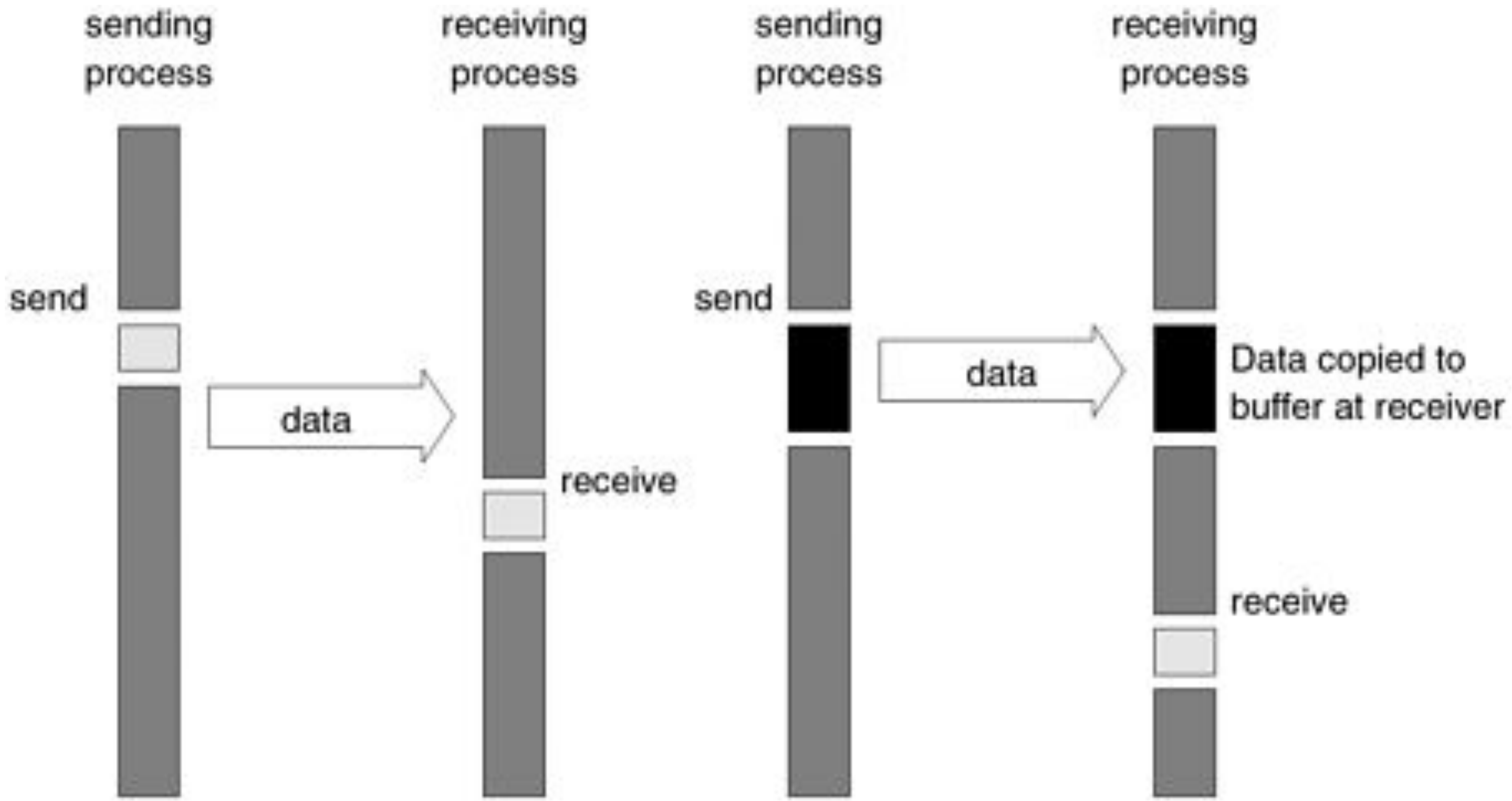
(b) Sender and receiver come at about the same time; idling minimized

(c) Receiver comes first; idling at receiver

```
P0                                    P1

send(&a, 1, 1);                       send(&a, 1, 0);
receive(&b, 1, 1);                    receive(&b, 1, 0);
```

- In **buffered blocking sends**, the sender simply copies the data into the designated buffer and returns after the copy operation has been completed.
- The data is copied at a buffer at the receiving end as well.
  - Buffering alleviates idling at the expense of **copying overheads**.

Blocking buffered transfer protocols: (**a**) in the presence of communication hardware with buffers at send and receive ends; and (**b**) in the absence of communication hardware, sender interrupts receiver and deposits data in buffer at receiver end.



```
        P0                                      P1

        for (i = 0; i < 1000; i++) {            for (i = 0; i < 1000; i++) {
            produce_data(&a);                       receive(&a, 1, 0);
            send(&a, 1, 1);                         consume_data(&a);
        }                                       }
```

- What if consumer was much slower than producer?
  - This can often lead to unforeseen overheads and performance degradation.

- Deadlocks are still possible with buffering since receive operations block.

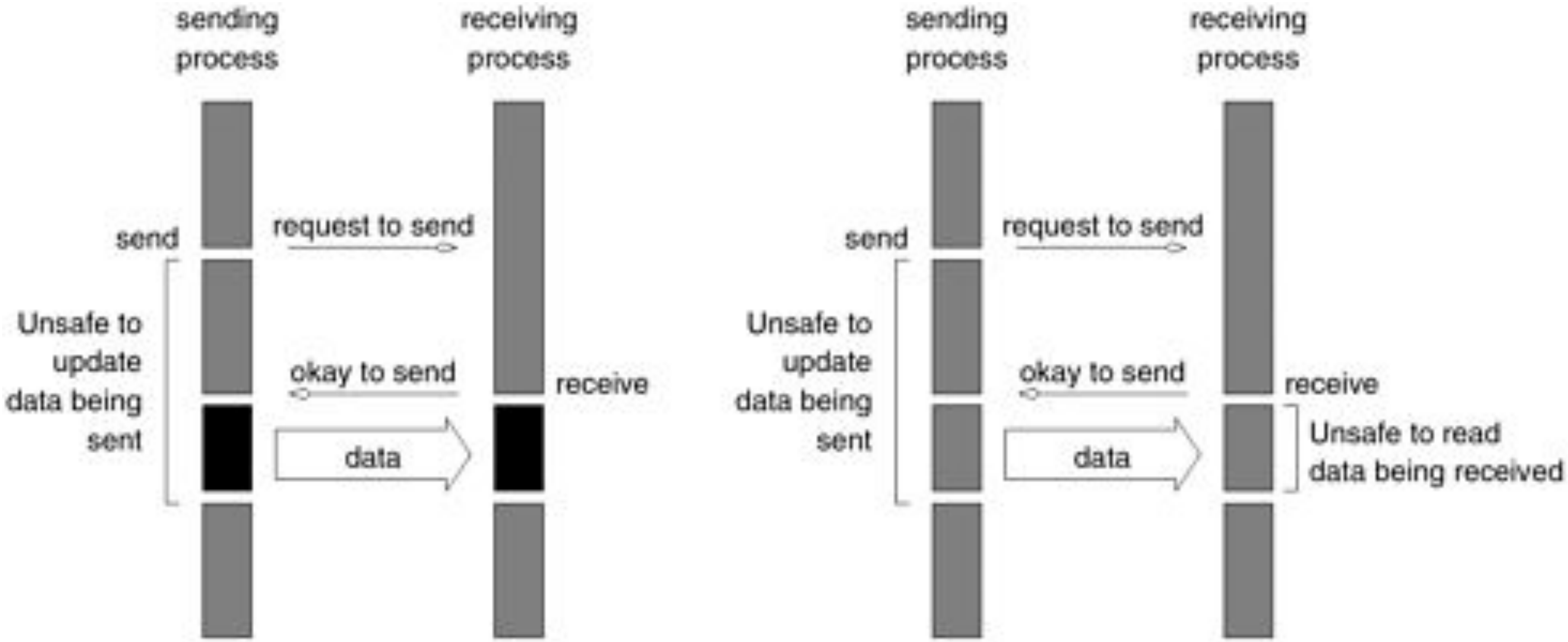
```
P0                          P1

receive(&a, 1, 1);          receive(&a, 1, 0);
send(&b, 1, 1);             send(&b, 1, 0);
```

Non-Blocking Message Passing Operations

- This class of non-blocking protocols returns from the send or receive operation before it is semantically safe to do so.

- Non-blocking operations are generally accompanied by a **check-status** operation.
    - When used correctly, these primitives are capable of overlapping communication overheads with useful computations.

- Message passing libraries typically provide both blocking and non-blocking primitives.



(a) Without hardware support

(b) With hardware support

Collective Communication and Computation Operations

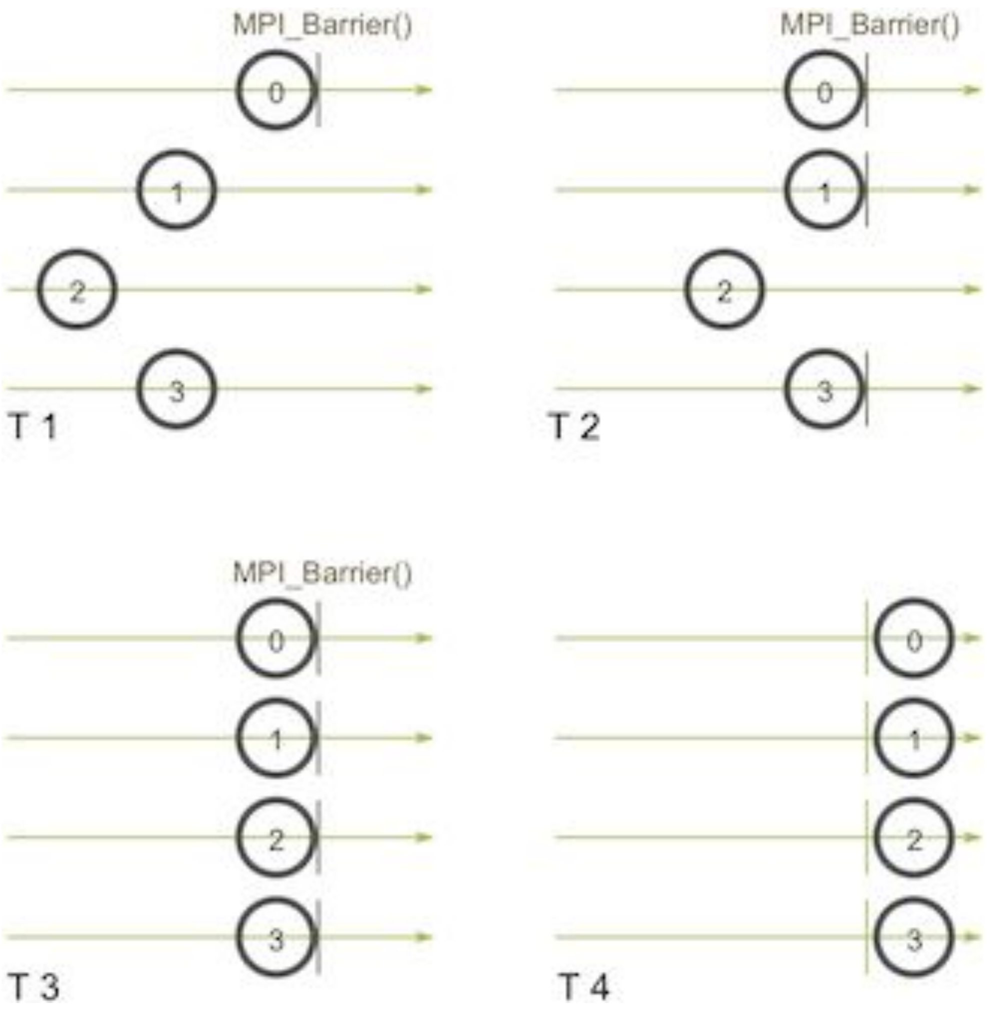
- All of the collective communication functions provided by MPI take a *communicator* as an argument that defines the group of processes that participate in the collective operation.

Collective Communication and Computation Operations : **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 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.


MPI_Barrier()
0
1
2
3
T 1
MPI_Barrier()
0
1
2
3
T 2
MPI_Barrier()
0
1
2
3
T 3
0
1
2
3
T 4

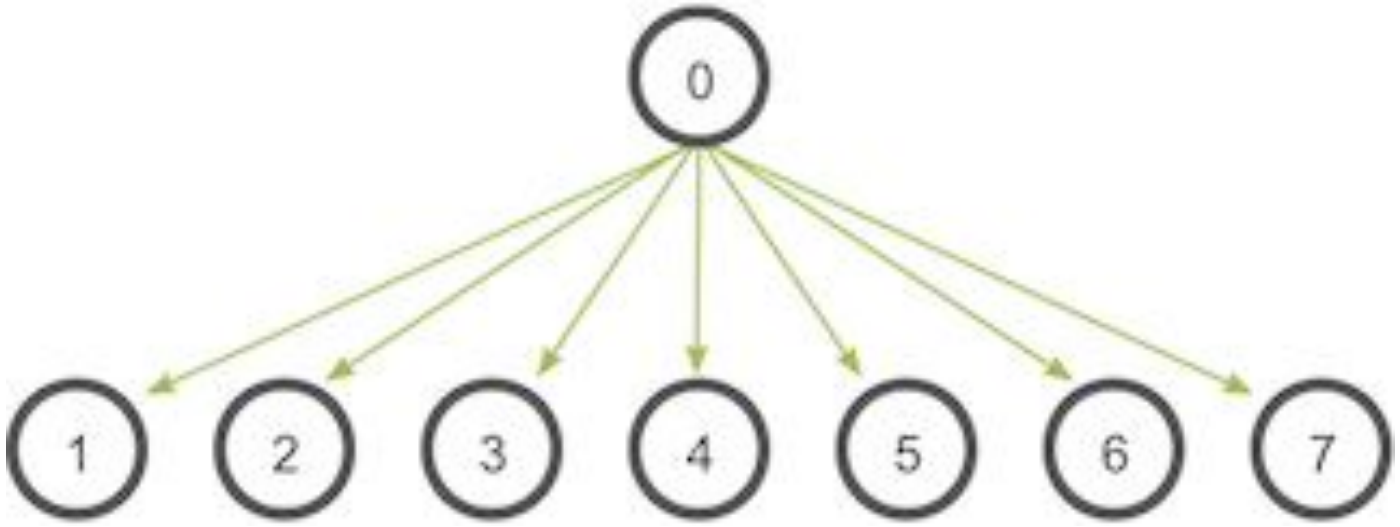
One-to-All Broadcast

- Parallel algorithms often require a single process to send identical data to all other processes or to a subset of them. This operation is known as ***one-to-all broadcast***.

- Initially, only the source process has the data of size ***m*** that needs to be broadcast.

- At the termination of the procedure, there are ***p*** copies of the initial data – one belonging to each process.

- 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.


0
1
2
3
4
5
6
7

One-to-All Broadcast

- The **one-to-all broadcast** operation 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` operation sends data from one member of one group to all members of the other group

- `MPI_Bcast` sends the data stored in the buffer `buf` of process source to all the other processes in the group

One-to-All Broadcast

- The data received by each process is stored in the buffer `buf`.

- The data that is broadcast consist of `count` entries of type `datatype`.

- Although the source process and receiver processes do different jobs, they all call the same **MPI_Bcast** function.
  - When the source process calls MPI_Bcast, the buffer data will be sent to all other processes.
  - When all of the receiver processes call MPI_Bcast, the buffer data will be filled in with the data from the root process.

All-to-One Reduction

- 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 **sum([1, 2, 3, 4, 5]) = 15**.
  - Similarly, the multiplication reduction would yield multiply([1, 2, 3, 4, 5]) = 120.

All-to-One Reduction

- In an **all-to-one** reduction operation, each of the *p* participating processes starts with a buffer *M* containing *m* words.

- The data from all processes are combined through an associative operator and accumulated at a **single** destination process into one buffer of size *m* .

- Reduction can be used to find the sum, product, maximum, or minimum of sets of numbers.

All-to-One Reduction

- The **all-to-one** reduction operations 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`

All-to-One Reduction

- 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_Op enumeration

- MPI provides a list of predefined operations that can be used to combine the elements stored in sendbuf .

- 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.
  - MPI_MAX, MPI_MIN, MPI_SUM, MPI_PROD, MPI_LAND, MPI_BAND, , MPI_LOR, , MPI_BOR, MPI_LXOR, MPI_BXOR, MPI_MAXLOC, MPI_MINLOC

Tasks:

1. Write a C program to do the following:
   1. On process 0, send a message "Hello, I am process 0" to other processes.
   2. On all other processes, print the process's ID, the message it receives and where the message came from.
2. Implement Broadcasting with MPI_send and MPI_receive
3. Calculate AVERAGE of N random numbers with MPI_Reduce

```c
float *rand_nums = NULL;
rand_nums =
create_rand_nums(num_elements_per_proc);

// Sum the numbers locally

float local_sum = 0;
int i;
for (i = 0; i < num_elements_per_proc; i++)
    { local_sum += rand_nums[i]; }

// Print the random numbers on each process
printf("Local sum for process %d - %f, avg = %f\n",
world_rank, local_sum, local_sum /
num_elements_per_proc);
```

```c
// 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);

// Print the result

if (world_rank == 0)
  { printf("Total sum = %f, avg = %f\n",
global_sum, global_sum / (world_size *
num_elements_per_proc)); }
```

```
 // 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);

// Print the result
if (world_rank == 0)
    { printf("Total sum = %f, avg = %f\n", global_sum, global_sum / (world_size * num_elements_per_proc));
}
```

Scatter and Gather

- In the ***scatter*** operation, a single node sends a unique message of size m to every other node.
  - This operation is also known as ***one-to-all personalized communication***.

- One-to-all personalized communication is different from one-to-all broadcast in that the source node starts with p unique messages, one destined for each node.

- Unlike one-to-all broadcast, one-to-all personalized communication does not involve any duplication of data.

Scatter and Gather

- The dual of one-to-all personalized communication or the scatter operation is the ***gather*** operation, or ***concatenation***, in which a single node collects a unique message from each node.

- A gather operation is different from an **all-to-one** reduce operation in that it does not involve any combination or reduction of data.

Scatter

- The `scatter` operation 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)
```

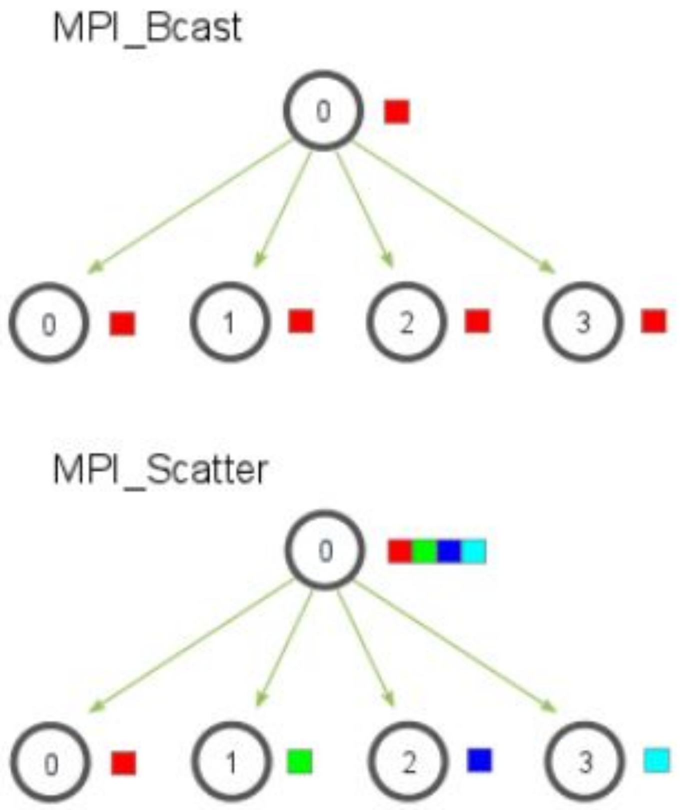
- The `source` process sends a different part of the send buffer `sendbuf` to each processes, including itself.

Scatter

- The data that are received are stored in `recvbuf` .
- Process `i` receives `sendcount` contiguous elements of type `senddatatype` starting from the `i`.
- `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.

Scatter vs Broadcast

- The primary difference between **MPI_Bcast** and **MPI_Scatter** that MPI_Bcast sends the same piece of data to all processes while MPI_Scatter sends chunks of an array to different processes.
    - MPI_Scatter takes an array of elements and distributes the elements in the order of process rank.
        - The first element (in red) goes to process zero, the second element (in green) goes to process one, and so on.
    - Although the source process (process zero) contains the entire array of data



Gather

- The gather operation 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 `comm` , the `target` process receives a total of `p` buffers.

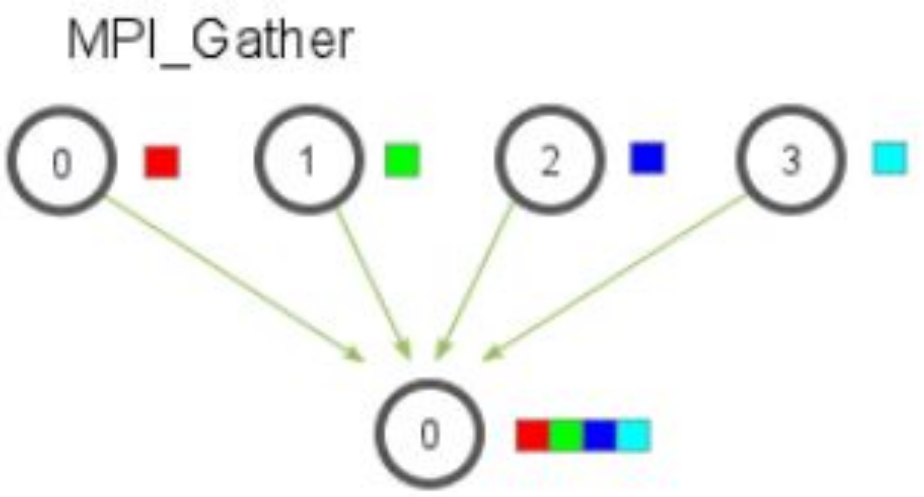
Gather

- The data is stored in the array `recvbuf` of the target process, in a rank order.

- 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.

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.
  - The elements are ordered by the rank of the process from which they were received.
- This routine is highly useful to many parallel algorithms, such as parallel sorting and searching.



Task(s)

1. Generate a random array of numbers on the root process (process 0).
   - Scatter the numbers to all processes, giving each process an equal amount of numbers.
   - Each process computes the average of their subset of the numbers.
   - Gather all averages to the root process. The root process then computes the average of these numbers to get the final average.
   - The root process should also display the **maximum among local averages**.

2. Write a parallel MPI program to calculate the wordcount in a large text file (containing more that 2K words).

```
if (world_rank == 0) {
  rand_nums = create_rand_nums(elements_per_proc * world_size);
}

// Create a buffer that will hold a subset of the random numbers
float *sub_rand_nums = malloc(sizeof(float) * elements_per_proc);

// Scatter the random numbers to all processes
MPI_Scatter(rand_nums, elements_per_proc, MPI_FLOAT, sub_rand_nums,
            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);

// Compute the total average of all numbers.
if (world_rank == 0) {
  float avg = compute_avg(sub_avgs, world_size);
}
```

- 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.

All-to-All Personalized Communication

- In ***all-to-all personalized communication***, each node (process) sends a distinct message of size *m* to every other node.

- Each node sends different messages to different nodes, unlike all-to-all broadcast, in which each node sends the same message to all other nodes.
- All-to-all personalized communication is also known as ***total exchange***.
  - This operation is used in a variety of parallel algorithms such as fast Fourier transform, matrix transpose, sample sort, and some parallel database join operations.

- The all-to-all personalized communication operation 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.

Task

- For every group communication function, explain two algorithms/problems/systems which utilize them.

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.

- Similarly, a blocking receive operation returns only after the message has been received and copied into the receive buffer.

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.

- The calling sequence of `MPI_Isend` is:

```
int MPI_Isend(void *buf, int count, MPI_Datatype
datatype, int dest, int tag, MPI_Comm comm,
MPI_Request *request)
```

- The calling sequence of `MPI_Irecv` is:

```
int MPI_Irecv(void *buf, int count, MPI_Datatype datatype,
int source, int tag, MPI_Comm comm, MPI_Request *request)
```

- 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 it 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.

MPI_Test and MPI_Wait

- To check the completion of non-blocking send and receive operations, MPI provides a pair of functions `MPI_Test` and `MPI_Wait`.
- `MPI_Test` tests whether or not a non-blocking operation has finished.
- `MPI_Wait` waits (i.e., gets blocked) until a non-blocking operation actually finishes.

- Note that these functions have similar arguments as the 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)
```

- `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 de-allocated and request is set to `MPI_REQUEST_NULL`. Also the status object is set to contain information about the operation.

- The `MPI_Wait` function blocks until the non-blocking operation identified by `request` completes.

- In that case it de-allocates the request object, sets it to `MPI_REQUEST_NULL`, and returns information about the completed operation in the `status` object.