Lecture 8

Programming Using the Message-Passing Paradigm IV

MPI: the Message Passing Interface; Overlapping, Multicast

IKC-MH.57 *Introduction to High Performance and Parallel Computing* at December 08, 2023

Dr. Cem Özdoğan
Engineering Sciences Department
İzmir Kâtip Çelebi University

Contents

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
  - either into a system buffer at the source process
  - or sent to the destination process.
- Similarly, a **blocking receive operation** returns only after the message has been received and copied into the receive buffer.
- It will be preferable if we can **overlap the transmission of the data with the computation**.

Non-Blocking Communication Operations I

- In order to overlap communication with computation, MPI provides a pair of functions for performing non-blocking send and receive operations.
  - MPI_Isend ⟹ starts a send operation but **does not complete**, that is, *it returns before the data is copied out of the buffer*.
  - MPI_Irecv ⟹ starts a receive operation but **returns** *before the data has been received and copied into the buffer*.

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

- **MPI_Isend** and **MPI_Irecv** functions **allocate a request object** and return a pointer to it in the *request* variable.
- **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 its computations.

Non-Blocking Communication Operations II

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

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

- The *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.

Non-Blocking Communication Operations III

- **MPI_Test** tests whether or not the non-blocking send or receive operation identified by its *request* has finished.

True — It returns flag = true (non-zero value in C) if it is completed.
- The *request* object pointed to by *request* is deallocated and *request* is set to *MPI_REQUEST_NULL*.
- Also the *status* object is set to contain information about the operation.

False — It returns flag = false (a zero value in C) if it is not completed.
- The *request* is not modified and the value of the *status* object is undefined.
- The **MPI_Wait** function blocks until the non-blocking operation identified by *request* completes.

- A non-blocking communication operation can be matched with a corresponding blocking operation.
- For example, a process can send a message using a non-blocking send operation and this message can be received by the other process using a blocking receive operation.

Non-Blocking Communication Operations IV

- *Avoiding Deadlocks;* by using non-blocking communication operations we can remove most of the deadlocks associated with their blocking counterparts.
- For example, the following piece of code is not safe.

```
int a[10], b[10], myrank;
MPI_Status status;
 ...
MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
if (myrank == 0) {
  MPI_Send(a, 10, MPI_INT, 1, 1, MPI_COMM_WORLD);
  MPI_Send(b, 10, MPI_INT, 1, 2, MPI_COMM_WORLD);
}
else if (myrank == 1) {
  MPI_Recv(b, 10, MPI_INT, 0, 2, &status, MPI_COMM_WORLD);
  MPI_Recv(a, 10, MPI_INT, 0, 1, &status, MPI_COMM_WORLD);
}
...
```

- However, if we replace either the send or receive operations with their non-blocking counterparts, then the code will be safe, and will correctly run on any MPI implementation.

Non-Blocking Communication Operations V

- <u>Safe</u> with non-blocking communication operations;

```c
int a[10], b[10], myrank;
MPI_Status status;
MPI_Request requests[2];
...
MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
if (myrank == 0) {
  MPI_Send(a, 10, MPI_INT, 1, 1, MPI_COMM_WORLD);
  MPI_Send(b, 10, MPI_INT, 1, 2, MPI_COMM_WORLD);
}
else if (myrank == 1) {
  MPI_Irecv(b, 10, MPI_INT, 0, 2, &requests[0],
     MPI_COMM_WORLD);
  MPI_Irecv(a, 10, MPI_INT, 0, 1, &requests[1],
     MPI_COMM_WORLD);
} //Non-Blocking Communication Operations
...
```

- This example also illustrates that the non-blocking operations started by any process can <u>finish in any order</u> depending on the transmission or reception of the corresponding messages.
- For example, the second receive operation will finish before the first does.

Collective Communication and Computation Operations I

- MPI provides an extensive set of functions for performing commonly used collective communication operations.
  - All of the collective communication functions provided by MPI take as an argument a communicator that defines the group of processes that participate in the collective operation.
  - All the processes that belong to this communicator **participate** in the operation,
  - and *all of them must call* the collective communication function.
- Even though collective communication operations do not act like **barriers**,
- act like a virtual synchronization step.
- **Barrier;** the barrier synchronization operation is performed in MPI using the MPI_Barrier function.

```
int MPI_Barrier(MPI_Comm comm)
```

- The call to **MPI_Barrier** *returns only after all* the processes in the group have called this function.

Broadcast I

- **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** sends the data stored in the buffer *buf* of process *source* to all the other processes in the group.
- The data that is broadcast consist of *count* entries of type *datatype*.
- The data received by each process is stored in the buffer *buf*.
- Since the operations are virtually synchronous, they do not require tags.

Broadcast II

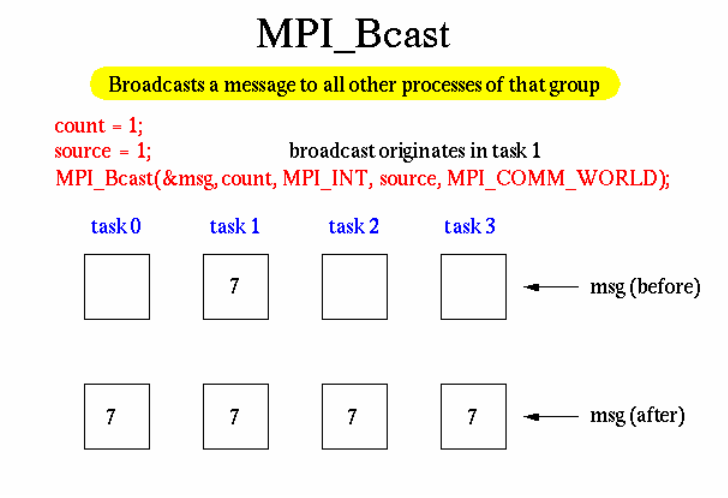

Figure: Diagram for Broadcast.

Reduction I

- **Reduction**; the all-to-one reduction operation 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)
```

  - **combines** the elements stored in the buffer *sendbuf* of each process in the group,
  - using the operation specified in ***op***,
  - returns the combined values in the buffer *recvbuf* of the process with rank *target*.
- Both the *sendbuf* and *recvbuf* must have the same number of *count* items of type *datatype*.
- When *count* is more than one, then the combine operation is applied element-wise on each entry of the sequence.
- Note that all processes must provide a *recvbuf* array, even if they are not the *target* of the reduction operation.

Reduction II

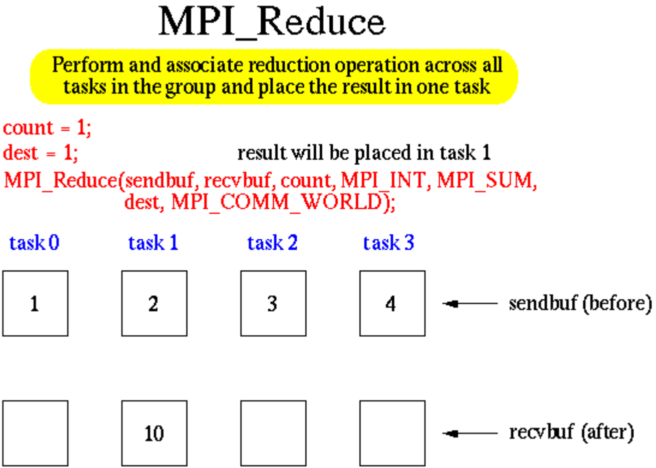

Figure: Diagram for Reduce.

Reduction III

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

Table: Predefined reduction operations.

Operation	Meaning	Datatypes
MPI_MAX	Maximum	C integers and floating point
MPI_MIN	Minimum	C integers and floating point
MPI_SUM	Sum	C integers and floating point
MPI_PROD	Product	C integers and floating point
MPI_LAND	Logical AND	C integers
MPI_BAND	Bit-wise AND	C integers and byte
MPI_LOR	Logical OR	C integers
MPI_BOR	Bit-wise OR	C integers and byte
MPI_LXOR	Logical XOR	C integers
MPI_BXOR	Bit-wise XOR	C integers and byte
MPI_MAXLOC	max-min value-location	Data-pairs
MPI_MINLOC	min-min value-location	Data-pairs

- MPI also allows programmers to define their own operations.

Gather I

- **Gather**; the all-to-one 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, the *target* process receives a total of *p* buffers (*p* is the number of processors in the communication *comm*).
- The data is stored in the array *recvbuf* of the target process, in a rank order.
- That is, the data from process with rank *i* are stored in the *recvbuf* starting at location *i * sendcount* (assuming that the array *recvbuf* is of the same type as *recvdatatype*).

Gather II

- 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.
- The information about the receive buffer, its length and type applies only for the target process and is ignored for all the other processes.
- The argument *recvcount* specifies the number of elements received by each process and not the total number of elements it receives.
- So, *recvcount* must be the same as *sendcount* and their datatypes must be matching.

Gather III

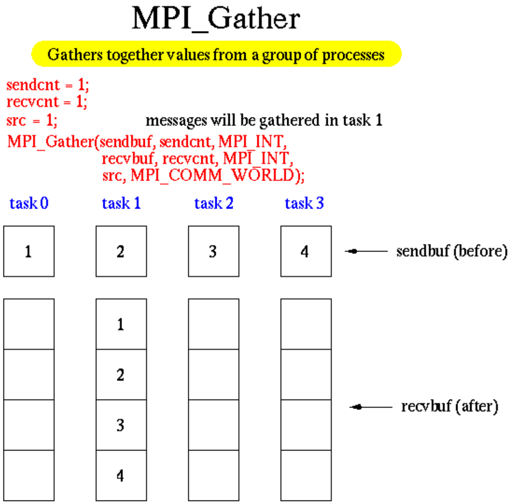

Figure: Diagram for Gather.

Gather IV

- 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.

Gather V

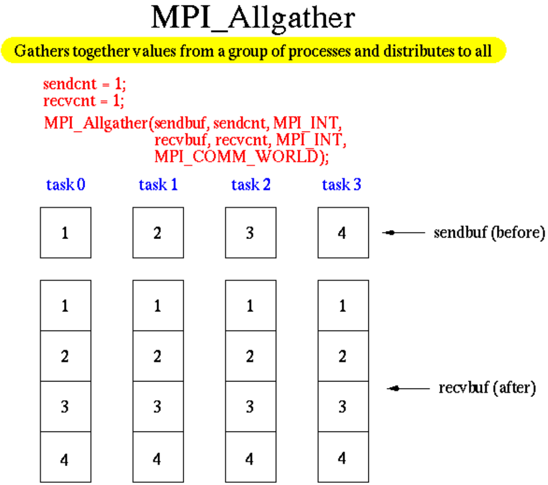

Figure: Diagram for All_Gather.

Scatter I

- **Scatter**; the one-to-all 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.
- The data that are received are stored in *recvbuf*.
- Process *i* receives *sendcount* contiguous elements of type *senddatatype* starting from the *i * sendcount* location of the *sendbuf* of the *source* process (assuming that *sendbuf* is of the same type as *senddatatype*).

Scatter II

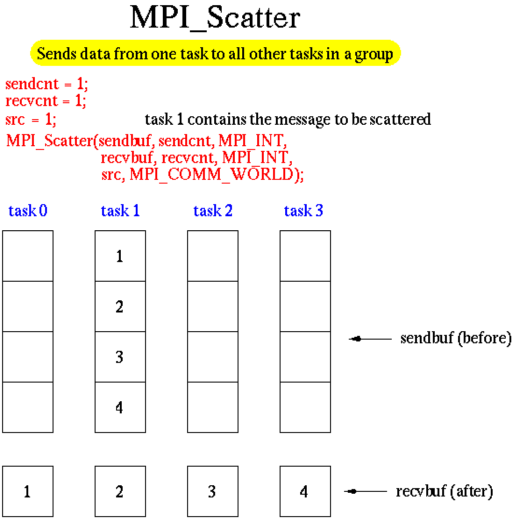

Figure: Diagram for Scatter.

All-to-All I

- **Alltoall**; the all-to-all 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.
- Each process sends to process *i sendcount* contiguous elements of type *senddatatype* starting from the *i * sendcount* location of its *sendbuf* array.
- The data that are received are stored in the *recvbuf* array.
- Each process receives from process *i recvcount* elements of type *recvdatatype* and stores them in its *recvbuf* array starting at location *i * recvcount*.

All-to-All II

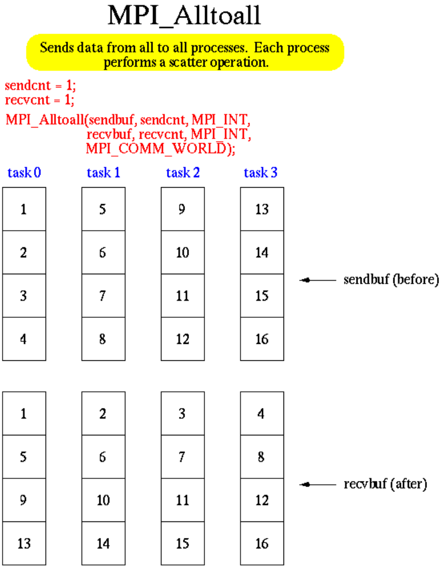

Figure: Diagram for Alltoall.