Parallel Computing

Ekkapot Charoenwanit

Software Systems Engineering
TGGS
KMUTNB

Lecture 13:

- ❑ Distributed-Memory Programming with MPI
  - Blocking Collective Communication

MPI: Collective Communication

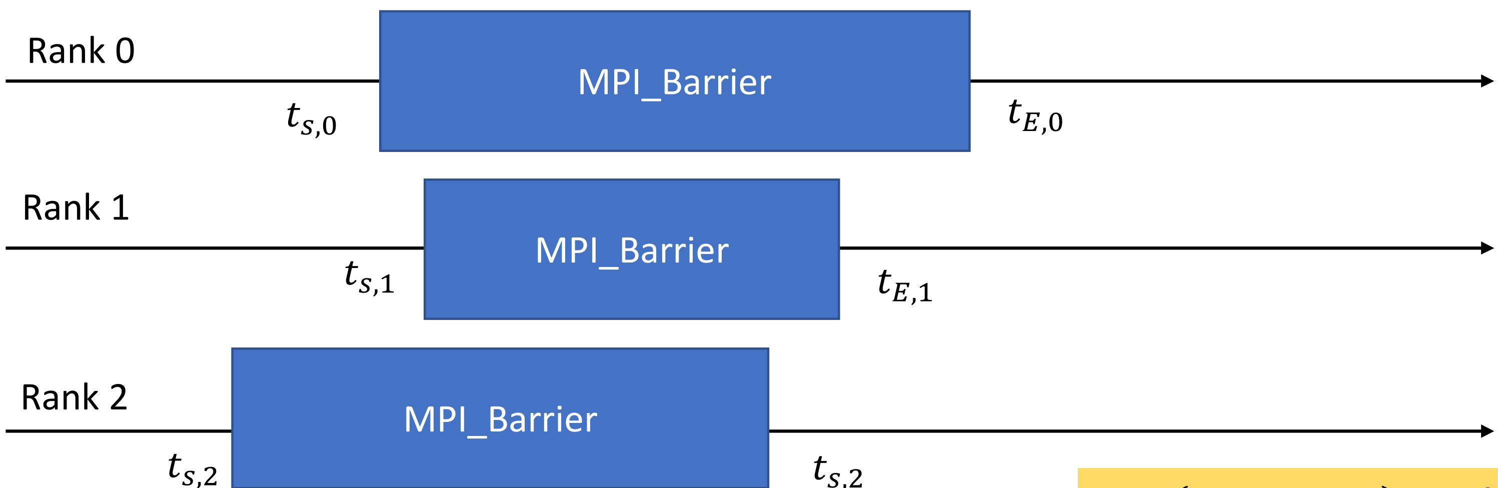
A collective operation involves all ranks in a given communicator:

- All the ranks in the communicator work together to distribute or gather a set of one or multiple values.
- All ranks must call the same MPI operations to succeed.
- There must be one call per MPI rank, i.e., not per thread.

MPI: Barrier

The barrier operation serves as an **explicit synchronization operation** in MPI.

MPI_Barrier(MPI_Comm comm) C

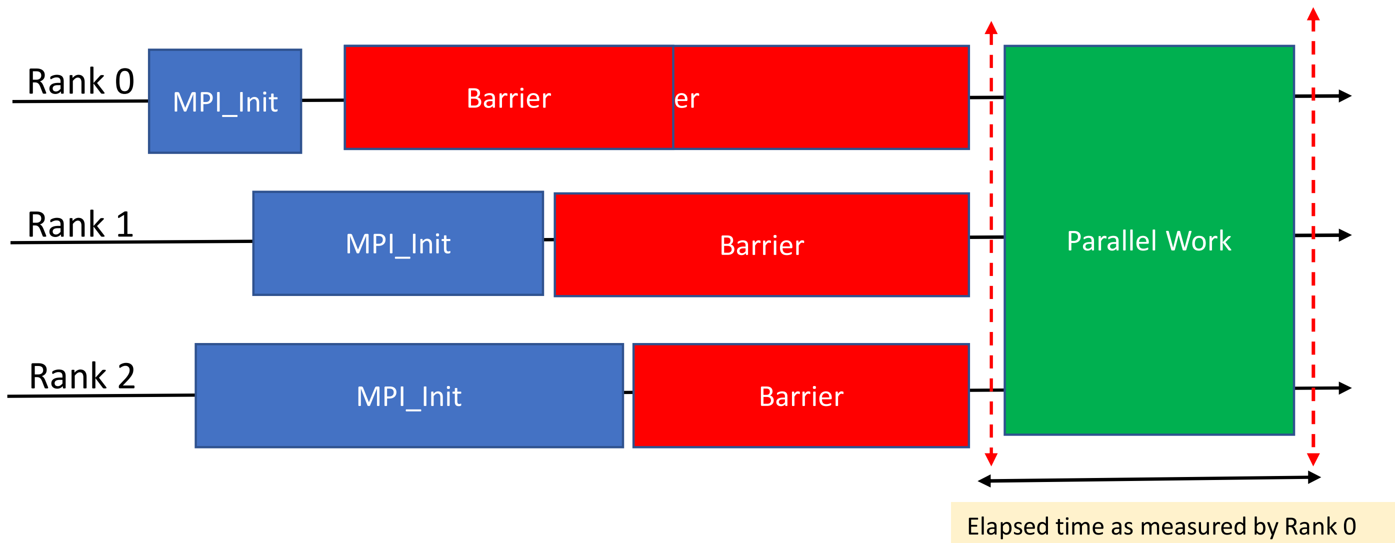


$$\max(t_{s,0}, t_{s,1}, t_{s,2}) < \min(t_{E,0}, t_{E,1}, t_{E,2})$$

MPI: Barrier

Barriers are useful for benchmarking:

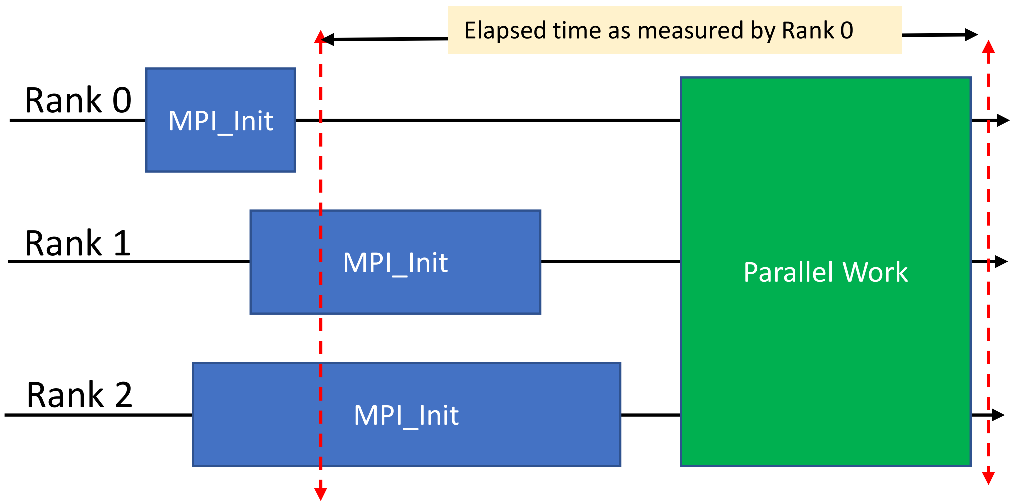
- They can be used to synchronize before taking time measurements.



MPI: Barrier

Barriers are useful for benchmarking:

- They can be used to synchronize before taking time measurements.



Without barriers, there is huge discrepancy between the time measurement and the parallel time of the parallel work.

MPI: Broadcast

The broadcast operation is a **one-to-many** collective operation:

- It distributes data from one rank to all the other ranks in the same communicator.

C

```
MPI_Broadcast(void * data, int count, MPI_Datatype type, int root,  MPI_Comm comm)
```

- ***data :*** send buffer on the ***root*** rank
  receiver buffer on the other ranks
- ***count :*** the number of data elements
- ***type :*** the type of the elements
- ***root :*** source rank ; all ranks must specify the same value
- ***comm :*** communicator

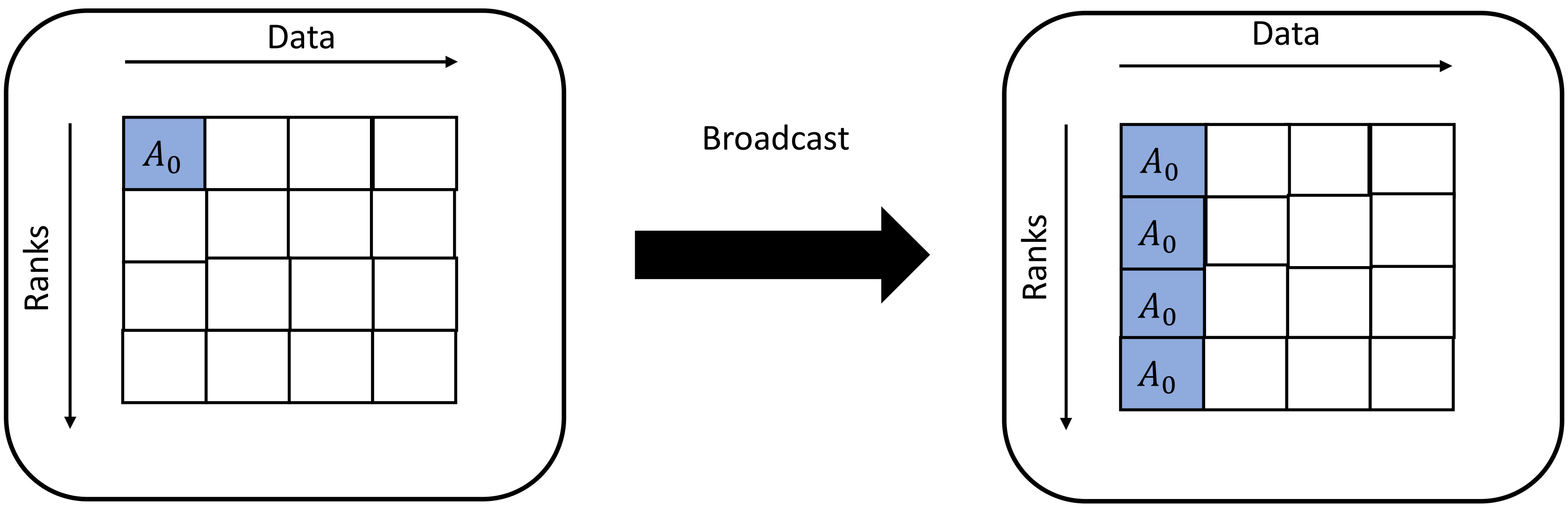
MPI: Broadcast

C

```
MPI_Broadcast(void * data, int count, MPI_Datatype type, int root,  MPI_Comm comm)
```

- On the ***root*** rank, the ***data*** buffer acts as an input argument.
- On all ranks, but the **root** rank, the ***data*** buffer acts as an output argument.
- The type signatures must match across all ranks in the given communicator.

MPI: Broadcast



MPI: Broadcast

Correct Usage: all ranks must call **MPI_Bcast**.

C

```
int data;
If(rank==0)//rank 0 is the root
		data = read_input_from_user();
MP_Bcast( &data , 1 , MPI_INT , 0 , MPI_COMM_WORLD );
```

MPI: Broadcast

Wrong Usage: only the root calls **MPI_Bcast**.

- This leads to **deadlock** !

C

```
int data;
If(rank==0){//rank 0 is the root
    data = read_input_from_user();
    MP_Bcast( &data , 1 , MPI_INT , 0 , MPI_COMM_WORLD );
}
```

MPI: Broadcast

Naïve Implementation:

C

```
void broadcast(void * data , int count , MPI_Datatype type , int root , MPI_Comm comm ){
        int rank , size;
        MPI_Comm_rank(MPI_COMM_WORLD,&rank);
        MPI_Comm_size(MPI_COMM_WORLD,&size);
        if(rank==root){
                for(int i=0;i<size;i++){
                        if(i!=root){
                                MPI_Send( data , count , type , i , TAG_BCAST , comm );
                        }
                }
        }else{
                MPI_Receive( data , count , type , root , TAG_BCAST , comm ,  MPI_STATUS_IGNORE );
        }
}
```

MPI: Scatter

The scatter operation is a **one-to-many** collective operation:

- It distributes chunks of data from one rank to all ranks in the same communicator.

C

```
MPI_Scatter(void * sendbuf, int sendcount, MPI_Datatype sendtype,
            void* recvbuf, int recvcount, MPI_Datatype recvtype,
            int root,  MPI_Comm comm)
```

These arguments matter only on the root rank

MPI: Scatter

The scatter operation is a **one-to-many** collective operation:

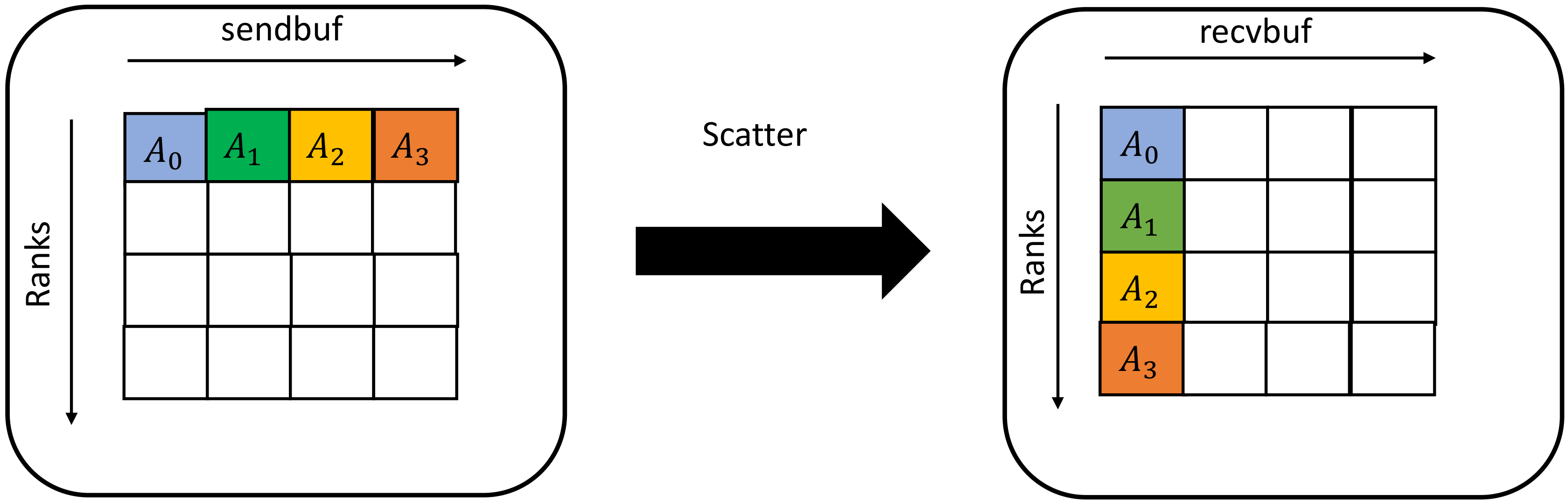
- It distributes chunks of data from one rank to all ranks in the same communicator.

C

```
MPI_Scatter(void * sendbuf, int sendcount, MPI_Datatype sendtype,
            void* recvbuf, int recvcount, MPI_Datatype recvtype,
            int root,  MPI_Comm comm)
```

- ***sendbuf*** must be large enough in order to supply ***sendcount*** elements of data to each rank in the communicator.
  - ***sendbuf*** can accommodate at least $\boldsymbol{p * sendcount}$ elements of ***sendtype***
- Data chunks are taken in increasing order of ranks.
- The ***root*** rank also sends data to itself.
- Type signatures must match across all ranks.

MPI: Scatter



MPI: Scatter

C

```
MPI_Scatter(void * sendbuf, int sendcount, MPI_Datatype sendtype,
            void* recvbuf, int recvcount, MPI_Datatype recvtype,
            int root,  MPI_Comm comm)
```

sendbuf is only accessed on the ***root*** rank.

recvbuf is accessed and written into on all ranks.

Each rank receives 2 elements into recvbuff.

The root sends 2 elements from sendbuf to each rank.

Usage:

C

```
int bigdata[8]; //2 x 4 elements
int localdata[2];
MPI_Scatter( bigdata , 2 , MPI_INT , localdata, 2 , MPI_INT ,  0 , MPI_COMM_WORLD );
```

MPI: Gather

The gather operation is a **many-to-one** collective operation:

- It distributes chunks of data from all ranks to one rank, i.e., the root.

C

```
MPI_Gather(void * sendbuf, int sendcount, MPI_Datatype sendtype,
           void* recvbuf, int recvcount, MPI_Datatype recvtype,
           int root,  MPI_Comm comm)
```

These arguments matter only on the root rank

MPI: Gather

The gather operation is a **many-to-one** collective operation:

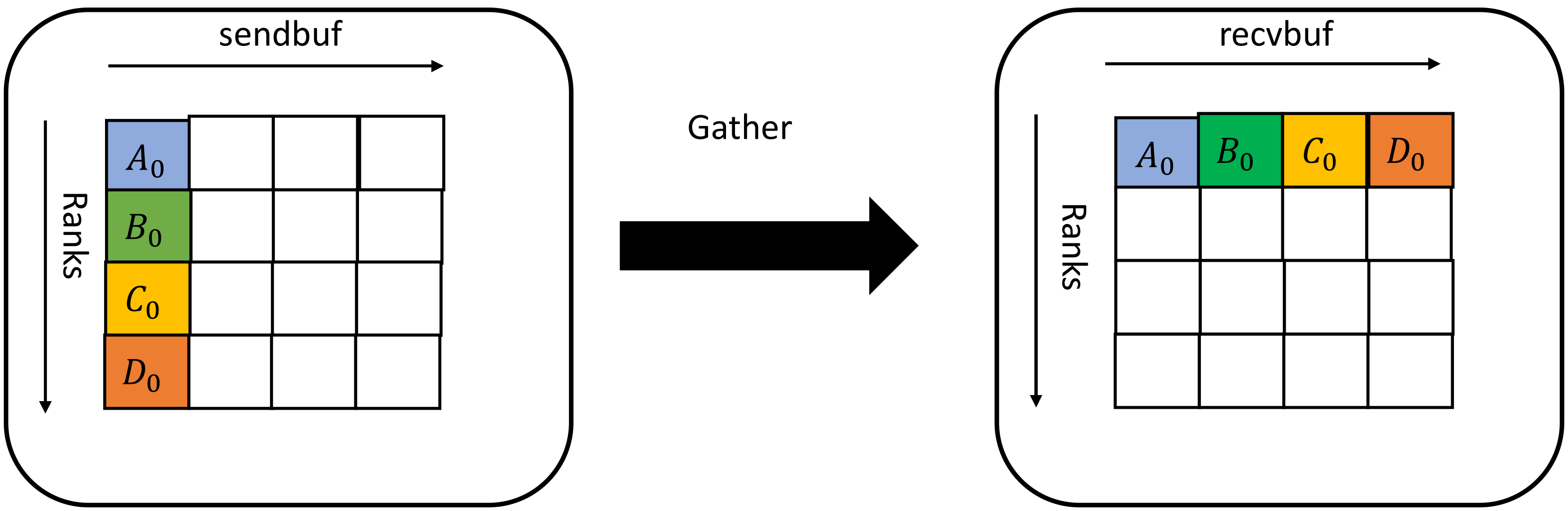
- It distributes chunks of data from all ranks to one rank, i.e., the root.

C

```
MPI_Gather(void * sendbuf, int sendcount, MPI_Datatype sendtype,
           void* recvbuf, int recvcount, MPI_Datatype recvtype,
           int root,  MPI_Comm comm)
```

- The gather operation is the inverse of the scatter operation.
- ***recvbuf*** must be large enough to accommodate ***recvcount*** elements from each rank.
  - ***recvbuf*** can accommodate at least $\boldsymbol{p * recvcount}$ elements of ***recvtype***
- The ***root*** rank also receives one chunk of data from itself.
- The elements are ordered by the rank of the process from which they were received.
- The type signature of the senders must match that of the receiver.

MPI: Gather



MPI: All-Gather

The all-gather operation is a **many-to-many** collective operation:

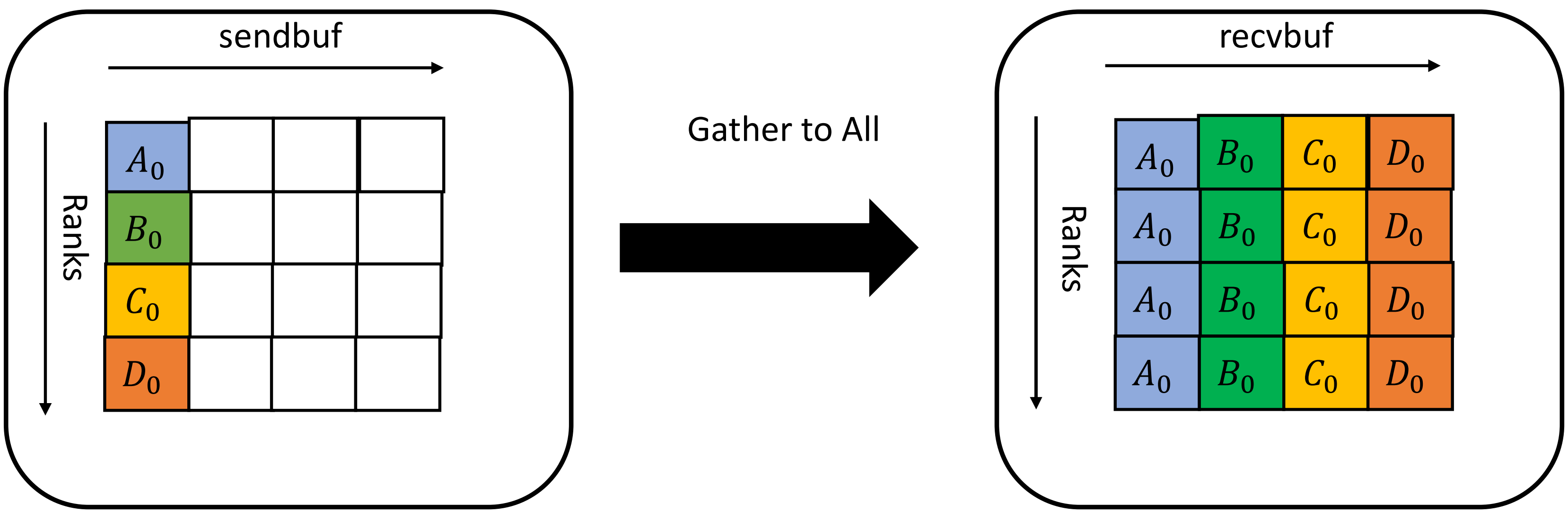
- It collects chunks of data from all ranks to all ranks.

C

```
MPI_Allgather(void * sendbuf, int sendcount, MPI_Datatype sendtype,
              void* recvbuf, int recvcount, MPI_Datatype recvtype,
              MPI_Comm comm)
```

- There is **no root rank** as all ranks receive a copy of the gathered data.
  - The function signature is almost identical to that of **MPI_Gather** except that there is no root.
- Each rank receives one chunk of data from itself.
- Data chunks are stored in increasing order of the sender's rank.
- Type signatures match across all ranks.
- Logically, it is equivalent to **MPI_Gather** followed by **MPI_Bcast**:
  - The actual implementation can potentially outperform such a straightforward implementation through some clever tricks.

MPI: All-Gather



MPI: All-to-All

The all-to-all operation is a **many-to-many** collective operation:

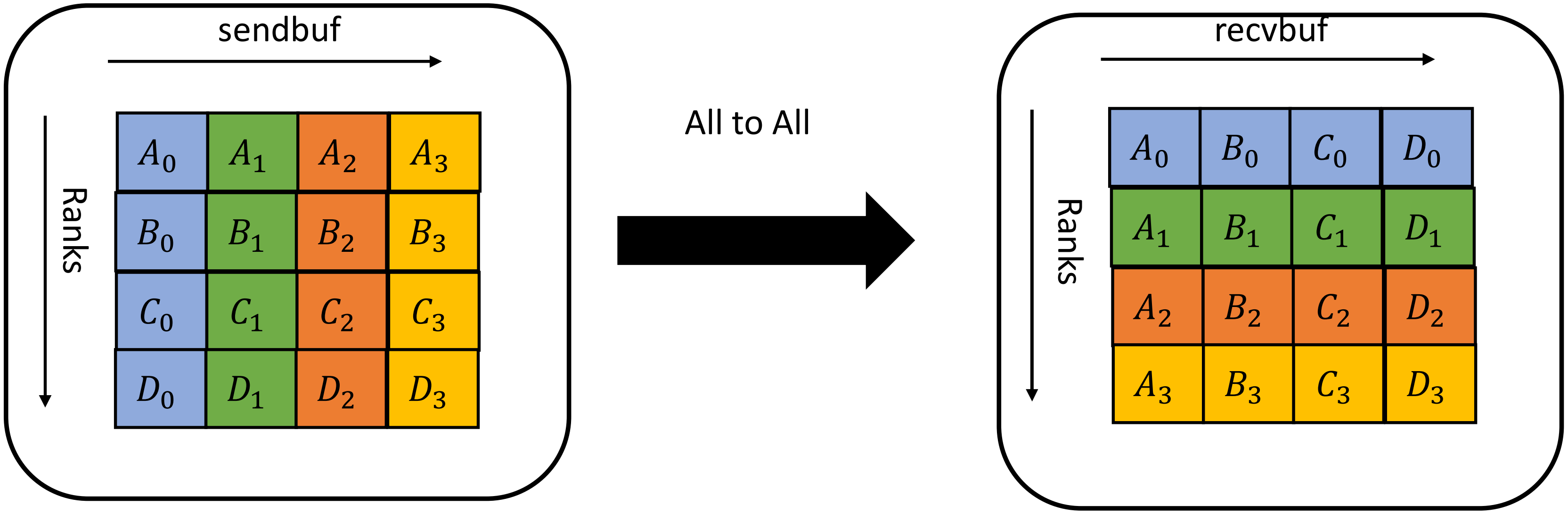
- It is a collective operation that combines the scatter and the gather operation.

C

```
MPI_Alltoall(void * sendbuf, int sendcount, MPI_Datatype sendtype,
             void* recvbuf, int recvcount, MPI_Datatype recvtype,
             MPI_Comm comm)
```

- Each rank distributes its ***sendbuf*** to every rank in the communicator including itself.
- Data chunks are read in increasing order of the receiver's rank.
- Data chunks are stored in increasing order of the sender's rank.
- Equivalent to **MPI_Scatter** + **MPI_Gather**

MPI: All-to-All



MPI: Reduce

The reduce operation is a **many-to-one** collective operation:

- It performs an arithmetic reduction on the gathered data.

C

MPI_Reduce(void * sbuf, void * rbuf, int count,
MPI_Datatype type, MPI_Op op, int root , MPI_Comm comm)

- ***sbuf*** : data to be reduced
- ***rbuf*** : location of the result (it matters only at the ***root*** rank)
- ***count*** : the number of elements per rank
- ***type*** : data type to be reduced
- ***op*** : reduction operation handle
- ***root*** : the root rank that performs the reduction operation on the gathered data
- ***comm*** : the communicator

MPI: Reduce

MPI_Op.	Result Value
MPI_MAX	Maximum Value
MPI_MIN	Minimum Value
MPI_SUM	Sum of all values
MPI_PROD	Product of all values
MPI_LAND	Logical AND of all values
MPI_BAND	Bitwise AND of all values
MPI_LOR	Logical OR of all values
…	…

- All the predefined reduction operations are **associative** and **commutative**.
- Pay attention to **non-commutative effects on floats**.

MPI: Reduce

$$\boldsymbol{rbuf[i] = sbuf_0[i] \oplus sbuf_1[i] \oplus sbuf_2[i] \ldots \oplus sbuf_{p-1}[i]}$$

$sbuf_0[\,]$	1	2	3	4	5	6
$sbuf_1[\,]$	10	20	30	40	50	60
$sbuf_2[\,]$	10	20	30	40	50	60
$sbuf_3[\,]$	100	200	300	400	500	600
$rbuf[\,]$	111	222	333	444	555	666

$\oplus = MPI\_SUM$

MPI: All-Reduce

The all-reduce operation is a **many-to-many** collective operation:

- The reduction result is available on all ranks in the given communicator.

C

MPI_Allreduce(void * sbuf, void * rbuf, int count,
MPI_Datatype type, MPI_Op op, MPI_Comm comm)

- It is logically equivalent to **MPI_Reduce** followed by **MPI_Bcast** using the same root rank.

MPI: Advantages of Collective Operations

- MPI Collective Operations implement common SPMD patterns portably.
- Their implementations are vendor-specific but their behaviors all conform to what the MPI standard specifies.
- Example: Broadcast
  - Naïve broadcast algorithm : $O(p)$
  - Binomial-tree-based broadcast algorithm: $O(\log p)$
  - Other smarter broadcast algorithms: $O(1)$ [See **[3]**]

MPI: Summary

- All ranks in the communicator must call the collective operation.
  - Be careful with the number of elements and data type of data on the sender and the receiver end.
- The order of collective calls must be the same across all ranks.
- MPI_Barrier is the only explicit synchronization collective in MPI.
  - Some may synchronize implicitly, e.g. MPI_Alltoall
- Communication paradigms do not interfere with one another.
  - Collective communication does not interfere with point-to-point communication on the same communicator.

References

[1] *William Gropp, Ewing Lusk, and Anthony Skjellum. 2014. Using MPI: Portable Parallel Programming with the Message-Passing Interface. The MIT Press.*

[2] *Marc-Andre Hermanns. 2021. MPI in Small Bites. PPCES 2021.*

[3] *T. Hoefler, C. Siebert and W. Rehm, "A practically constant-time MPI Broadcast Algorithm for large-scale InfiniBand Clusters with Multicast," 2007 IEEE International Parallel and Distributed Processing Symposium, 2007, pp. 1-8, doi: 10.1109/IPDPS.2007.370475.*