Parallel and Distributed Computing

CS3006

Lecture 17

MPI-IV

29th April 2024

Dr. Rana Asif Rehman

Collective Communication and Computation Operations

Collective Communication and Computation Operations

- MPI provides its **own optimized implementations** for most of the collective operations that we performed in CH#4
- These operations are called collective as all the processes must have a call to collective functions
- Every collective operation take a communicator (such as MPI_COMM_WORLD) as argument
  - all the processes within that communicator must have a corresponding call to the operation

Collective Communication and Computation Operations

Barrier synchronization operation

- The barrier synchronization operation is performed in MPI using:

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

- The call to the MPI_barrier returns only all the processes in the group have called this function

Collective Communication and Computation Operations

The one-to-all broadcast :

```
int MPI_Bcast(void *buf, int count, MPI_Datatype datatype,
        int source, MPI_Comm comm)
```

- Buffer of the source process is copied to the buffers of other processes

Collective Communication and Computation Operations

The all-to-one reduction operation

- Dual of one-to-all broadcast
- Every process including target provides sendbuf for its value that is to be used for the reduction
- After the reduction, reduced value is stored in recvbuf of target process
- Every process must also provide recvbuf, though it may not be target of the reduction

```
int MPI_Reduce(void *sendbuf, void *recvbuf, int count,
 MPI_Datatype datatype, MPI_Op op, int target,
 MPI_Comm comm)
```

- Here MPI_Op is MPI defined set of operations for reduction

Collective Communication and Computation Operations

The all-to-one reduction operation

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

Collective Communication and Computation Operations

MPI_MAXLOC and MPI_MINLOC

- The operation `MPI_MAXLOC` combines pairs of values (v_i, l_i) and returns the pair $(v, \bar{l})$ such that v is the maximum among all v_i 's and l is the corresponding l_i (if there are more than one, it is the smallest among all these l_i 's).
- `MPI_MINLOC` does the same, except for minimum value of v_i.

Value	15	17	11	12	17	11
Process	0	1	2	3	4	5

```
MinLoc(Value, Process) = (11, 2)
MaxLoc(Value, Process) = (17, 1)
```

An example use of the `MPI_MINLOC` and `MPI_MAXLOC` operators.

Collective Communication and Computation Operations

MPI_MAXLOC and MPI_MINLOC

MPI datatypes for data-pairs used with the `MPI_MAXLOC` and `MPI_MINLOC` reduction operations.

MPI Datatype	C Datatype
`MPI_2INT`	pair of ints
`MPI_SHORT_INT`	short and int
`MPI_LONG_INT`	long and int
`MPI_LONG_DOUBLE_INT`	long double and int
`MPI_FLOAT_INT`	float and int
`MPI_DOUBLE_INT`	double and int

Collective Communication and Computation Operations

The All-Reduce operation

- MPI_AllReduce is used when the result of the reduction operation is needed by all processes
- Equal to All-to-one reduction followed by one-to-all broadcast

```
int MPI_Allreduce(void *sendbuf, void *recvbuf,
      int count, MPI_Datatype datatype, MPI_Op op,
      MPI_Comm comm)
```

- After Allreduce operation, recvbuf of all the processes contain reduced value
- **Note:** no target for reduction is given

Collective Communication and Computation Operations

Prefix(scan) operation

- Recall 4.3-for prefix-sum: After the operation, every process has sum of the buffers of the previous processes and its own.
- MPI_Scan() is MPI primitive for the prefix operations.
- All the operators that can be used for reduction can also be used for the scan operation
- If buffer is an array of elements, then recvbuf is also an array containing **element-wise prefix** at each position.

```
int MPI_Scan(void *sendbuf, void *recvbuf, int
      count, MPI_Datatype datatype, MPI_Op op,
      MPI_Comm comm)
```

Collective Communication and Computation Operations

Exclusive-Prefix (Exscan) operation

- Exclusive-prefix-sum: After the operation, every process has sum of the buffers of the previous processes excluding its own.
- MPI_Exscan() is MPI primitive for the exclusive-prefix operations.
- The recvbuf of first process is **remains unchanged** as there is no process before it. Some MPI distributions place **identity value** for the given associative operator.

```
int MPI_Excan(void *sendbuf, void *recvbuf, int
          count, MPI_Datatype datatype, MPI_Op op,
      MPI_Comm comm)
```

Collective Communication and Computation Operations

MPI_Gather and its variants

- Recall section 4.4: After the **Gather** operation, a single target process accumulates[concatenates] buffers of all the other processes without any reduction operator.
- Each process sends element(s) in its *sendbuf* to the target process.
- Total number of elements to be sent by each process must be same. This number is specified in *sendcount* and is equal to *recvcount*.
- On the target, *recvbuf* stores elements sent by all the processes in rank order. Elements received at target by process#i, will be stored starting from (i*sendcount)th index of *recvbuf*.

```
int MPI_Gather(void *sendbuf, int sendcount,
       MPI_Datatype senddatatype, void *recvbuf,
       int recvcount, MPI_Datatype recvdatatype,
       int target, MPI_Comm comm)
```

Collective Communication and Computation Operations

MPI_Gather and its variants

- Gatherv
  - Each process can have different message length.
  - Recvcounts[i] = Total elements to be received by ith processing node
  - Displs[i]= starting index in *recvbuf* to store message received from ith process.

```
int MPI_Gatherv(void *sendbuf, int sendcount,
        MPI_Datatype senddatatype, void *recvbuf,
        int *recvcounts,int *displs, MPI_Datatype
           recvdatatype,int target, MPI_Comm comm)
```

Collective Communication and Computation Operations

MPI_Gather and its variants

- Gatherv (Displs calculation example)
  - Let each process have elements one more than their rank.
  - Then calculation of *displs[]* at **target** is calculated as:

P0	P1	P2	P3
32	12, 15	4,9,14	20,23,27,31

Recvcounts	1	2	3	4
displs	0	0+1=1	1+2=3	3+3=6

Collective Communication and Computation Operations

MPI_Gather and its variants

- **MPI_Allgather**
- Same as All-to-All broadcast described in section 4.2
- Every process serve as target for the gather

```
int MPI_Allgather(void *sendbuf, int sendcount,
   MPI_Datatype senddatatype, void *recvbuf,
      int recvcount, MPI_Datatype recvdatatype,
      MPI_Comm comm)
```

- Note: No target for gather
- Unlike MPI_Gather, it gathers *sendbufs* of all the processes in *recvbufs* of all the processes.

Collective Communication and Computation Operations

MPI_Gather and its variants

- **MPI_Allgatherv**

```
int MPI_Allgatherv(void *sendbuf, int sendcount,
    MPI_Datatype senddatatype, void  *recvbuf, int
    *recvcounts, int *displs,  MPI_Datatype recvdatatype,
    MPI_Comm comm)
```

- Here every process will have to supply the valid calculated arrays of *recvcounts* and *displs*.
- *Furthermore, it is also necessary for all the processors to provide a recvbuf [an array] of sufficient size to store all the elements of all the processes.*

```
intial values at source::0::=83 86     77     15     93     35     86     92
rank=0: Received:83     86
rank=1: Received:77     15
rank=2: Received:93     35
rank=3: Received:86     92
```

MPI_scatter

- Scatters data stored in *sendbuf* of source process between all the processes as discussed in ch#4.

```
int MPI_Scatter(void *s, int sendcount,
                MPI_Datatype senddatatype, void *recvbuf,
                int recvcount, MPI_Datatype recvdatatype,
                int source, MPI_Comm comm)
```

- Sendcount and recvcount should be the same and represent total elements to be given to each process.

```
intial values at source::0::=83 86      77      15      93      35      86      92      49      21
rank=0: Received:83
rank=1: Received:86     77
rank=2: Received:15     93      35
rank=3: Received:86     92      49      21
```

MPI_scatterv

- Here *sendcounts* is an array of *P* size such that its i^th^ index contains number of elements to be sent to i^th^ process.
- *displs[i]* indicates the index in *sendbuf* from which *sendcounts[i]* values are to be sent to ith process.

```
int MPI_Scatterv(void *sendbuf, int *sendcounts, int *displs,
            MPI_Datatype senddatatype, void *recvbuf, int
    recvcount,  MPI_Datatype recvdatatype, int source,
    MPI_Comm comm)
```

- Values of *sendbuf* and *sendcounts* at all processes except the source are ignored but, you have to provide pointers though pointing to nothing.
- Every process will have to calculate its own *recvcount*

- **Challenge:** Write a program that scatters even rows of a 2P X P array to P processes. Assume that we are using 1d array to simulate 2D data.

Collective Communication and Computation Operations

MPI_AlltoAll

- This routine is used to perform operation known as alltoall personalized communication in CH#4.
- Each process has P messages, one for each process.
  - parallel matrix transpose operations.

```
int MPI_Alltoall(void *sendbuf, int sendcount,
    MPI_Datatype senddatatype, void *recvbuf,
        int recvcount, MPI_Datatype recvdatatype,
    MPI_Comm comm)
```

- Here sendbuf is of size p*message size for each process.
- Size of receive buffer is equal to sendbuf.
- Sendcount and recvcount have same integer value representing elements to be sent to each process and elements to be received from each process, respectively.

- **Read and implement vector variant for alltoall personalized communication**

Collective Communication and Computation Operations

Groups and Communicators

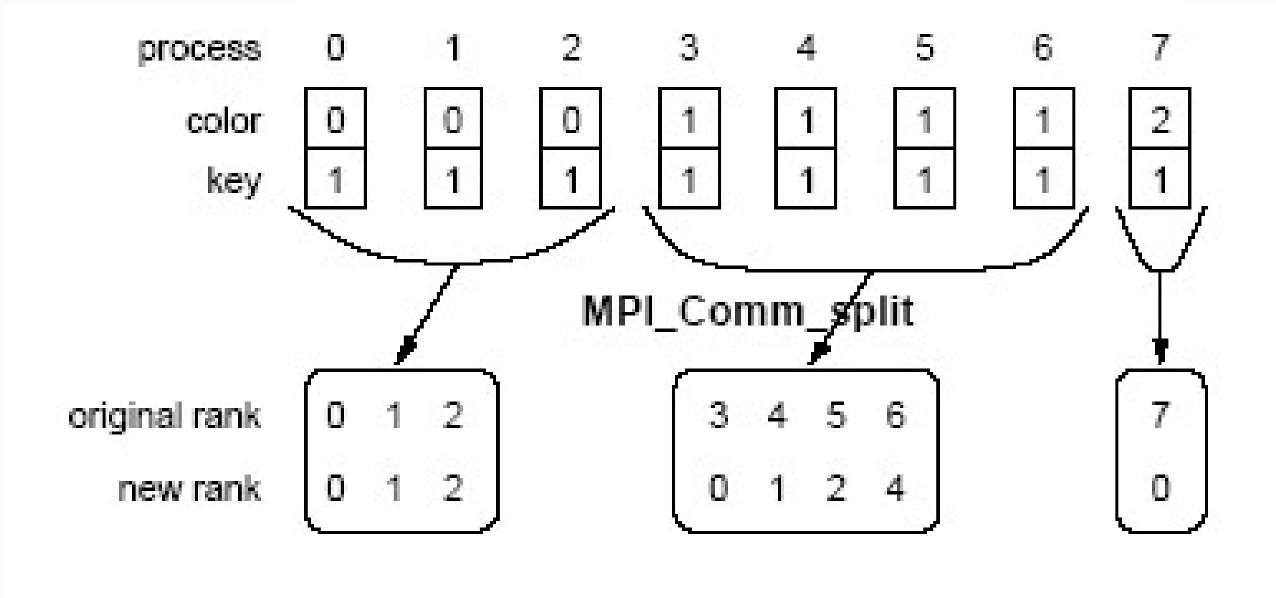
- In many parallel algorithms, communication operations need to be restricted to certain subsets of processes.
- MPI provides mechanisms for partitioning the group of processes that belong to a communicator into subgroups each corresponding to a different communicator.
- The simplest such mechanism is:

```
int MPI_Comm_split(MPI_Comm comm, int color, int key,
                   MPI_Comm *newcomm)
```

- This operation groups processors by color and sorts resulting groups on the key (i.e., highest key→ highest new_rank).

Collective Communication and Computation Operations

Groups and Communicators



Using `MPI_Comm_split` to split a group of processes in a communicator into subgroups.

Questions

References

1. Kumar, V., Grama, A., Gupta, A., & Karypis, G. (2017). *Introduction to parallel computing*. Redwood City, CA: Benjamin/Cummings.