Parallel and Distributed Computing
CS3006 (BCS-6C/6D)
Lecture 20

Instructor: Dr. Syed Mohammad Irteza

Assistant Professor, Department of Computer Science, FAST

06 April, 2023

MPI: Point-to-Point Communication

- Processes can be collected into groups
- Each message is sent in a context, and
- must be received in the same context
- A group and context together form a Communicator
- A process is identified by its rank in the group
- Associated with a communicator
- Messages are sent with an accompanying user defined integer tag, to assist the receiving process in identifying the message
- MPI_ANY_TAG

Finding count, probing messsages

`MPI_Get_count(MPI_Status *status, MPI_Datatype datatype, int *count)`

- `MPI_Get_count()` returns in `count` the number of elements of type `datatype` being received in the message associated with `status`.

`MPI_Probe(int source, int tag, MPI_Comm comm, MPI_Status *status)`

- `MPI_Probe()` synchronizes the reception of the next message by returning in `status` information about the message without effectively receiving it.
- To receive the message, a call to `MPI_Recv()` is required.
- It is useful for cases where we do not know beforehand the size of the message, thus allowing to avoid overflowing the receiving buffer.

Example

```c
int number_amount;
if (world_rank == 0) {
    const int MAX_NUMBERS = 100;
    int numbers[MAX_NUMBERS];
    // Pick a random amount of integers to send to process one
    srand(time(NULL));
    number_amount = (rand() / (float)RAND_MAX) * MAX_NUMBERS;

    // Send the random amount of integers to process one
    MPI_Send(numbers, number_amount, MPI_INT, 1, 0, MPI_COMM_WORLD);
    printf("0 sent %d numbers to 1\n", number_amount);
} else if (world_rank == 1) {
    MPI_Status status;
    // Probe for an incoming message from process zero
    MPI_Probe(0, 0, MPI_COMM_WORLD, &status);

    // When probe returns, the status object has the size and other
    // attributes of the incoming message. Get the message size
    MPI_Get_count(&status, MPI_INT, &number_amount);

    // Allocate a buffer to hold the incoming numbers
    int* number_buf = (int*)malloc(sizeof(int) * number_amount);

    // Now receive the message with the allocated buffer
    MPI_Recv(number_buf, number_amount, MPI_INT, 0, 0,
             MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    printf("1 dynamically received %d numbers from 0.\n",
           number_amount);
    free(number_buf);
}
```

https://mpitutorial.com/tutorials/dynamic-receiving-with-mpi-probe-and-mpi-status/

SPMD model

```
int main (int argc, char *argv[]) {
  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, & myrank); // process rank
  if (myrank == 0)
    master();
  else
    slave();
  MPI_Finalize();
}
```

Another Example

Process other than root generates the random value less than 1 and sends to root. Root sums up and displays sum.

```c
#include <stdio.h>
#include <mpi.h>
#include<stdlib.h>
#include <string.h>
#include<time.h>

int main(int argc, char **argv) {
   int myrank,  p;
   int tag =0, dest=0;
   int i;
   double randIn, randOut;
   int source;

   MPI_Status status;
   MPI_Init(&argc, &argv);
   MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
   MPI_Comm_size(MPI_COMM_WORLD, &p);
```

```c
if(myrank == 0) { // I am the root
   double total=0,average=0;
   for(source = 1; source < p; source++) {
    MPI_Recv(&randIn,1, MPI_DOUBLE, source, MPI_ANY_TAG, MPI_COMM_WORLD, &status);
    printf("Message from root: From %d received number %f\n", source, randIn);
    total += randIn;
   } // end for
   average=total/(p-1);
} // end if
else { // I am other than root
  randOut=rand();
  printf("randout=%f, myrank=%d\n",randOut, myrank);
  MPI_Send(&randOut, 1, MPI_DOUBLE, dest, tag, MPI_COMM_WORLD);
 } // end If-Else
 MPI_Finalize();
 return 0;
}
```

Collective Communication & Computation Operations

Collective Communication and Computation Operations

- MPI provides its ***own optimized implementations*** for most of the collective operations that we performed in chapter 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 an 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)
```

- Blocks until all processes in the communicator have reached this routine.
  - See: https://www.mpich.org/static/docs/v3.2/www3/MPI_Barrier.html

Collective Communication and Computation Operations

The MPI One-to-All broadcast:

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

- The buffer of the source process is copied to the buffers of other processes

Collective Communication and Computation Operations

The MPI All-to-One Reduction operation:

- The 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`, even though it may not be the `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 an 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, 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 of the 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 integers
`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 the `MPI_Allreduce` operation, `recvbuf` of all the processes contains the reduced value
- Note: no `target` for this reduction is given

Collective Communication and Computation Operations: **Prefix (`MPI_Scan`)**

- Recall 4.3 - for Prefix-Sum - After the operation, every process has the sum of the buffers of the previous processes and its own.
- `MPI_Scan()` is the 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: **`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_scatter**

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

Collective Communication and Computation Operations: **MPI_Alltoall**

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

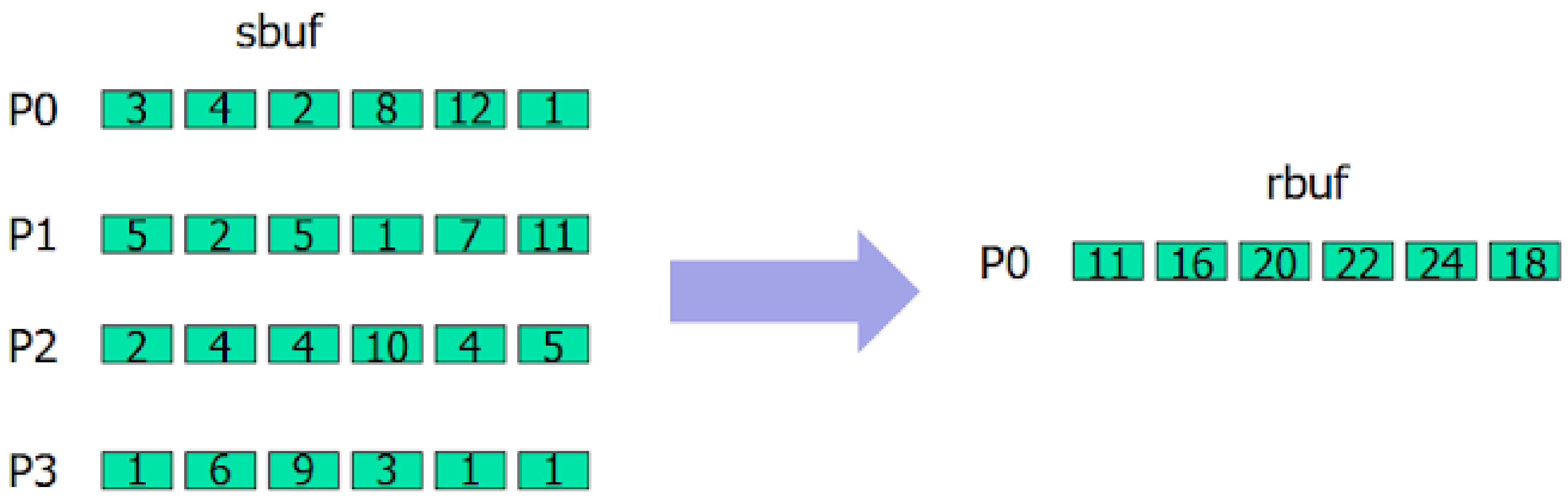
```
int MPI_Alltoall(void *sendbuf, int sendcount, MPI_Datatype senddatatype, void
  *recvbuf, int recvcount, MPI_Datatype recvdatatype, MPI_Comm comm)
```

- 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 all-to-all personalized communication**

Examples (for practice)

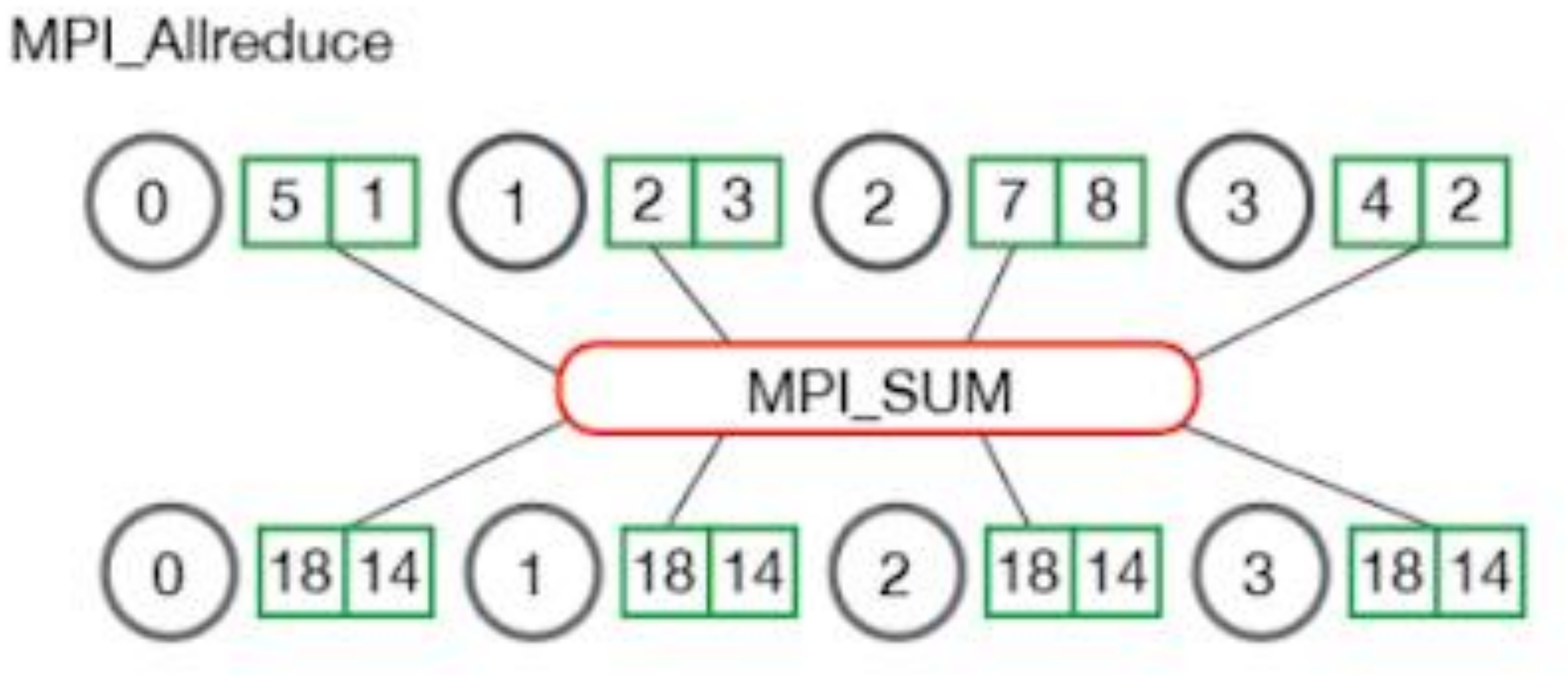
MPI_Reduce example

MPI_Reduce(sbuf,rbuf,6,MPI_INT,MPI_SUM,0,MPI_COMM_WORLD)



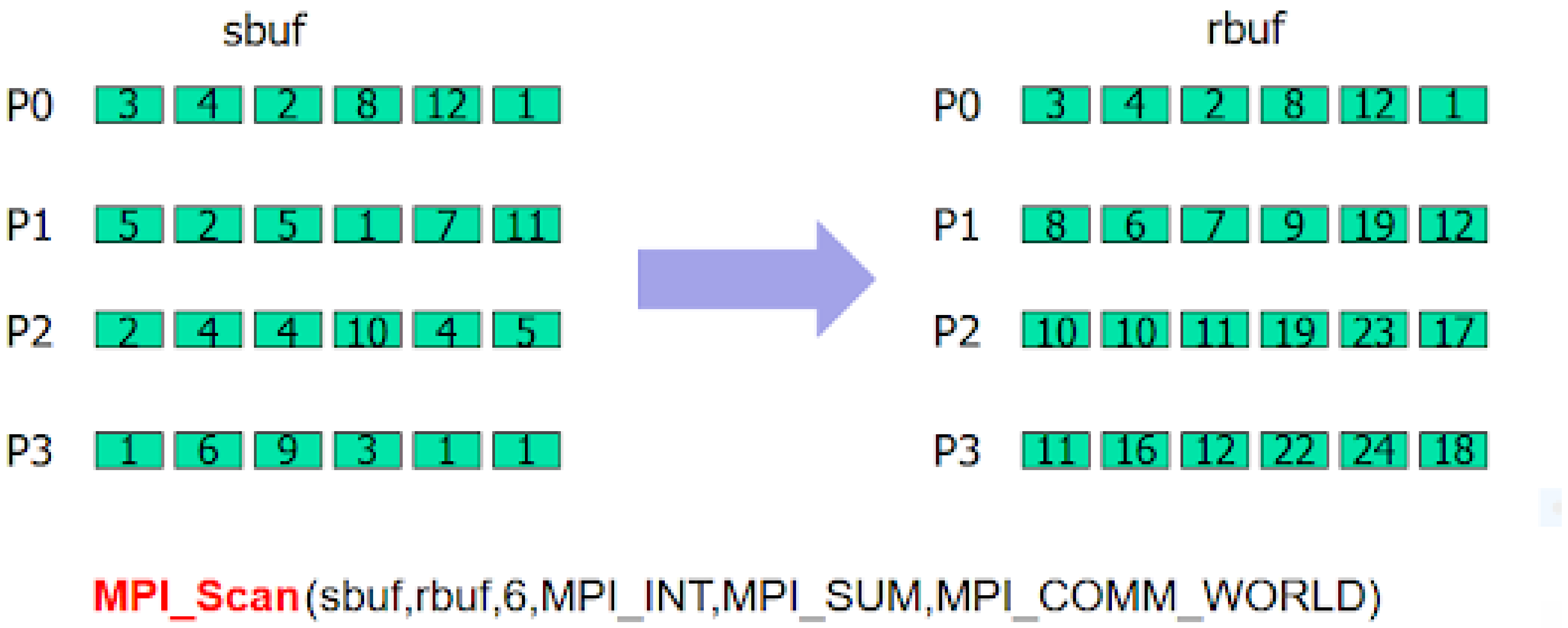
Source: https://www.sachinpbuzz.com/2020/12/overview-of-mpi-reduction-operations.html

MPI_Allreduce example



Source: https://www.sachinpbuzz.com/2020/12/overview-of-mpi-reduction-operations.html

MPI_Scan example



Source: https://www.sachinpbuzz.com/2020/12/overview-of-mpi-reduction-operations.html

References

1. Slides of Dr. Rana Asif Rehman & Dr. Haroon Mahmood
2. Kumar, V., Grama, A., Gupta, A., & Karypis, G. (1994). *Introduction to parallel computing* (Vol. 110). Redwood City, CA: Benjamin/Cummings.
3. Quinn, M. J. Parallel Programming in C with MPI and OpenMP, (2003).

Helpful Links:

1. https://mpitutorial.com/tutorials/mpi-send-and-receive/
2. https://mpitutorial.com/tutorials/mpi-scatter-gather-and-allgather/
3. (odd-even sorting, next topic) http://boron.physics.metu.edu.tr/ozdogan/GraduateParallelComputing.old/week11/node2.html