Zain Al Abidin 21L-6260

Q#1

MPI-Send() does not have the 'Status' parameter, which provides us with the information regarding the received message. It may include the source process, tag or the error code. It is important in circumstances where we are dealing with non-blocking receives. This allows us to ensure that the message has been received correctly. Also, in circumstances where we need to process the received message differently based on it's source or tag, the Status parameter becomes essential for determing how to handle the received data.

Q#2

We would need to add a destination identifier, meaning that we need to include an addition field in the message to specify which processor we are sending the message to. This way we would save computational cost and only send messages to the target processors. We would achieve this by using an if...else statement and if the current processor matches the destination processo in the message, the we would proceed

```
procedure SCATTER (d, my-id, X, dest_id)
begin
        mask := $2^d - 1$
        for i := d-1 downto 0 do
            mask := mask XOR $2^i$
            if (my-id AND mask) = 0 then
                if (my-id == dest_id) then
                    send X to msg_destination.

            else
                if (my_id == msg_source) then
                    receive X from msg_source

        endfor
end SCATTER
```

[illegible]

Q#2

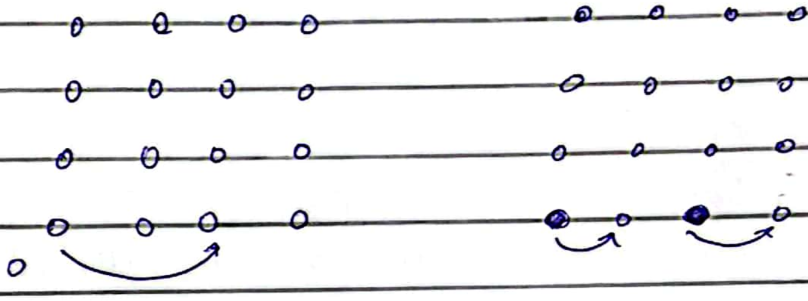
I'm 0: Received: 3 from 3 and Sent: 3 to 1.
I'm 1: Received: 0 from 0 and sent: 0 to 2.
I'm 2: Received: 1 from 1 and sent: 1 to 3
I'm 3: Received: 2 from 2 and sent: 2 to 0.

Q#1

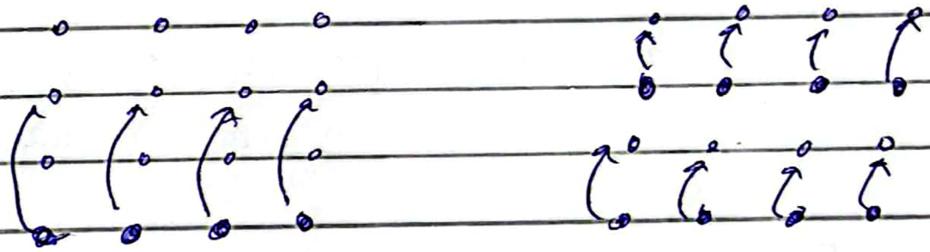
Initially, the message will be sent from node 0 to it's immediate neighbours. In the next step, the message the message will travel Twice as far from the source node. This phase will take $\log(n)$ steps where n is the number of nodes in the mesh.

In the next phase, each node starts sending messages vertically in the same way therefore the cost for this phase will also be $\log(n)$.

Phase 1.



Phase2



Therefore, the total cost would be $\log(n) + \log n$

$\Rightarrow 2\log(n)$

$\Rightarrow \underline{\log n}.$

S#5

MPI Collective Operations

MPI helps us implement collective communication and computational operations. It's called collective because all the processes must participate in the communication.

Communicator

MPI_COMM_WORLD is used to define a group of processes that participate in communications and performing collective operations.

Barrier Synchronization

refers to a synchronization mechanism where multiple processes or threads must wait until all of them have reached a certain point in their execution before any of them can proceed any further.

One to All Broad Cast

broadcast data from source process to all other processes in the communicator.

All to One Reduction

is a useful operation where data from multiple processes need to be combined into a single result. Possible use Case is computing global sum, finding global ~~sum~~ max and min or performing similar operations across distributed data sets.

All Reduce

performs a reduction like MPI_Reduce but ensures that the result is available to processes.

MPI_Scan

computes the prefix operation on data, where each process received the partial reduction result up to it's own data.

MPI_ExScan

computes the exclusive prefix on data excluding each process own contribution and store the result in receiver buffer.

MPI_Gather

MPI_Gather collects data from all processes onto a single target process.

MPI_Gatherv

Gathers data from all processes onto a single target process. while allowing each message to have different lengths.

MPI_AllGather

gathers data from all processes and distributes the gathered data to all the processes in the communicator.

MPI_Scatter

Scatters data from one process to all processes in the Communica

MPI_ScatterV

Same as scatter except it allows messages to have different lengths.

MPI_All to All

is a collective communication process that exchanges data between all processes in a communicator. Each process sends distinct data to all other processes and each process receives distinct data from all processes.

Q#5

1) MPI_Init ⇒ initializes the MPI environment.

int MPI_Init (int * argc, char ** argv).

2) MPI_Finalize ⇒ ends the MPI environment.

int MPI_Finalize()

3) MPI-Comm_rank ⇒ determines the rank of the calling process in the communicator.

int MPI-Comm_rank (MPI_Comm comm, int *rank)

4) int MPI_Comm_size (MPI-comm comm, int * size)

⇒ determines the size of the group associated with a communicator.

5) int MPI_Send(void * buf, int count, MPI_Datatype dtype, int dest,
int tag, MPI_Comm comm).

⇒ used for sending a message from one process to another.

6) int MPI_Recv(void *buf, int count, MPI_Datatype dtype, int source,
int tag, MPI_Comm Comm, MPI_Status * stat)

⇒ used ~~for~~ for receiving messages sent by another process.

7) int MPI_Bcast (void * buf, int count, MPI_Datatype dtype, int root,
MPI_Comm comm).

⇒ broadcasts a message from one process to all the processes in the communicator.

8) int MPI_Barrier (MPI_Comm comm).

⇒ blocks until all the processes in the communicator have reached this routine

9) int MPI_Scatter(void * sendbuf, int sendcount, ~~MPI_Datatype dtype~~,
MPI_Datatype send_dtype, void * recv buf, int recvcount,
MPI_Datatype recv_dtype, int root, MPI_Comm comm).

⇒ Scatters data from one process to all processes in a communicator.

10) int MPI_Gather(// same parameters and syntax as Scatter)

⇒ gathers data from all processes in a communicator to one process.

11) int MPI_Reduce (void * sendbuf, void * recvbuf, int count,
MPI_Datatype dtype, MPI_Op op, int root, MPI_Comm_comm

⇒ performs a reduction operation (eg sum, max, min) across all processes in a communicator.

12) int MPI-Allreduce (void * sendbuf, void * recvbuf, int count,
MPI_Datatype dtype, MPI_Op op, MPI_Comm comm)
⇒ combines values from all processes in a communicator and
distributes the result back to all processes.

13) MPI_Alltoall (void * sendbuf, int sendcount, MPI_Datatype send dtype,
void * recvbuf, int recvcount, MPI_Datatype recv_dtype,
MPI_Comm comm).
⇒ sends data from all processes to all processes in a communicator.

14) int MPI_Wait (MPI_Request * request, MPI_Status * status).
⇒ waits for an asynchronous operation to complete.

15) int MPI_Test (MPI_Request * request, int * flag, MPI_Stautus * stat).
⇒ tests for the completion of an asynchronous operation.

16) int MPI_Irecv (void * buf, int count, MPI-Datatype dtype, int source,
int tag, MPI_Comm comm, MPI_Request * request).
⇒ Initiates a non-blocking receive operation.

17) int MPI_Isend (void * buf, int count, MPI_Datatype dtype, int dest,
int tag, MPI_Comm Comm, MPI_Request * request).
⇒ initiates a non-blocking send operation.

18) int MPI_Comm_split (MPI_Comm comm, int color, int key,
MPI_Comm * newcomm).
⇒ splits an existing communicator into multiple
new communicators.

19) int MPI_Scan (void * sendbuf, void * recvbuf, int count,
MPI_Datatype dtype, MPI_Op op, MPI_Comm comm).
⇒ computes prefix reductions on data distributed
across processes.

20) int MPI-Exscan(void * sendbuf, void * recvbuf, int count,
MPI_Datatype dtype, MPI_Op op, MPI_Comm comm).

⇒ Computes exclusive-prefix reductions on data distributed across processes

21) int MPI_Gatherv (void * sendbuf, int sendcount, MPI_Datatype send_dtype,
void * recvbuf, int * recvcounts, int * displs
MPI_Datatype recv_dtyp, int root, MPI_Comm comm).

⇒ gathers data from all processes and stores it in the Receive buffer on the root process. Each process can contribute a different amount of data, hence why we use this.

22) int MPI_Allgather (void * sendbuf, int sendcount, MPI_Datatype
send_dtype, void * recv buf, int recvcount,
MPI_Datatype recvdtype, MPI_Comm Comm)

⇒ gathers data from all processes and distributes it to all processes in the communicator.

23) int MPI_Allgather v (void * sendbuf, int sendcount,
MPI_Datatype send_dtype, void * recv buf,
int recvcount, int * displs, MPI_Datatype recvdtype
MPI_Comm Comm).

⇒ Same as Allgather except for the fact that each process Can contribute a different amount of data.

24) int MPI_Scatter v (void * sendbuf, int * sendcount, int * displs,
MPI_Datatype send_dtype, void * recvbuf, int recvcount
MPI_Datatype recv_type, int root,
MPI_Comm Comm).

→ distributes data from the source/root process to all processes in the communicator, allowing each process to receive a different number of elements