Recitation 5: MPI

Team:

- Manuel O. Maldonado Figueroa (mom32)
- Jumana Dakka (jdakka)

Exercises:

There are a total of five exercises on this recitation.

EX1

After initializing MPI, each process gets its rank id by calling the `MPI_Comm_rank` function and the total number of MPI processes by calling the `MPI_Comm_size`. For each process to print a message with its rank id and total number of MPI processes, we simply use the following line:

```
printf("hello world, I am task %d from total %d\n", me, nprocs );
```

where `me` has the processe's rank id and `nprocs` has the total number of processes.

EX2.1

Deadlock is encountered when process 0 tries to send and receive data from another process. All of the other processes simply call MPI_Recv (receiving from their neighboring lower process) and then MPI_Send (sending the value to their neighboring higher process) to pass the value along the ring. MPI_Send and MPI_Recv will block until the message has been transmitted. We fix this by allowing all other processes to finish receiving messages and finally permitting process 0 to receive its message from the last process.

EX2.2

MPI_Isend and MPI_Irecv are non-blocking, meaning that the function call returns before the communication is completed. Deadlock then becomes impossible with non-blocking communication, but other precautions must be taken in order to ensure that data is received properly. Placing an MPI_Wait call for each send and receive call ensures that the process receives/sends before advancing the program. The MPI_Request will stall until MPI_Wait call is completed.

EX3

The `MPI_Scatter` and `MPI_Gather` functions are used to evenly distribute elements in an array among multiple MPI processes. Each process gets `num_elem` elements from the main array (as sent by the root element) in a buffer that each process creates. The problem comes when the number of elements in the original array is not divisible by the number of MPI processes available, if this happens some elements in the array will not be distributed. To overcome this, we used `MPI_Scatterv` and `MPI_Gatherv`. Each one of these 'vector' functions take two new pointers; a pointer to an array that contains the offset of the original array that each one of the MPI processes should receive; and the count of elements each MPI process should receive starting from the offset specified. These functions allow for a multitude of ways to slice and dice an array of data among MPI processes, in our case, we just first divide the array evenly (i.e. give each process `N / total_ranks` where `N` is the total number of elements in the array) then if there is any elemnts left over, easily checked by a modulus (i.e. `%`) operation, we add an extra element to each MPI process starting with process `0`.

EX4

For this exercise we used `MPI_Scatterv` instead of `MPI_Scatter` to overcome the issue seen on EX3 (i.e. the number of elements is not easily divisible by the total number of MPI processes). Each process calculates the total sum over each element they receive and then the `MPI_Reduce` function is used to gather all local sums into a global sum.