CSCI351 Project 1: Communication

Joseph Carluccio, Yang Hong, Andrew Passero, Warren Wang

February 18, 2023

1 Problem 1

We implement two algorithms for this problem – broadcast and convergecast – from the pseudocode given in lecture notes.

For broadcast, converting from pseudocode to python code was fortunately not too bad. The only problem we encountered was initially trying to use blocking sends and receives, but switching over to non-blocking sends and receives fixed our program hanging issues.

For convergecast, we used our knowledge from writing broadcast to avoid problems with fighting with the MPI library. Figuring out what to loop forever with specific exit conditions was the most difficult part in the translation.

2 Problem 2

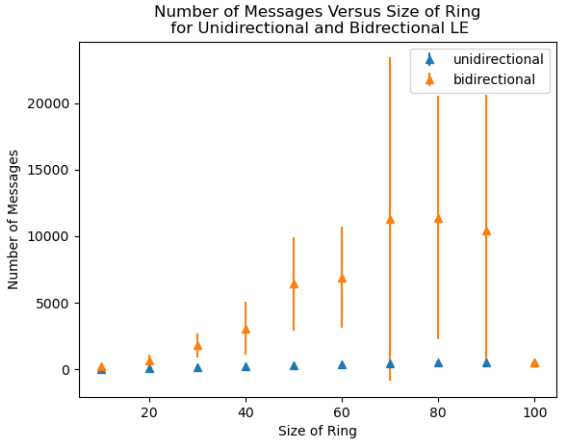
Problem two required the implementation of a unidirectional and bidirectional algorithm for solving leader election in rings.

The implementation for the unidirectional problem was fairly straight forward but some challenges did arise in understanding how to use the mpi python library. Additionally, debugging can be a challenging in a distributed system where nothing seems to print in the order in which it happened. Another challenge we faced was collecting the total number of messages. This was a difficult because different processes are terminating at different times making it hard to collect totals from every process. The solution we came up with was adding an extra process to every execution that did nothing but receive message totals from processes before they terminated. So if our graphs specified 10 processes, 11 would be the command line argument for mpi4py and the extra node would not participate in the algorithm. We understand there are probably more eloquent solutions to this problem by better utilizing the tools in the library.

For the bidirectional algorithm, we built off of the unidirectional algorithm algorithm and instead send messages with multiple variables needed to efficiently pass information around in both left and right directions. However, we ran into

much trouble in the implementation details. While the fundamental LE problem is solved, we are counting an unexpectedly large number of messages being passed from process to process in the ring. While we may expect slightly more messages than the unidirectional algorithm for small number of processes, our tests do not scale as expected, with the bidirectional using much, much more messages than we expect compared to the unidirectional algorithm for larger rings.

To test the difference in the total messages sent per execution of each of the algorithms above, we wrote a script to first randomly generate rings of size n, and then another script to simulate the algorithms on different rings. The first script simply takes a size n and a random ring is generated and the adjacency list for that graph is saved to a text file. The second script loops through a range of ring sizes running the unidirectional and bidirectional LE algorithms for 10 different randomly gernated rings of that size. The results are collected for each execution and the mean and STD are calculated and plotted. The results for the average number of messages for rings of different size n are shown below.



Overall, it is clear that the bidirectional algorithm has a lower message complexity than the unidirectional algorithm. As n increases, this difference in total number of message becomes more apparent.

3 Problem 3

The bidirectional leader election algorithm fails for a tree of three processes p_0, p_1, p_2. If p_0 is the root of the tree with both p_1 and p_2 as children, the execution will terminate with both p_1 and p_2 terminating as leader. This occurs because to start the execution, each process will send its id to all of its neighbors. In round 0, p_1 and p_2 both send there id to p_0. Upon receiving both IDs, p_0 determines

that because this is round zero, the message needs to travel a distance $2^0 = 1$ before sending a reply. In this case, the distance for the received messages is already 1 and p_0 will send a reply to p_1 and p_2 respectively containing each processes own ID. p_1 and p_2 will receive their own ID and both terminate as leader.

4 Problem 4

On the basis of our convergecast program from Problem 1, we accept an extra argument for the function used for comparison. We use the eval function to allow that argument to be considered a valid Python function. Therefore by replacing the original max function with that argument function we succeed in fulfilling the requirement.

One tricky point is dealing with cases in which an inappropriate or invalid function is used in the argument. In order to detect that, we force the program to run a dummy test by inputting [3,4,5] into the function and see if it can return a non-Null output. By doing that, we can confirm if the argument function is able to make the comparison of an array of data and return something useful. If the function is determined as inappropriate then the process will directly exit with an error message.