Justin Burzachiello
10/20/2024
AMS 530

Project 2 Report

Project Description

This project entails comparing the time it takes for an `MPI_Allgather()` operation to complete on two different network topologies. In particular, the student must write custom code to implement leverage parallel broadcasting to transfer N different floating point numbers, where N is an element of {10, 20, 30, 40}, from each of 32 nodes to the other nodes using `MPI_Allgather()`. Two different custom network topologies must be created. Each of these is a ring network topology with 32 nodes, but with shortcuts added in different ways, as illustrated in the following figure, Figure 1. The first network is called Bid32 and it has shortcuts that, when the graph drawing represents a circle, connect nodes to those nodes on the other side of the circle. Likewise, the other network, called Opt32, has shortcuts, but these shortcuts usually do not connect a node to the node directly across the circular graph drawing. The MPI function `MPI_Graph_create()` is used to create both the Bid32 and Opt32 network topologies / communicators. Additionally, the timing of `MPI_Allgather()` on the different network topologies with different amounts of sent data is done and the results are collected into a table. This project highlights the utility of `MPI_Allgather()` and how its performance changes depending on the network topology being used.

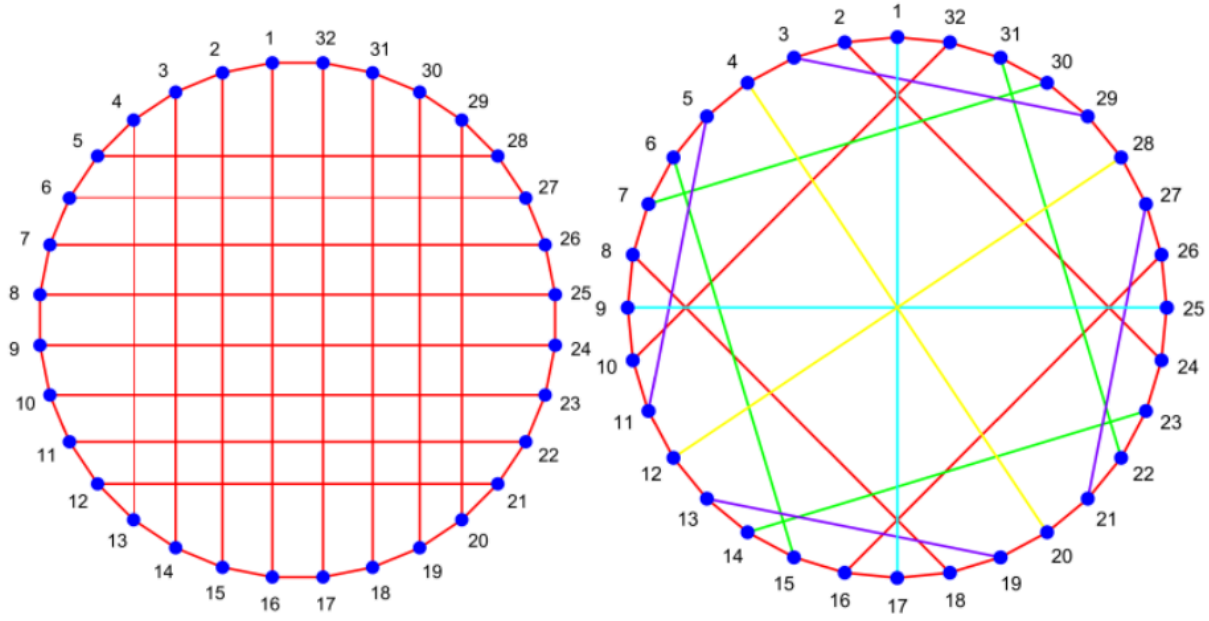

Figure 1: Bid32 (left); Opt32 (right)

Description of My Solution

My code uses MPI (Message Passing Interface) and constructs the two graph topologies illustrated in the project instructions. The program begins by initializing the MPI environment and determining the size of the process pool and the rank of each process. It checks that exactly 32 processes are used, as required by the network topology in the project. The number 32 is required since there are 32 nodes in both Bid32 and Opt32. If the number of processes is incorrect then the program terminates early. Next, a graph communicator is created using `MPI_Graph_create()`, which establishes the vertex connections that define how the data will flow between the processes. This code leverages a for loop to iterate through the two different network topologies. Once a particular network topology is chosen, the code then incorporates another for loop to iterate through the different values of N.

Once the corresponding network topology and value of N is chosen, a vector of data to send out, a different vector of data to read in, and a graph communicator are all created. The vector of data to send out has size N while the vector of data to read in has size N * M, where M = 32. Next, `MPI_Allgather()` is called with these vectors and the communicator. The length of time it takes for `MPI_Allgather()` to complete is computed with `MPI_Wtime()`.

Finally, I have the rank 0 node output the length of time it took `MPI_Allgather()` to complete. This output happens for both network topologies and all values of N. Additionally, my submitted code contains commented-out code to print out the values received at each node from the MPI_Allgather() for each network topology and all values of N. This outputs a lot of text, hence why it is commented out. However, the grader can uncomment it to confirm the validity of my code's output.

Results

The results presented in the table compare the performance of `MPI_Allgather()` on two different ring network topologies with shortcuts, all with 32 nodes. Implementing these network topologies, called Bid32 and Opt32, and transferring data among the respective nodes was completed using `MPI_Graph_create()` and `MPI_Allgather()`. Timing results are conveyed in the following table, Table 1.

	On Bid32 (s)	On Opt32 (s)
N = 10	0.000034	0.000007
N = 20	0.000579	0.000008
N = 30	0.000020	0.000031
N = 40	0.000056	0.000011

Table 1: Timing results for Bid32 and Opt32 with each value of N.

The data indicate that `MPI_Allgather()` ran faster on Opt32 for all values of N except for N = 30. Additionally, the time for `MPI_Allgather()` to complete does not increase monotonically with the value of N. For instance, the runtime on Bid32 increases drastically N = 10 to N = 20 but then decreases drastically to a value lower than that of N = 10 for N = 30. However, due to the small number of processors used, this behavior is reasonable. Notably, the time it takes for `MPI_Allgather()` to complete changes from run to run.

These results suggest that using the Opt32 is more efficient for `MPI_Allgather()` compared with Bid32 for datasets of most, but not all, sizes. The differences between them may also be more extreme with larger numbers of processors. It is likely true that shortcuts are, on average, more feasible and accessible for transferring data amongst nodes on the Opt32 network topology compared with the Bid32 network topology. Finally, I suspect but cannot confirm that the reason why Opt32 tends to perform so much better than Bid32 in this test is related to the additional degrees of rotational symmetry in its graph drawing, but I cannot confirm.

The following figure, Figure 2, shows the data stored on each node for sending out to the other nodes when N = 10. Note that these data being sent out have a higher numerical precision than what is displayed.

```
Data (node 0): 0.08 0.04 0.08 0.08 0.09 0.02 0.03 0.08 0.03 0.06
Data (node 1): 1.08 1.04 1.08 1.08 1.09 1.02 1.03 1.08 1.03 1.06
Data (node 2): 2.07 2.08 2.01 2.01 2.03 2.04 2.07 2.01 2.06 2.06
Data (node 3): 3.06 3.02 3.04 3.04 3.03 3.01 3.06 3.09 3.09 3.02
Data (node 4): 4.09 4.01 4.02 4.03 4.02 4.04 4.04 4.07 4.02 4.03
Data (node 5): 5.03 5.00 5.10 5.01 5.01 5.01 5.03 5.04 5.05 5.09
Data (node 6): 6.01 6.10 6.08 6.04 6.01 6.08 6.06 6.02 6.03 6.10
Data (node 7): 7.05 7.09 7.06 7.02 7.00 7.05 7.10 7.00 7.06 7.01
Data (node 8): 8.04 8.08 8.09 8.00 8.10 8.02 8.09 8.08 8.09 8.07
Data (node 9): 9.02 9.07 9.02 9.04 9.09 9.05 9.07 9.06 9.07 9.07
Data (node 10): 10.06 10.06 10.05 10.02 10.08 10.02 10.06 10.04 10.00 10.03
Data (node 11): 11.09 11.05 11.03 11.05 11.08 11.04 11.04 11.02 11.08 11.04
Data (node 12): 12.08 12.04 12.01 12.08 12.07 12.01 12.03 12.05 12.06 12.05
Data (node 13): 13.06 13.03 13.09 13.01 13.06 13.03 13.02 13.08 13.09 13.06
Data (node 14): 14.10 14.03 14.02 14.05 14.01 14.06 14.00 14.06 14.07 14.07
Data (node 15): 15.04 15.07 15.05 15.03 15.10 15.08 15.04 15.04 15.00 15.02
Data (node 16): 16.02 16.01 16.03 16.06 16.09 16.00 16.02 16.07 16.04 16.08
Data (node 17): 17.06 17.00 17.01 17.09 17.09 17.02 17.01 17.05 17.07 17.09
Data (node 18): 18.09 18.09 18.09 18.02 18.08 18.09 18.10 18.03 18.05 18.05
Data (node 19): 19.03 19.03 19.02 19.01 19.02 19.07 19.03 19.06 19.03 19.01
Data (node 20): 20.01 20.07 20.05 20.04 20.07 20.04 20.07 20.04 20.06 20.07
Data (node 21): 21.05 21.02 21.03 21.07 21.01 21.06 21.01 21.07 21.09 21.02
Data (node 22): 22.09 22.06 22.06 22.00 22.05 22.08 22.04 22.05 22.02 22.03
Data (node 23): 23.07 23.05 23.09 23.03 23.05 23.06 23.03 23.03 23.05 23.09
Data (node 24): 24.01 24.09 24.02 24.07 24.09 24.03 24.06 24.01 24.08 24.05
Data (node 25): 25.09 25.08 25.00 25.10 25.03 25.10 25.10 25.08 25.01 25.06
Data (node 26): 26.08 26.02 26.03 26.03 26.03 26.07 26.09 26.06 26.04 26.07
Data (node 27): 27.07 27.01 27.02 27.01 27.07 27.04 27.07 27.04 27.07 27.03
Data (node 28): 28.05 28.00 28.05 28.04 28.01 28.02 28.01 28.07 28.00 28.03
Data (node 29): 29.09 29.05 29.08 29.03 29.06 29.04 29.04 29.05 29.08 29.09
Data (node 30): 30.02 30.09 30.06 30.06 30.10 30.06 30.08 30.08 30.06 30.05
Data (node 31): 31.06 31.08 31.09 31.04 31.04 31.03 31.02 31.06 31.04 31.06
```

Figure 2: Data to-be-sent-out to other nodes

Here is a figure showing the data stored on the process with rank 0 with N = 10 after calling `MPI_Allgather()`.

```
Data (node 0): 0.08 0.04 0.08 0.08 0.09 0.02 0.03 0.08 0.03 0.06 1.08 1.04 1.08 1.08 1.09 1.02 1.03 1.08 1.03 1.06 2.07 2.08 2.01
 2.01 2.03 2.04 2.07 2.01 2.06 2.06 3.06 3.02 3.04 3.04 3.03 3.01 3.06 3.09 3.09 3.02 4.09 4.01 4.02 4.03 4.02 4.04 4.04 4.07 4.0
2 4.03 5.03 5.00 5.10 5.01 5.01 5.01 5.03 5.04 5.05 5.09 6.01 6.10 6.08 6.04 6.01 6.08 6.06 6.02 6.03 6.10 7.05 7.09 7.06 7.02 7.
00 7.05 7.10 7.00 7.06 7.01 8.04 8.08 8.09 8.00 8.10 8.02 8.09 8.08 8.09 8.07 9.02 9.07 9.02 9.04 9.09 9.05 9.07 9.06 9.07 9.07 1
0.06 10.06 10.05 10.02 10.08 10.02 10.06 10.04 10.00 10.03 11.09 11.05 11.03 11.05 11.08 11.04 11.04 11.02 11.08 11.04 12.08 12.0
4 12.01 12.08 12.07 12.01 12.03 12.05 12.06 12.05 13.06 13.03 13.09 13.01 13.06 13.03 13.02 13.08 13.09 13.06 14.10 14.03 14.02 1
4.05 14.01 14.06 14.00 14.06 14.07 14.07 15.04 15.07 15.05 15.03 15.10 15.08 15.04 15.04 15.00 15.02 16.02 16.01 16.03 16.06 16.0
9 16.00 16.02 16.07 16.04 16.08 17.06 17.00 17.01 17.09 17.09 17.02 17.01 17.05 17.07 17.09 18.09 18.09 18.09 18.02 18.08 18.09 1
8.10 18.03 18.05 18.05 19.03 19.03 19.02 19.01 19.02 19.07 19.03 19.06 19.03 19.01 20.01 20.07 20.05 20.04 20.07 20.04 20.07 20.0
4 20.06 20.07 21.05 21.02 21.03 21.07 21.01 21.06 21.01 21.07 21.09 21.02 22.09 22.06 22.06 22.00 22.05 22.08 22.04 22.05 22.02 2
2.03 23.07 23.05 23.09 23.03 23.05 23.06 23.03 23.03 23.05 23.09 24.01 24.09 24.02 24.07 24.09 24.03 24.06 24.01 24.08 24.05 25.0
9 25.08 25.00 25.10 25.03 25.10 25.10 25.08 25.01 25.06 26.08 26.02 26.03 26.03 26.03 26.07 26.09 26.06 26.04 26.07 27.07 27.01 2
7.02 27.01 27.07 27.04 27.07 27.04 27.07 27.03 28.05 28.00 28.05 28.04 28.01 28.02 28.01 28.07 28.00 28.03 29.09 29.05 29.08 29.0
3 29.06 29.04 29.04 29.05 29.08 29.09 30.02 30.09 30.06 30.06 30.10 30.06 30.08 30.08 30.06 30.05 31.06 31.08 31.09 31.04 31.04 3
1.03 31.02 31.06 31.04 31.06
```

Figure 3: Data received by process with rank 0.