Justin Burzachiello
12/01/2024
AMS 530

Project 4 Report

**Project Description**

This project involves optimizing the parallel broadcast of floating-point numbers across an network of 32 vertices organized into two different unweighted, undirected network topologies. These networks are titled 4x8a and 4x8b, and are each composed of 8 graph “squares” which are then connected to one another in different ways between 4x8a and 4x8b. The 4 in 4x8 comes from the fact that we construct squares, which have 4 vertices; the 8 is because there are 8 of these squares in each network. These two networks are illustrated in Figure 1. The student must write custom code to implement parallel broadcasting of floats across the respective network topologies then compare the results of this custom implementation to that of the built-in MPI implementation for broadcasting data. In particular, basic, non-collective MPI (Message Passing Interface) functions, such as `MPI_Graph_create()`, `MPI_Isend()`, and `MPI_Irecv()`, should be used to create the network topology / communicator and send / receive messages between nodes connected according to this created topology. Comparing the performance of the custom algorithm to the built-in algorithm `MPI_Bcast()`is done by comparing the time taken to perform the broadcast with the respective algorithms for vectors of floating point numbers of size 10^6, 10^7, and 10^8 for both 4x8a and 4x8b. Results are collected into a table. This project highlights the importance of optimizing parallel communications in distributed computing environments and using performance metrics to evaluate different approaches. Additionally, adjacency matrices and distance matrices are computed for both 4x8a and 4x8b.

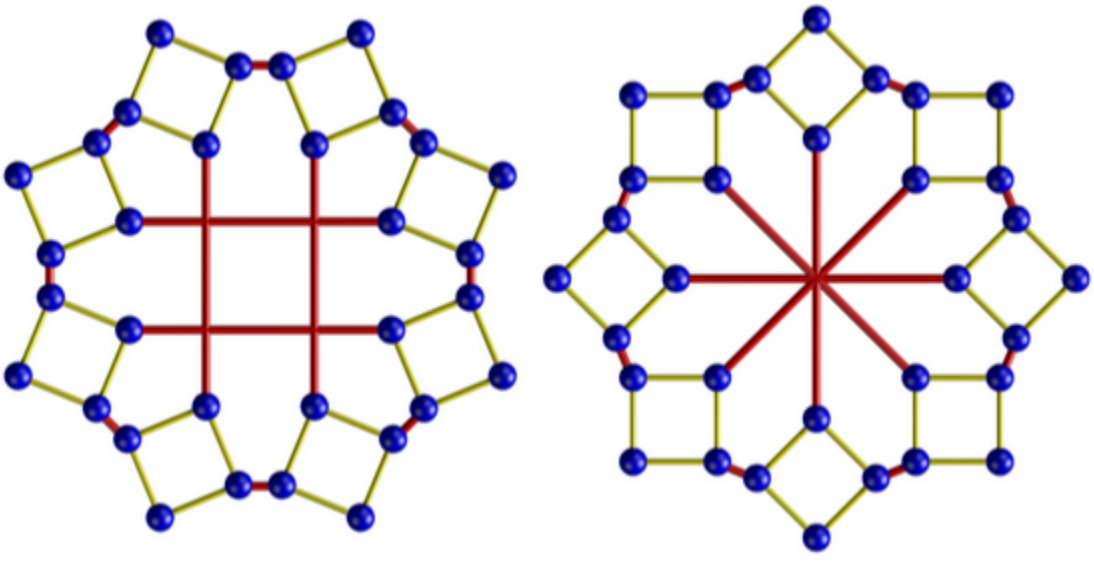

Figure 1: 4x8a (left) and 4x8b (right). The yellow edges are “intraconnections” that form squares while the red edges are “interconnections” that connect different squares. Note that each square is connected to 2 proximal squares and one distal square.

**Description of My Solution**
**Custom:**

The custom implementation of broadcast in this code uses MPI (Message Passing Interface) and mimics the respective network topologies of 4x8a and 4x8b which are illustrated in Figure 1. 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. This is required since each of the network topologies in the project have 32 vertices. If the number of processes is incorrect, the program terminates early. Next, a custom graph communicator is created using `MPI_Graph_create()`. This establishes the vertex connections that define how the data will flow between the processes.

The routes that data propagate through in parallel can be represented by a tree structure. The root node of this tree is where the data start from, and it is arbitrary. The project instructions for this assignment do not specify which node to begin the propagation from. Thus, I chose a node that is connected with an “interconnecting” edge. In Figures 2 and 3 I illustrate my labeling scheme for nodes. In my code, node 31 served as the root node for 4x8a while node 24 served as the root node for 4x8b. The labels for each node is arbitrary so long as connectivity is correctly quantified through the adjacency matrix. An adjacency list was initially created by hand for both 4x8a and 4x8b. This list was then used to create the C++ STL vectors of edges and indices that are used to create the respective graph communicators.

After constructing the graph communicators for both 4x8a and 4x8b, I initialized a vector of floating-point numbers that represents the data to be broadcast. Using non-blocking send operations, each node in the corresponding network topology first receives the data (using (`MPI_Irecv()`) then sends it off to its connected neighbors in the graph (using (`MPI_Isend()`). This process begins with the respective root node of the graph, and the tree data structures I illustrate in Figures 4 and 5 show how to propagate data through the respective network topologies with minimum redundant operations. This receive / send process ensures that the broadcast happens in a structured manner, mimicking the specified graph topology instead of relying on the collective `MPI_Bcast()` function. Most importantly, due to the tree structure that conveys which nodes should send / receive to / from which other nodes, the algorithm is designed so that vertices do not send data to any vertex that has already received it.

The non-blocking nature of the communication (`MPI_Isend()` and `MPI_Irecv()`) ensures that processes do not idle while waiting for messages. This allows for asynchronous propagation of data across the network. Importantly, the program uses the MPI function

`MPI_Wtime()` to measure the time taken to complete the broadcast. This is computed by measuring the time on the root node for the respective topology from after the graph communicator is created until the broadcast is complete. Thus, timing is only measured on the root node. Notably, `MPI_Barrier()` is used to synchronize the processes before and after the broadcast to ensure that timing is only measured (on the root node) after the entire broadcast is complete for all processes in the system. The custom broadcast implementation successfully distributes data across all 32 vertices in the graph while adhering to the specific constraints and connections outlined by the project.

**Built-in:**
This built-in solution to this broadcast problem uses the built-in MPI function `MPI_Bcast()` to handle the distribution of data across the 32 processes in the respective networks. The program begins by initializing the MPI environment and determining the rank and size of the processes within the communicator. Like the custom implementation, it checks to ensure that exactly 32 processes are used, in accordance with the problem's requirements. If the number of processes does not meet this specification, the program terminates early.

The root node (root process) of the respective network acts as the source of the data in the broadcast. A vector of data is filled with a constant float of 42.0 on the root node while all other processes prepare to receive this data by initializing a vector of the same size. The built-in `MPI_Bcast()` function is then called to broadcast the data from the root node to all other nodes within the network topology defined by `MPI_Graph_Create()`. This collective operation ensures that all 32 processes receive the same data efficiently, leveraging MPI's optimized internal algorithms for broadcasting.

The program also measures the time taken for the broadcast to complete by capturing the start and end times using `MPI_Wtime()`. After the broadcast finishes, the time is printed out by the root process of the respective network topology. This implementation simplifies the broadcast logic significantly, relying on MPI's internal optimizations to manage the communication efficiently across all processes, as opposed to the custom implementation which manually handles data propagation through non-blocking sends and receives. Notably, this built-in implementation requires far less code for the end-user since most of it is abstracted by `MPI_Bcast()`.

**Adjacency and Distance Matrices**

Adjacency matrices are created from the aforementioned adjacency lists. Distance matrices are then computed from the adjacency matrices using NetworkX's Floyd-Warshall algorithm.

**Results**
The results presented in the table compare the performance of a custom broadcast algorithm against MPI's built-in broadcast function, `MPI_Bcast()`, for three different values of N (where N

is the number of floats being broadcast): 10^6, 10^7, and 10^8. This is done for each network topology. This table is shown in Table 1. All values of time are in seconds.

| N | My_BCast() Time: 4x8a | My_BCast() Time: 4x8b | MPI_BCast() Time: 4x8a | MPI_BCast() Time: 4x8b |
|---|---|---|---|---|
| 10^6 | 0.00817 | 0.00863 | 0.00393 | 0.00358 |
| 10^7 | 0.07924 | 0.06951 | 0.03078 | 0.02979 |
| 10^8 | 0.76074 | 0.66768 | 0.32482 | 0.31496 |

Table 1: Timing results.

The data indicate that for all values of N, the built-in MPI broadcast function is significantly faster than the custom implementation. However, they are on the same order of magnitude. This suggests that the built-in `MPI_Bcast()` is highly optimized for large-scale broadcast operations, significantly outperforming the custom broadcast implementation. While the custom implementation may be useful for understanding the underlying mechanics of broadcasting in a specific network topology, the built-in methods are clearly superior in terms of efficiency and scalability for larger data sets.

The adjacency matrices and distance matrices are shown on the figures in the following pages.

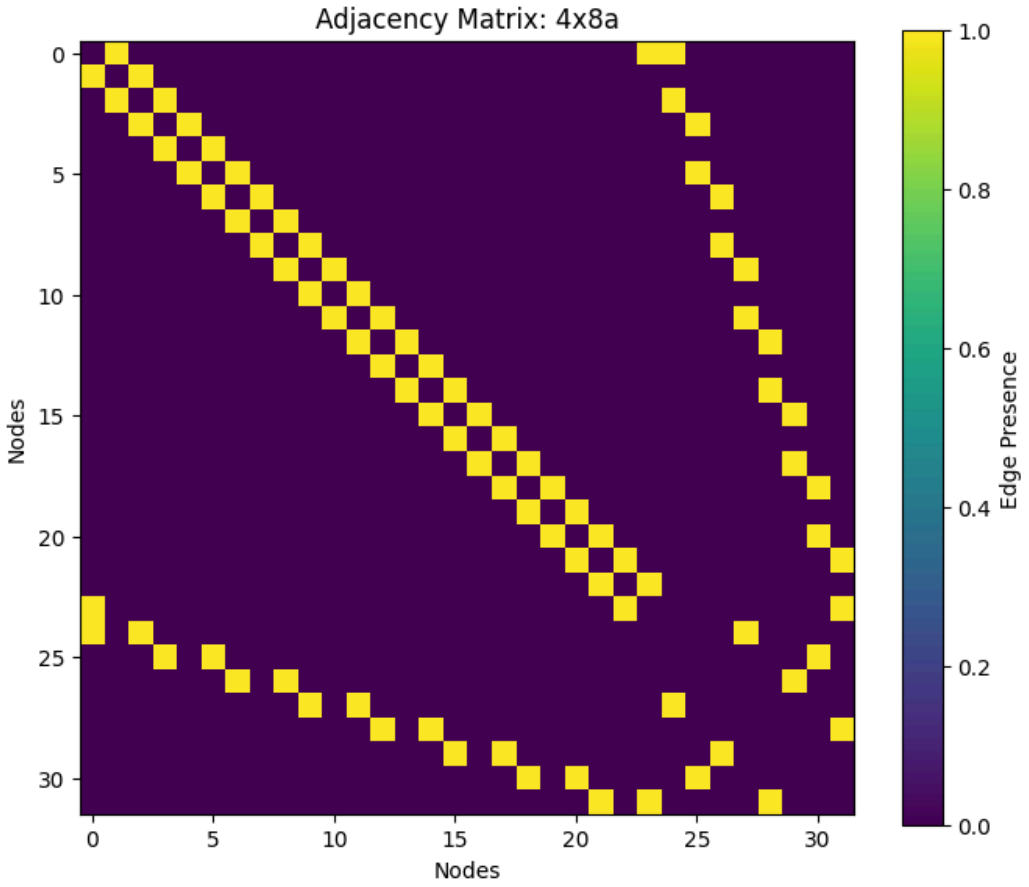


Figure 2: Adjacency matrix for 4x8a

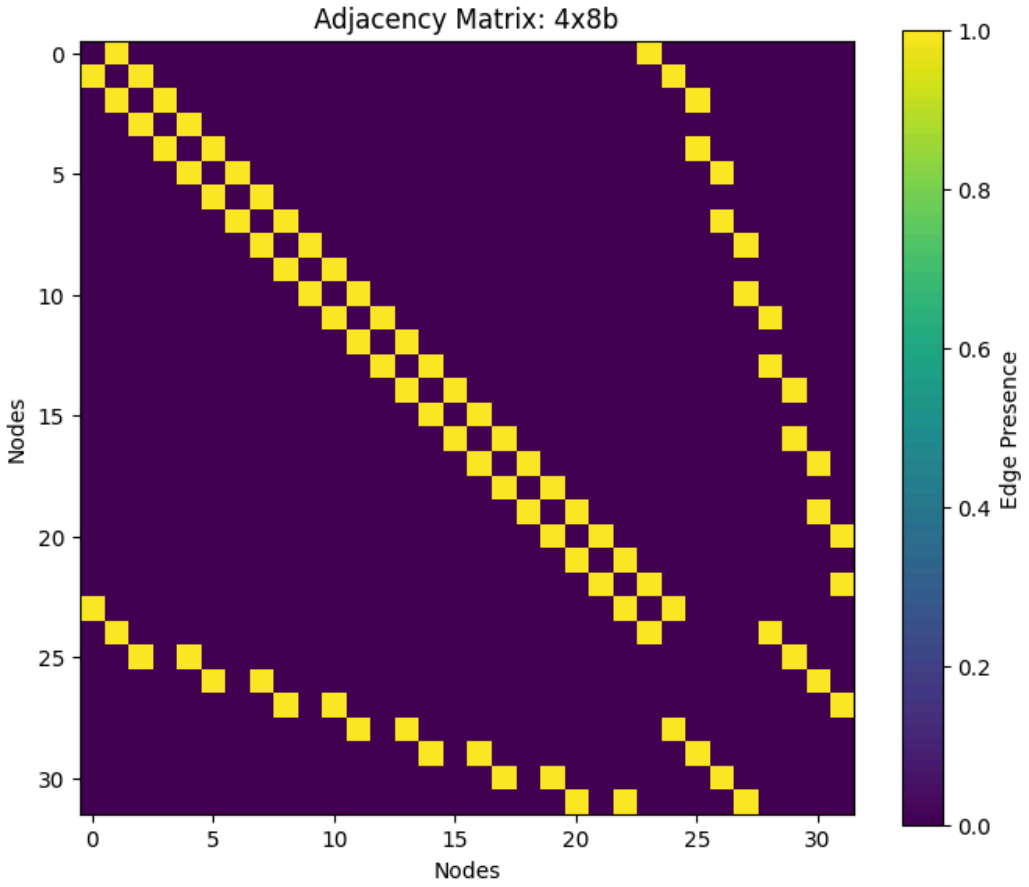


Figure 3: Adjacency matrix for 4x8b

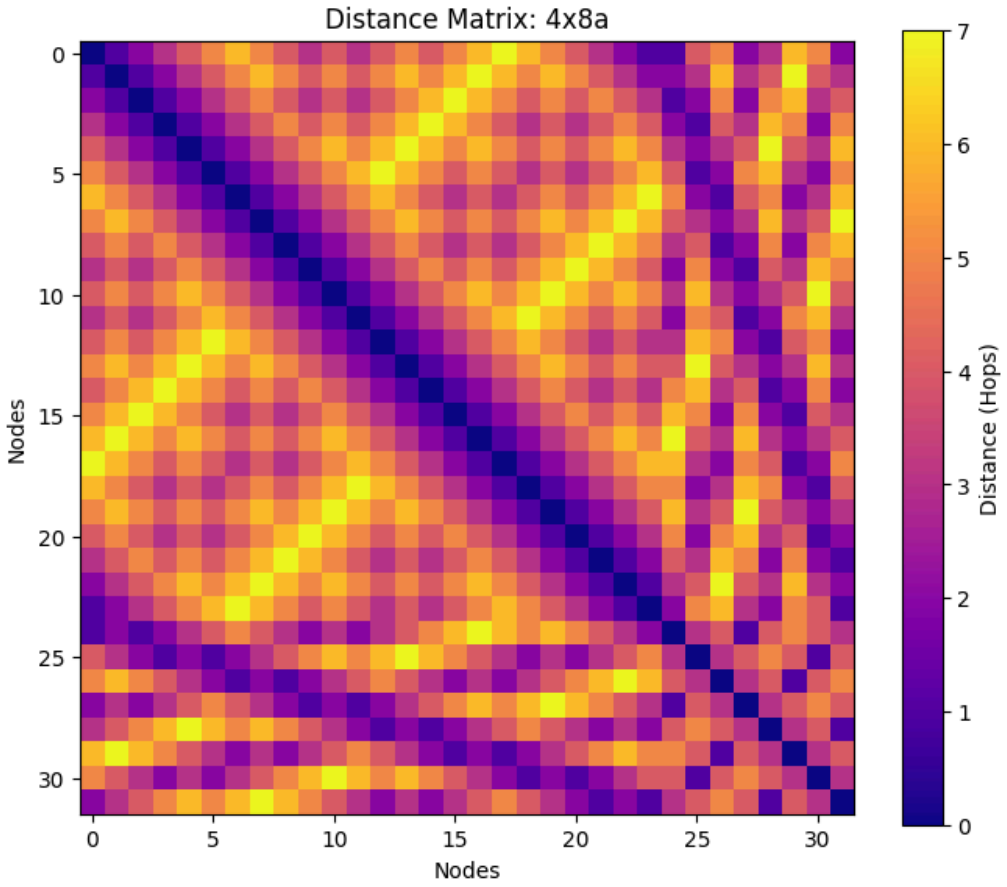


Figure 4: Distance matrix for 4x8a

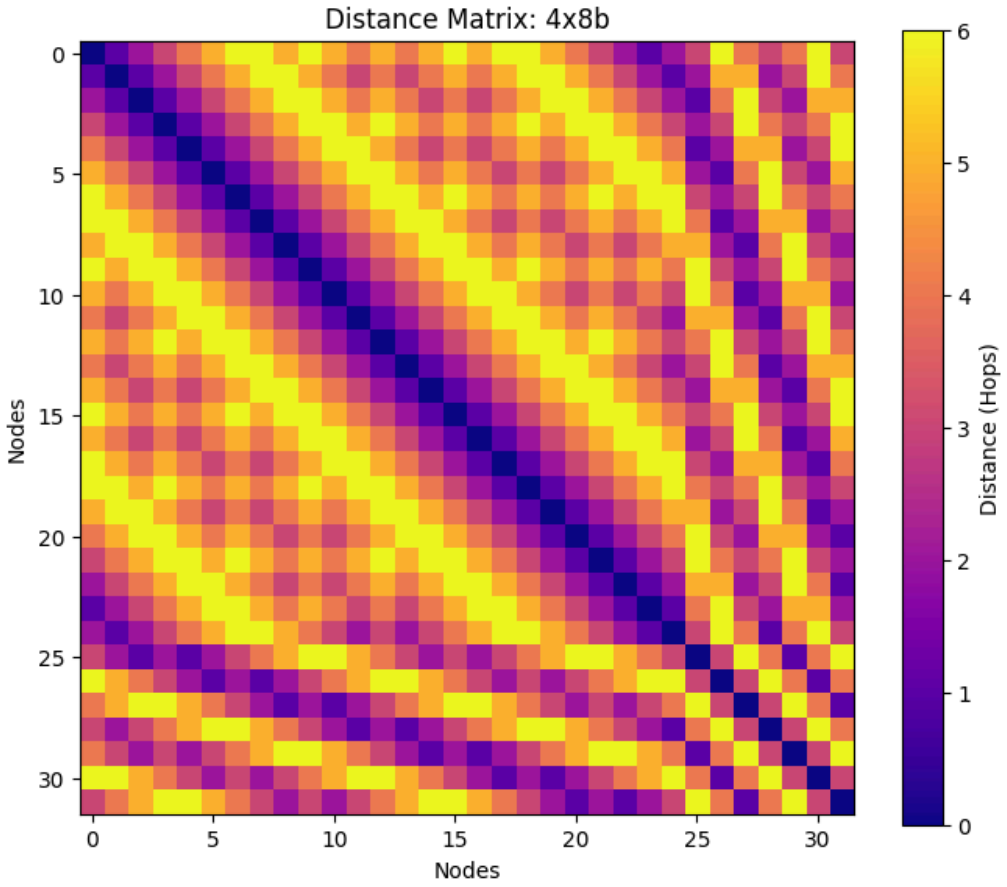


Figure 5: Distance matrix for 4x8b