Justin Burzachiello
09/29/2024
AMS 530

Project 1 Report

Project Description

This project entails optimizing the parallel broadcast of floating-point numbers across an network of 16 vertices organized into a particular network topology represented as an unweighted graph. In particular, the student must write custom code to implement parallel broadcasting of these floats across the network topology then compare the results of this custom implementation to that of the built-in MPI implementation for broadcasting data. The project recommends using basic, non-collective MPI (Message Passing Interface) functions, such as `MPI_Graph_create()`, `MPI_Isend()`, and `MPI_Irecv()` to create the network topology / communicator and send / receive messages between nodes connected according to this created topology. Additionally, 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. Results are collected into a table. The project highlights the importance of optimizing parallel communications in distributed computing environments and using performance metrics to evaluate different approaches.

Description of My Solution
Custom:

The custom implementation of the broadcast in this code uses MPI (Message Passing Interface) and mimics the graph topology from 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 16 processes are used, as required by the network topology in the project. If the number of processes is incorrect, the program terminates early. The main logic revolves around creating a custom graph communicator using `MPI_Graph_create()`, which establishes the vertex connections that define how the data will flow between the processes.

The core of the broadcast begins with the process of rank 0, which acts as the central vertex (C1) in the graph. This process initializes a vector of floating-point numbers, representing the data to be broadcast. Using non-blocking send operations (`MPI_Isend()`), the rank 0 process begins distributing chunks of this data to its connected neighbors in the graph. The neighbors, upon receiving data using non-blocking receive operations (`MPI_Irecv()`), further propagate the data to their own connected vertices. This process ensures that the broadcast happens in a structured manner, mimicking the specified graph topology instead of relying on the collective `MPI_Bcast()` function. Importantly, 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. The program also uses the MPI function `MPI_Wtime()` to measure the time taken to complete the broadcast, which is crucial for performance comparison against the standard `MPI_Bcast()` function. The custom broadcast implementation successfully distributes data across all 16 vertices in the graph while adhering to the specific constraints and connections outlined by the project.

Built-in:
This solution to the broadcast problem uses the built-in MPI function `MPI_Bcast()` to handle the distribution of data across the 16 processes in the network. 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 16 processes are used, in accordance with the problem's requirements. If the number of processes does not meet this specification, the program terminates early.

In this implementation, the process of rank 0, which acts as the source or central vertex (C1), initializes a vector of floating-point numbers with a fixed size. The data is filled with a constant value (42), 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 process (rank 0) to all other processes within the network topology defined by `MPI_Graph_Create()` that has a custom communicator. This collective operation ensures that all 16 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 process of rank 0. 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.

Results
The results presented in the table compare the performance of a custom broadcast algorithm (denoted as `My_BCast()` despite the fact that I did not create a custom function called `My_BCast()`) against MPI's built-in broadcast function (`MPI_Bcast()`) for three different values of N, specifically 10^6, 10^7, and 10^8 floating-point numbers. The column labeled "T1" represents the time taken by the custom implementation, while "T2" represents the time taken by the built-in MPI function. The final column shows the ratio of T1 to T2, which provides insight into the relative efficiency of the custom implementation in comparison to MPI's native solution.

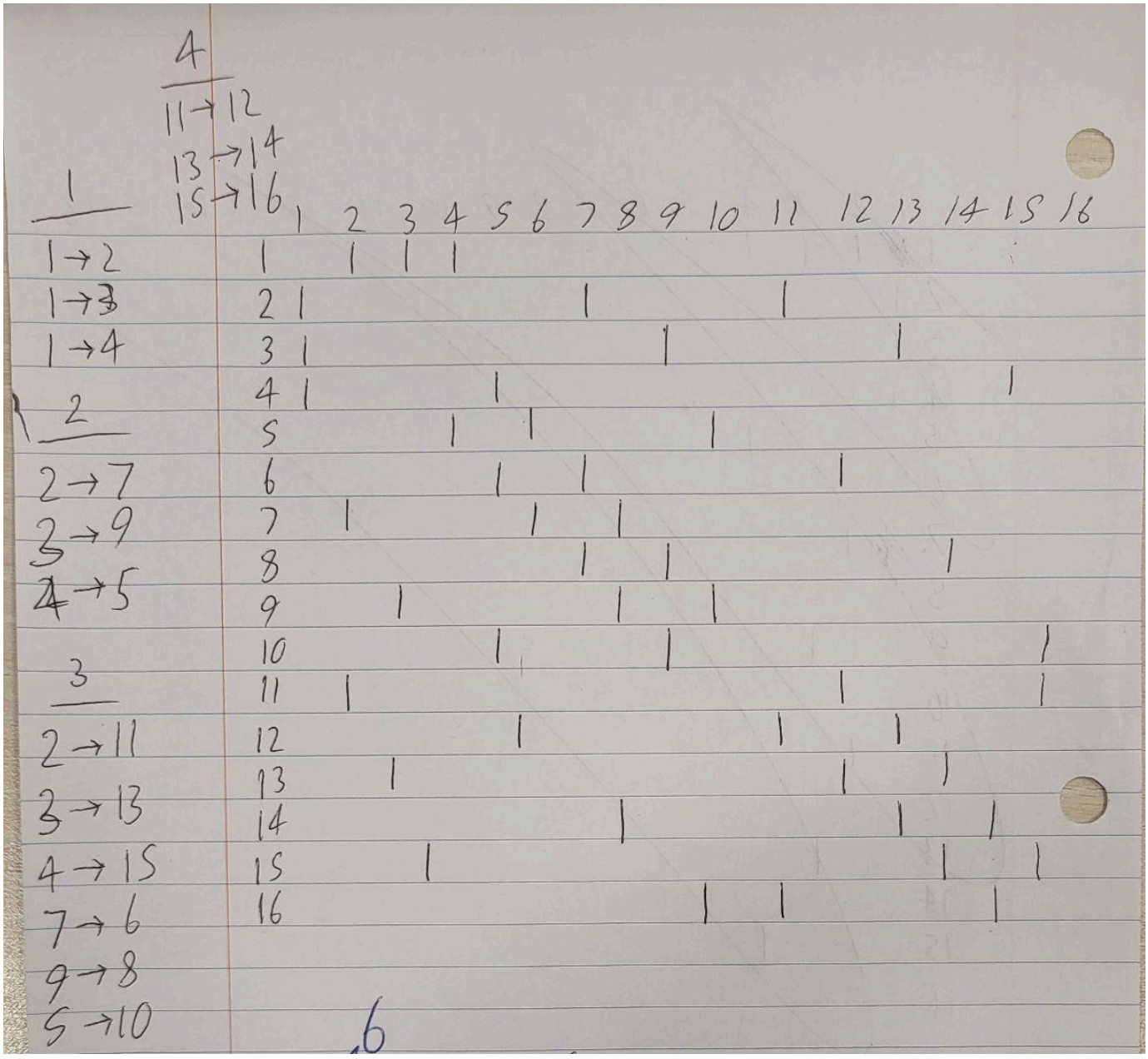
The data indicates that for all values of N, the built-in MPI broadcast function is significantly faster than the custom implementation. For N = 10^6, the custom algorithm takes approximately

0.159402 seconds, while `MPI_Bcast()` only takes 0.00932074 seconds, resulting in a performance ratio (T1/T2) of around 17.10186101. This trend continues for larger values of N, with the performance gap decreasing slightly as N increases. For N=10^7, the custom broadcast takes 1.36534 seconds, compared to 0.0812659 seconds for MPI's function, with a ratio of 16.80089681. Similarly, for N=10^8, the custom method takes 11.96 seconds, while the MPI function takes only 0.732255 seconds, leading to a performance ratio of approximately 16.333108.

These results suggest that the built-in `MPI_Bcast()` is highly optimized for large-scale broadcast operations, significantly outperforming the custom broadcast implementation. The consistent performance ratio, ranging between around 16 and 17 times faster across different data sizes, highlights the advantage of using optimized collective communication routines in MPI. 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.

	A	B	C	D
1	N	My_BCast() Time (T1)	MPI_Bcast() Time (T2)	T1 / T2
2	10^6	0.159402	0.00932074	17.10186101
3	10^7	1.36534	0.0812659	16.80089681
4	10^8	11.96	0.732255	16.333108

The image on the following page shows the adjacency matrix for this network topology along with some of my notes used for solving the problem.


4
11→12
13→14
15→16
1
1→2
1→3
1→4
2
2→7
3→9
4→5
3
2→11
3→13
4→15
7→6
9→8
5→10
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
6