Project 2 Report for Probelm 2.1

Zhu Liang

October 14, 2023

1 Project Description

In this project, we undertake two primary tasks. The first task is to provide a detailed description of the algorithms behind five collective communication operations provided by MPI. The description are provided in Section 2.1. The second task is an empirical comparison between the built-in `MPI_Bcast` and a custom broadcast implementation named `MY_Bcast()`.

2 Algorithm Description

2.1 MPI Build-in Operations

`MPI_Bcast()`: This function broadcasts a message from the process with the designated root rank to all other processes in the communicator. All processes must call this function, with matching arguments.

`MPI_Scatter()`: This function distributes distinct blocks of data from the root process to each process in the communicator. The root sends data to itself as well as to the other processes.

`MPI_Allgather()`: Each process sends its own data to all other processes and gathers data from all processes. At the end, every process has the data from all the other processes.

`MPI_Alltoall()`: This function allows each process to send distinct data to every other process. It generalizes the functionality of both scatter and gather, with different data being sent to each process.

`MPI_Reduce()`: All processes in the communicator contribute their own data, which is combined (reduced) into a single result using a specified operation, like sum, max, etc. The result is stored on the root process.

2.2 Custom Broadcast Function: `MY_Bcast()`

In large-scale parallel computations, efficient data transfer across processes is paramount. A simple broadcast approaches, which involve a root process sending messages to every other process sequentially, can be inefficient, especially when the number of processes grows, leading to a linear time complexity of $O(n)$.

To enhance this, we leverage it with a binary tree structure. Here, each process communicates only with its parent and potential left and right children (when children exist). Thus, it works even when the number of processes is a full binary tree. The root initiates the broadcast, and the message cascades down the tree, reaching all processes. Pleasee refer to Figure 1 for an

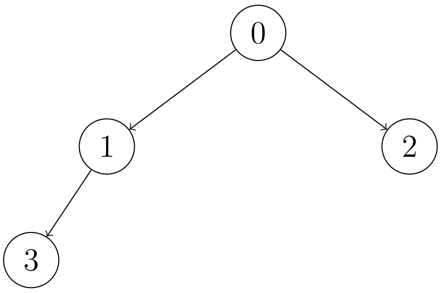

Figure 1: Binary Tree Structure when $P = 4$

illustration of the binary tree structure when $P = 4$, which is not a full binary tree. This approach reduces the time complexity from $O(n)$ to $O(log(n))$.

For more details on the project structure and the implementation, please refer to the `README.md` file in the `project2` folder.

The pseudocode is shown below.

Algorithm 1 Custom Broadcast Function Using Binary Tree

```
procedure MY_ Bcast
    Get rank and size of processes
    if current process rank is not root then
        Receive content of buffer from parent
    end if
    if left child exists then
        Send content of buffer to left child
    end if
    if right child exists then
        Send content of buffer to right child
    end if
    return MPI_SUCCESS
end procedure
```

3 Results

In the following tables, we present the execution times for two different broadcast implementations: `MPI_Bcast` and `MY_Bcast`. Please note that individual run times might vary depending on the specific runtime environment, but the general pattern observed should remain similar across different runs.

Referring to Tables 1 and 2, we can see the aforementioned times for various P and N values. Here, P represents processor count and N dictates size of data being broadcasted.

As part of our exercise to ensure accuracy in measurement, we implemented a rigorous tactic to eliminate various factors that could contribute to measurement errors. Specifically, for each pair of N and P, we performed 50,000 broadcast operations using the Bcast function, recorded the whole execution time, and then computed the average time. This repeated procedure allowed us to smooth out any potential anomalies - such as inconsistent system performance

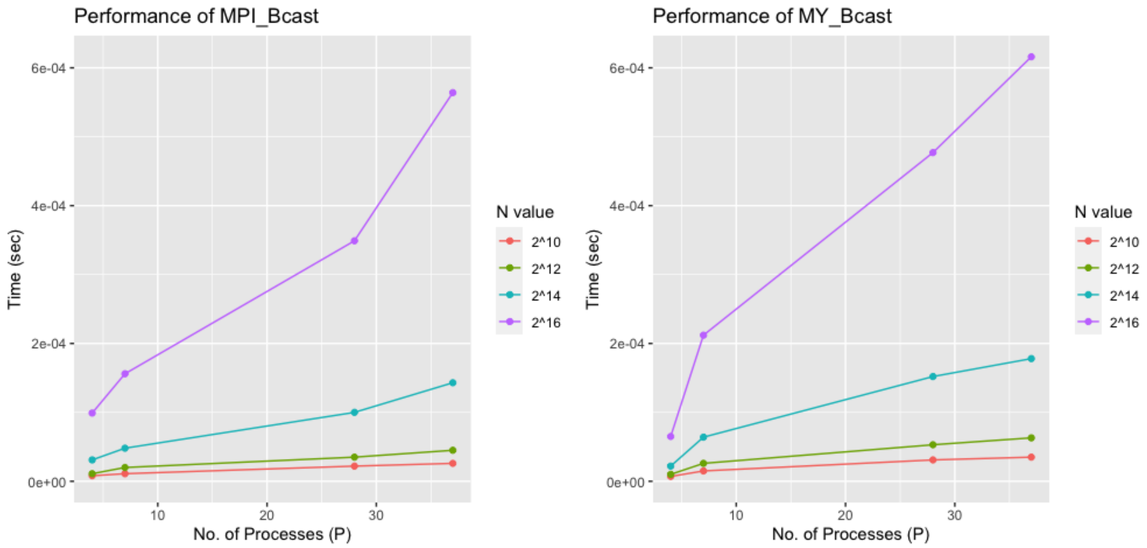

Figure 2: Comparison of Execution Time between `MPI_Bcast` and `MY_Bcast`

or unexpected delays - thereby providing us with a more reliable measure of typical execution time. We also add a warm-up phase before the actual measurement to ensure that the system is fully warmed up and ready to deliver consistent performance.

Upon assessing the results, an intriguing pattern has surfaced. When $P = 4$, `MY_Bcast` exhibits shorter execution time compared to `MPI_Bcast`. However, as P increases beyond the value of 4, `MPI_Bcast` begins to outperform `MY_Bcast`, offering better performance. This suggests that `MPI_Bcast` is potentially more beneficial for larger processor counts, where it delivers improved broadcast times.

	P = 4	P = 7	P = 28	P = 37
$N = 2^{10}$	0.000008s	0.000011s	0.000022s	0.000026s
$N = 2^{12}$	0.000011s	0.000020s	0.000035s	0.000045s
$N = 2^{14}$	0.000031s	0.000048s	0.000100s	0.000143s
$N = 2^{16}$	0.000099s	0.000156s	0.000349s	0.000564s

Table 1: Average Execution time using `MPI_Bcast`

	P = 4	P = 7	P = 28	P = 37
$N = 2^{10}$	0.000007s	0.000015s	0.000031s	0.000035s
$N = 2^{12}$	0.000010s	0.000026s	0.000053s	0.000063s
$N = 2^{14}$	0.000022s	0.000064s	0.000152s	0.000178s
$N = 2^{16}$	0.000065s	0.000212s	0.000477s	0.000616s

Table 2: Average Execution time using `MY_Bcast`

4 Analysis

Based on the empirical results obtained, both `MPI_Bcast()` and our custom implementation `MY_Bcast()` demonstrate closely matched performance. This is a significant observation given that the `MY_Bcast()` function was optimized to operate in $O(\log(n))$ time complexity, using a binary tree approach.

Delving deeper into the specifics of the performance under different scenarios reveals some noteworthy patterns. When the processor count is very samll, the MY_Bcast() function manages to edge out the native MPI_Bcast() in terms of execution time. This could be attributed to the simplistic structure of the MY_Bcast() function. Its binary tree-based broadcasting strategy doesn't involve complex computations or arrangements for data broadcast, which could potentially give it an advantage when dealing with a smaller number of processors. The reduced communication overhead and streamlined data management in these smaller settings might favor the less complicated design of MY_Bcast(), thereby leading to better performance as compared to MPI_Bcast().

However, as we scale up and increase the processor count beyond 4, the nature of these performance dynamics takes a noticeable shift. Under these circumstances, the native MPI_Bcast() function starts exhibiting superior performance over the MY_Bcast(). It handles the increased parallelism better and delivers faster broadcast times.

This phenomenon might be because MPI's built-in broadcast operation is highly optimized for larger processor counts, and it may employ sophisticated algorithms under the hood to manage inter-processor communication efficiently. In contrast, the simplistic binary-tree approach used in MY_Bcast() might not scale equally well with an increasing number of processors.

File Notes

The source code of the program is in the project2 folder. For more details, please refer to the README.md file in the folder.