AMS 530 PRINCIPLES OF PARALLEL COMPUTING
PROJECT 1

Wenhan Gao
Department of Applied Mathematics and Statistics, Stony Brook University, Stony Brook, NY 11794, USA
wenhan.gao@stonybrook.edu

1 Problem 1: MY Bcast

1.1 Problem Description

The MPI implementation has many collective operation and communication functions such as

- MPI_Bcast(): one processor (the root) sends the same data to all other processors in a communicator.
- MPI_Scatter(): one processor (the root) sends chunks of an array to all processors in a communicator.
- MPI_Allgather(): gather all of the data elements across all processors to all the processors; in other words, Allgather will collect all the individual data elements in different processors and broadcast them to all the processors.
- MPI_Alltoall(): the send buffer in each processor will send chunks of an array to all processors and then each column of chunks is gathered by the respective processor, whose rank matches the number of the chunk column.
- MPI_Reduce(): Takes an array of input elements on each process and returns an array of output elements to the root process[1]; reduces a set of numbers into a smaller set of numbers via a function. For example, suppose each processor contains one integer, MPI_MAX in MPI_Reduce() will return the maximum value across all processors to the root processor; as a reduction operation, the maximum among the n integers, where n is the number of processors (size), will be stored on the root process.

In this problem, we wish to write our own Broadcast function with only MPI_Isend and MPI_Irecv and compare the performance of our broadcast function with MPI_Bcast provided by MPI with different data sizes and numbers of processors.

1.2 Algorithm Description and Pseudo-code

The most straightforward but inefficient way to write this function is to have the root processor send the data to all other processors, and other processors receive the data from the root as demonstrated in Figure 1.1. This is very inefficient as it only utilizes the outgoing link of the root processor.

For efficiency, in this program, we implement the broadcast with **binomial trees [2]**. With binomial tree structure, we utilize all the processors that have the data to send the data to other processors. As demonstrated in Figure 1.2, in the first step, only 0 has the data, and it will send the data to another processor, then in the next step, there are two processors that have the data, and each of them can send the data to another processor. In each step, optimally, we may double the number of processors that have the data.

Figure 1.1: One to All Broadcast; image from [3]

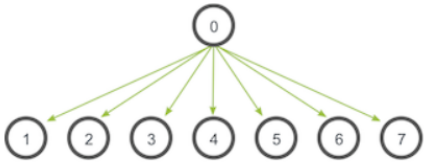
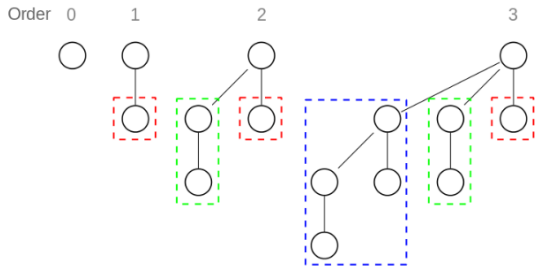


Figure 1.2: Binomial Tree Broadcast; image from [4]



Algorithm 1.1: MY Bcast with Binomial Tree Structure (efficient)

Result: Broadcast the data from the root processor, WLOG let it be 0, to all processors in the network.
Require: data, size, rank

```
left := 0; right := size -1;
Loop
    if left == right then
        break;
    mid = (left + right + 1)/2;
    if rank ≥ left & rank < mid then
        if rank == left then
            MPI send data to Processor mid (rank is mid);
        right = mid -1;
    else
        if rank == mid then
            MPI receive data from Processor left (rank is left);
        left = mid;
    end
EndLoop
```

Algorithm 1.1 is the pseudocode for our efficient broadcast algorithm. In Algorithm 1.1, we bisect the network in each step to implement the binomial tree. We always send the data to the middle processor, and then bisect the network into two sub-networks, and the middle becomes the "start" of the new sub-network. This is similar to the array bisection method aka binary search method. It is not hard to tell that the complexity of this algorithm is $O(\log n)$, whereas the complexity of Algorithm 1.2 is $O(n)$.

Algorithm 1.2: MY Bcast with One-to-all Structure (inefficient)

Result: Broadcast the data from the root processor, WLOG let it be 0, to all processors in the network.
Require: data, size, rank

```
if rank == 0 then
    MPI send data to all other Processors
else
    MPI receive data from Processor 0;
end
```

1.2.1 Main Structure of the Program

The main program contains only one C script. In this C script, there are one main function and two broadcasting functions as stated above (Algorithms 1.2 and 1.1).

For the main function, i.e., comparison of MY Bcast with MPI Bcast, we create a floating number (4 bytes) array of a given size. We take the maximum time across all processors as the global elapsed running time. We record the

global elapsed running time that each broadcasting function takes and we run each case 10 times for each broadcasting function and then take the average to reduce time measurement anomalies.

1.2.2 Slurm Configuration

To run the program on Seawulf, we have a slurm script. In this script, we request the node **long-96core** and request 30 seconds of run-time for trials in Table 1 and 03:00 minutes of run-time for trials in Table 2. Also in this script, we include our C compiler module (mvapich2/gcc12.1/2.3.7), and we compile the C code with mpicc. All stdout texts will be saved into a text file.

Upload the C file and slurm file to Seawulf, and then enter the following commands to run the program:

module load slurm

sbatch problem1.slurm

1.3 Results

Table 1 records the performance of MY Bcast function (binomial tree structure) and the MPI Bcast function with various total numbers of processors and data-sizes measured by the number of floating-point numbers.

		P=4	P=7	P=28	P=37
$N = 2^{10}$	MPI_Bcast	0.000015	0.000066	0.000164	0.000252
	MY_Bcast_Binomial	0.000010	0.000046	0.000064	0.000116
$N = 2^{12}$	MPI_Bcast	0.000022	0.000034	0.000151	0.000250
	MY_Bcast_Binomial	0.000015	0.000031	0.000100	0.000212
$N = 2^{14}$	MPI_Bcast	0.000045	0.000082	0.000192	0.000292
	MY_Bcast_Binomial	0.000026	0.000056	0.000142	0.000224
$N = 2^{22}$	MPI_Bcast	0.005662	0.007705	0.020284	0.017206
	MY_Bcast_Binomial	0.004191	0.007881	0.023949	0.027700

Table 1: Performance Table: Small Scale

In Table 1, in most cases, MY Bcast actually outperforms the MPI Bcast. However, it has to be pointed out that it is due to the small scale of the broadcast, most of these cases are done with one tenth of a second. When the task is relatively "large", for example when $N = 2^{22}, P = 7$, MPI Bcast starts to outperform MY Bcast. By looking at the MPI Bcast source code [5], MPI Bcast uses a binomial tree algorithm for short messages, but for long messages, MPI Bcasts does a scatter followed by an allgather. One possible reason is that MPI Bcast is optimized for large scale broadcasting, so there are additional features that slow it down in tiny scale broadcasting. Here we provide additional timing results in Table 2 to show the performance of MY Bcast for relatively larger broadcasting. Figure 1.3 shows an example output of the program.

		P=37	P = 95
$N = 2^{26}$	MY Bcast OneToAll	1.011304	2.757015
	MPI Bcast	0.240675	0.307109
	MY Bcast Binomial	0.246781	0.422343
$N = 2^{28}$	MY Bcast OneToAll	4.079726	11.104211
	MPI Bcast	0.882373	1.243521
	MY Bcast Binomial	0.910246	1.592811

Table 2: Performance Table: Relatively Large Scale

Figure 1.3: Problem 1 Example Output: 7 Processors and a Data-size of 2^{10} On Short 28 Core With Interactive Shell

```
[wenhgao@sn144 ~]$ mpirun ./p1_main
Performance with 7 processors and a data-size of 1024
Average global elapsed runtime for MY_bcast_oneToall is 0.000026 seconds
Average global elapsed runtime for MPI_Bcast is 0.000043 seconds
Average global elapsed runtime for MY_bcast_binomialTree is 0.000029 seconds
[wenhgao@sn144 ~]$
```

1.4 Analysis of Program Performance

As stated above, the complexity of MY Bcast with binomial tree algorithm is $O(\log n)$ with respect to the number of processors. Not surprisingly, it outperforms the OneToAll structure broadcast by a lot. From the MPI documentation [5], the MPI Bcast also uses binomial trees for short messages. MY Bcast is highly efficient as demonstrated by numerical results showing in Tables 1 and 2.

1.5 Source Code for Submission

Source code will be submitted along with this report. Source code can also be found in this git repo (Clickable Link).

This git repo will be deleted after the end of the semester to prevent future students from reusing the code.

2 Problem 2: MY Global Max Loc

2.1 Problem Description

The MPI implementation has many collective operation and communication functions such as

- MPI_Bcast(): one processor (the root) sends the same data to all other processors in a communicator.
- MPI_Scatter(): one processor (the root) sends chunks of an array to all processors in a communicator.
- MPI_Allgather(): gather all of the data elements across all processors to all the processors; in other words, Allgather will collect all the individual data elements in different processors and broadcast them to all the processors.
- MPI_Alltoall(): the send buffer in each processor will send chunks of an array to all processors and then each column of chunks is gathered by the respective processor, whose rank matches the number of the chunk column.
- MPI_Reduce(): Takes an array of input elements on each process and returns an array of output elements to the root process [1]; reduces a set of numbers into a smaller set of numbers via a function. For example, suppose each processor contains one integer, MPI_MAX in MPI_Reduce() will return the maximum value across all processors to the root processor; as a reduction operation, the maximum among the n integers, where n is the number of processors (size), will be stored on the root process.

In this problem, we wish to write our own global maximum location function without MPI_Reduce(). Given an array of length N in each processor, we wish to return an array of length N in which the i-th entry is the rank of the processor that holds the maximum value in this entry. For example,

$$P = 0, A_0 = \{1, 2, 3, 4\}$$
$$P = 1, A_1 = \{5, 6, 7, 2\}$$
$$P = 2, A_2 = \{9, 8, 6, 1\}$$

for the $A_0[0], A_1[0], A_2[0]$ elements, the max (9) resides on Proc 2 ;

for the $A_0[1], A_1[1], A_2$ [1] elements, the max (8) resides on Proc 2;

for the $A_0[2], A_1[2], A_2$ [2] elements, the max (7) resides on Proc 1 ;

for the $A_0[3], A_1[3], A_2$ [3] elements, the max (4) resides on Proc 0 ;

Thus, we wish to report an array of ranks $\{2, 2, 1, 0\}$.

2.2 Algorithm Description and Pseudo-code

The most straightforward but inefficient way to write this function is to have all the processors send their local array to a master processor, and this master processor collects all the local arrays, compares the values, and returns the max loc array.

An efficient way is to use the **binomial trees [2]** inversely; just FYI, butterfly structure should work as well. With binomial tree structure, in each step, we divide the network of processors into two sub-networks, WLOG, we pair each processor in a sub-network with a processor in the other sub-network, compare their values, and record the maximum values and respective ranks.

Algorithm 2.1 is the pseudocode for MY_Global_Max_Loc(). So basically, in this algorithm, each processor has a local max value array and rank array. We bisect the network into two sub-networks in each step and then pair processors in these two sub-networks and compare their values to update the max value arrays and rank arrays in one sub-network, and then bisect this sub-network again. Continue this progress, at the end, we will have one processor whose max value array will record the maximum values across the network and rank array will record the respectively ranks.

2.2.1 Main Structure of the Program

The main program contains only one C script. In this C script, there are one main function and a MY_Global_Max_Loc() function as stated above (Algorithm 2.1).

In the main function, we generate random integer local arrays for all processors, and then we run MY_Global_Max_Loc() to get the array of ranks that represents the maximum locations. Also, we use MPI_Reduce() to get the array of ranks to check if our program is giving correct results.

2.2.2 Slurm Configuration

To run the program on Seawulf, we have a slurm script. In this script, we request the node **short-28core**, set the number of processors (tasks) to be 28, and request 15 seconds of run-time. Also in this script, we include our C compiler module (mvapich2/gcc12.1/2.3.7), and we compile the C code with mpicc. All stdout texts will be saved into a text file.

Upload the C file and slurm file to Seawulf, and then enter the following commands to run the program:

module load slurm

sbatch problem2.slurm

2.3 Results

Figure 2.1 shows an example output of the program.

Figure 2.1: Problem 2 Example Output: 8 Processors On Short 24 Core With Interactive Shell

```
[wenhgao@sn058 ~]$ mpirun ./p2_main
Local array from Processor 0: [46 85 68 40 25 40 72 76 ]
Local array from Processor 2: [75 65 10 72 76 32 20 49 ]
Local array from Processor 3: [41 85 12 65 8 85 86 43 ]
Local array from Processor 4: [77 99 99 71 25 43 86 97 ]
Local array from Processor 5: [96 44 42 49 11 93 82 21 ]
Local array from Processor 6: [15 74 65 7 30 53 20 78 ]
Local array from Processor 7: [95 8 78 25 18 77 75 71 ]
Local array from Processor 1: [1 83 74 26 63 37 25 63 ]
MY max_loc array is: [5 4 4 2 2 5 3 4 ]
The max_loc array given by MPI_reduce is: [5 4 4 2 2 5 3 4 ]
[wenhgao@sn058 ~]$
```

Algorithm 2.1: MY Global Max Loc

Result: Returns an array of ranks with the maximum values
Require: local arrays of length N, size, rank
Initialize, for all processors, my_max_array := local array, my_max_loc_array of the same size as the local array and all entries equal to its rank;
Initialize, for all processors, temp_max_array and temp_max_loc_array of the same size as the local array;
right := size -1;
Loop
  if *right = 0* **then**
    break;
  mid = (right + 1)/2;
  ; ▷ when the size is an odd number
  if *right = 2 * mid* **then**
    if *rank > mid* **then**
      MPI send my_max_array and my_max_loc_array to Processor rank - mid (the processor's id is rank - mid);
    if *rank <= mid and rank ≠ 0* **then**
      MPI receive temp_max_array and temp_max_loc_array to Processor rank + mid;
      for *i in range (0, N)* **do**
        compare the i-th element in my_max_array and temp_max_array, if equal, update my_max_loc_array as min(rank, temp_max_loc_array[i]), if my_max_array[i] < temp_max_array[i], update my_max_array[i] := temp_max_array[i] and my_max_loc_array[i] := temp_max_loc_array[i];
      end
  ; ▷ when the size is an even number
  else
    if *rank >= mid* **then**
      MPI send my_max_array and my_max_loc_array to Processor rank - mid (the processor's id is rank - mid);
    if *rank < mid* **then**
      MPI receive temp_max_array and temp_max_loc_array to Processor rank + mid;
      for *i in range (0, N)* **do**
        compare the i-th element in my_max_array and temp_max_array, if equal, update my_max_loc_array as min(rank, temp_max_loc_array[i]), if my_max_array[i] < temp_max_array[i], update my_max_array[i] := temp_max_array[i] and my_max_loc_array[i] := temp_max_loc_array[i];
      end
  end
  right = right/2;
  MPI Barrier;
EndLoop
return my_max_array and my_max_loc_array in Processor 0;

2.4 Analysis of Program Performance

As shown in Figure 2.1, MY_Global_Max_Loc() gives the correct result. We did more experiments with various local array lengths and numbers of total processors, MY_Global_Max_Loc() is stable and gives correct results. Since each time we reduce the size of the sub-networks by half, the complexity of our algorithm is $O(\log n)$ with respect to the total number of processors. Therefore, the program is efficient even for a large network.

2.5 Source Code for Submission

Source code will be submitted along with this report. Source code can also be found in this git repo (Clickable Link).

This git repo will be deleted after the end of the semester to prevent future students from reusing the code.

References

[1] MPI Reduce and Allreduce; MPI Tutorial. https://mpitutorial.com/tutorials/mpi-reduce-and-allreduce/, . [Accessed 03-Oct-2022].

[2] Binomial Heap. https://www.geeksforgeeks.org/binomial-heap-2/. [Accessed 02-Oct-2022].

[3] MPI Broadcast and Collective Communication; MPI Tutorial. https://mpitutorial.com/tutorials/mpi-broadcast-and-collective-communication/, . [Accessed 02-Oct-2022].

[4] Wikipedia contributors. Binomial heap — Wikipedia, the free encyclopedia, 2022. URL https://en.wikipedia.org/w/index.php?title=Binomial_heap&oldid=1076478439. [Online; accessed 2-October-2022].

[5] MPI File Reference. https://formalverification.cs.utah.edu/sawaya/html/d5/d65/mpi_2coll_2bcast_8c-source.html. [Accessed 02-Oct-2022].