DESIGN AND IMPLEMENT A COLLECTIVE COMMUNICATION USING NON_BLOCKING MPI PRIMITIVES

Homework # 5

By

Yichen Huang

CS 481 High-Performance Computing

November 15, 2021

I. Problem Specification

The Open MPI is an open-source Message Passing Interface library project. The main object of this assignment is to design and implement a collective communication primitive using non-blocking Message Passing Interface (MPI) primitives. The MPI_Allgather function is to collect data from each process and join the collected data to the receive buffer in the destinate process. After joining the data, the receive buffer will be broadcast to all processes. Therefore, the buffer in all processes will have the same value. In this assignment, two methods will be applied to mimic the MPI_Allgather function. The first one will use non-blocking point-to-point functions to pass the message from process to process. The second method will apply the pair-wise exchange method by MPI_Sendrecv function to emulate the MPI_Allgather function.

II. Program Design

The assignment will have five files – driver.c, allgather1.h, allgather1.c, allgather2.h, and allgather2.c.

- driver.c and allgather_driver.c

  The main function of driver.c will be implemented in this file to test and debug the whole assignment. This file is modified by the myscatter.c [1] from the blackboard. The myscatter function will firstly scatter the whole sendbuf to the local myrecvbuf in each process. Then allgather function will gather the myrecvbuf together to the recvbuf in each process. After allgather function, assert function will determine if each element in recvbuf in each process is same as the those in sendbuf.

  The allgather_driver.c is from Blackboard. The program will only allocate the local send buffer and receive buffer. The function will call the allgather function to gather local send buffer to the receive buffer of all processes. The time taken for different message size will be recorded.

- allgather1.h and allgather1.c

  The head file – allgather1.h – is to declare the function allgather with the non-blocking point-to-point method. The allgather1.c file is to implement the function. The function will let every process send each chunk of data to the root process which is the same as MPI_Gather function. Then the root

process will send the collected data to each process which is the same as the MPI_Broadcast function. If the function runs successfully, it will return 0.

- allgather2.h and allgather2.c

  The head file – allgather2.h – is to declare the function allgather with the pair-wise exchange method. The allgather2.c is to implement the function. To reach the pair-wise exchange method; firstly, each process will send the local chuck of data to the local receive buffer with an offset that will record the location of that chunk of data is located. Then, for N processes, the log2(N) times are needed to swap between each process; therefore, for each processer, the destination for each iteration – i – is (rank + 2^i) or (rank – 2^i). After the pair-wise swap, the receive buffer for all processes will be the same.

III. Testing Plan

To test the non-blocking point-to-point method and pair-wise exchange method, allgather1.c and allgather2.c should compile separately with the driver.c. The testing plan has two parts, the first part is to check if the allgather works and the second part will check the time for different message sizes and verify the correctness of the allgather function.

- **Part I**

  mpicc -O -Wall -D DEBUG -lm -o ver1 driver.c allgather1.c
  mpicc -O -Wall -D DEBUG -lm -o ver2 driver.c allgather2.c

  These two commands will generate two executable files – ver1 and ver2 – in debug mode. In debug mode, the message size will be set to 8. The num_process*8 lengths of sendbuf will be scattered to local myrecvbuf. Then allgather function will collect myrecvbuf from each process and gather to recvbuf in each process. The program will print out the recvbuf and process number which is helpful to check out.

- **Part II**

  mpicc -O -Wall -lm -o ver1 driver.c allgather1.c
  mpicc -O -Wall -lm -o ver2 driver.c allgather2.c
  mpicc -Wall -lm -o myallgather1 allgather_driver.c allgather1.c
  mpicc -Wall -lm -o myallgather2 allgather_driver.c allgather2.c

  These four commands will also generate two executable files – ver1, ver2, myallgather1, and myallgather2. These programs will test allgather functions by different message sizes. The maximum time running time for the allgather function will be recorded and printed out.

IV. Test Cases

The results of part I are stored at /Result/Debug. The result of ver1and ver2 are stored at /Result/Test1 folder, and the results of myallgather1 and myallgather2 are stored at /Result/Test2 in the GitHub repo[1].

- Time measurement for non-blocking point-to-point method from allgather_driver.

		Processes				
		1	2	4	8	16
Bytes	32	1.82E-06	2.61E-06	4.02E-06	7.35E-06	1.71E-05
	64	3.06E-07	1.76E-06	2.67E-06	6.42E-06	1.25E-05
	128	3.39E-07	1.71E-06	3.51E-06	6.36E-06	1.45E-05
	256	3.35E-07	2.11E-06	3.56E-06	7.09E-06	2.79E-05
	512	1.10E-06	3.82E-06	5.83E-06	1.61E-05	3.66E-05
	1024	1.55E-06	4.53E-06	9.27E-06	1.92E-05	5.09E-05
	2048	8.57E-07	5.86E-06	1.04E-05	2.58E-05	8.50E-05
	4096	1.43E-06	9.72E-06	1.79E-05	4.73E-05	1.54E-04
	8192	1.02E-06	1.27E-05	2.69E-05	7.79E-05	2.77E-04
	16384	1.89E-06	2.05E-05	4.48E-05	1.41E-04	5.34E-04
	32768	2.25E-06	3.47E-05	7.83E-05	2.77E-04	1.01E-03
	65536	5.01E-06	6.76E-05	1.53E-04	5.18E-04	2.04E-03
	131072	8.04E-06	1.45E-04	2.81E-04	1.05E-03	4.07E-03
	262144	1.72E-05	2.90E-04	5.61E-04	2.08E-03	8.42E-03
	524288	3.57E-05	5.71E-04	1.07E-03	4.12E-03	1.71E-02
	1048576	1.50E-04	1.20E-03	2.15E-03	8.88E-03	3.43E-02

[1] https://github.com/yhuang86/UA-High_Performance_Computing/tree/main/Homework%235

- Time measurement for pairwise exchange method

		Processes				
		1	2	4	8	16
Bytes	32	1.52E-06	3.86E-06	1.07E-05	2.18E-05	5.05E-05
	64	1.95E-07	3.02E-06	8.11E-06	1.81E-05	3.92E-05
	128	1.90E-07	3.25E-06	8.81E-06	2.01E-05	4.27E-05
	256	1.94E-07	3.33E-06	9.46E-06	2.16E-05	4.60E-05
	512	2.98E-07	4.41E-06	1.25E-05	2.89E-05	6.00E-05
	1024	4.96E-07	5.18E-06	1.43E-05	3.17E-05	6.76E-05
	2048	3.55E-07	5.52E-06	1.54E-05	3.58E-05	8.09E-05
	4096	4.22E-07	1.04E-05	3.28E-05	7.76E-05	1.71E-04
	8192	5.39E-07	1.34E-05	4.21E-05	1.05E-04	2.43E-04
	16384	1.42E-06	1.89E-05	6.13E-05	1.65E-04	3.92E-04
	32768	1.94E-06	2.97E-05	1.07E-04	2.94E-04	7.21E-04
	65536	3.61E-06	5.37E-05	1.81E-04	4.63E-04	1.15E-03
	131072	7.65E-06	1.01E-04	3.21E-04	8.02E-04	2.10E-03
	262144	1.59E-05	2.11E-04	6.33E-04	1.56E-03	5.16E-03
	524288	3.44E-05	4.13E-04	1.20E-03	2.94E-03	1.01E-02
	1048576	1.49E-04	9.90E-04	2.55E-03	7.03E-03	2.06E-02

V. Analysis and Conclusion

The machine used to run the program is the 2021 MacBook Pro. The operating system is macOS Monterey (Version 12.0.1). The CPU of the machine is Apple M1 Max with 10 core CPU and 32 core GPU. The basic clock speed of the CPU is 3.20 GHz. The memory in the system is 32 GB LPDDR5 unified memory. When running the code on the ASC cluster, the number of cores used for the test is the same as the number of processes described in Section IV.

The program achieves the goal of implementing the allgather function by the non-blocking point-to-point method and pair-wise exchange method. The send buffer will be scattered to local myrecvbuf and allgather function will gather myrecvbuf to recvbuf in each process. After allgather function, assert function will compare the recvbuf and sendbuf. If the allgather function works correctly, the two buffers should be the same.

From the timetables in Section IV Part II, the time taken for method 1 is less than method 2 when the byte size is small; and when byte size is large, method 2 takes less time than method 1. This is because although method 1 uses less sending and receiving operation compared with method 2 when message size is large, the whole message will be separated into several chunks and sent to the destination process. After all, the cash size of the CPU is fixed. When the byte size is same, the method 1 spends less time than method 2. The reason is probably that MPI_Sendrecv will spend more time waiting than method 1.

From the table of method 1 and method 2, when message size is the same, the time is taken increases with the increasing number of processes. The possible reason is the more processes, the more communication between different processes. Because there is a waiting time between sending a message and receiving a message between processes, the time taken will increase when the process number increase. Moreover, when the processes number is fixed, the time taken increases with the message size increase for the same method. The possible reason is that the cash size for CPU is fixed when sending a long message between processes, the message will be separated into several chunks and all chunks will be sent one by one. Longer the message, the larger chunks. The overall time taken will increase because there will be more wait time between sending chunks of messages.

To further improve and optimize the program, I think the MPI_Barrier function has been used a lot in the program. The barrier will make sure all process goes to the next step when they call the same barrier function which can make the program more stable. However, the abuse of the function will waste lots of time waiting for all processes. To optimize the overall program, the number of MPI_Barrier could be reduced which can shorten the time taken.

Reference

[1] The myscatter.c is from https://github.com/pvbangalore/Fall2021-CS481-mpi-examples/blob/main/myscatter.c