COLLECTIVE COMMUNICATION PRIMITIVE DESIGN

Homework 5

By

Colin Sanders

CS 581 High Performance Computing
November 19, 2021

Problem Specification

The goal of this homework is to design and implement the "allgather" collective communication operation by using non-blocking Message Passing Interface (MPI) primitives. The "allgather" function first gathers array segments from different processes into a complete array, and then distributes the array to every process.

The objectives of this project are:

1. implement the "allgather" operation using non-blocking point-to-point message passing primitives provided by MPI using the following algorithms:
    a. an implementation of the "gather" operation followed by an implementation of the "broadcast" operation
    b. pair-wise exchange using MPI_Sendrecv, where processes pair with each other to each gather portions of the array simultaneously
2. test the program using the driver provided
3. plot the time taken by each of the above "allgather" implementations for different numbers of processes and message sizes
4. provide analysis of the timing measurements
5. specifically for graduate students, develop expressions to compute the total time spent in communication for each of the above algorithms

Program Design

The first implementation of "allgather" requires the root process performing a "gather" operation, where it receives all pieces of an array split amongst processes, and then a "broadcast" operation, where the root process distributes the complete array to each process. As indicative by this high-level explanation, I split the logic into two portions: code to be executed only by the

root and code to be executed by other process. Every process begins by sending their respective portion of the array, and then all of the non-root processes go into a period of waiting. The root process then calculates where process's data should be offset into the complete array and sets up receive calls for each process with the correct pointers and data sizes. The root process then waits for all of these receives to be complete, indicating that the entire array has been gathered. Now the non-root processes, having been unblocked from their Isend->Wait, now set up receive requests for the root to distribute the entire array, and wait till competition. The root initiates Isend requests for each other process with the full data array and then finishes with a Waitall to guarantee all processes have received the complete array before returning, as the traditional "allgather" call is blocking.

The second implementation does not have a root process leading a gather into broadcast, but rather pairs of processes combine their portions of the array with each other until the entire array is built on all processes. The key things to determine in this algorithm are which process another should pair with in a given phase, how much data should be transferred in said phase, and where in the processes' receive buffers should the data be transferred to. Two assumptions given to us as part of the requirements make determining these factors much easier: both the number of processes and the amount of data each process begins with are guaranteed to be powers of 2. This means that the number of iterations that processes will pair up in order for all processes to receive the entire array is guaranteed to be log2(size) where size is the number of processes. Beginning with 1, the difference in process rank between any two processes is multiplied by 2 at each generation, described as the "partnerOffset" in my implementation. Because of the assumptions stated above, the number of bytes transferred in a given iteration is the "partnerOffset" times the number of bytes in the original split arrays. Additionally, all processes preload their own final arrays with their data and create a pointer that points to its own portion of the final array, titled "bufptr".

The next piece of information to determine is whether a given process pairs with the process of rank difference "partnerOffset" to the "left" of them ("partnerRank" < "myRank") or to the "right" of them ("partnerRank" > "myRank"). In the first phase, all even processes partner with the process to their right by adding 1 to their rank, and the odd processes conversely subtract 1 from their rank. For every phase after this, processes get split into chunks where certain chunks of processes partner left and certain chunks partner down, with the chunk calculated by (rank / "partnerOffset") being odd or even (odd pairs left, even pairs right). Whenever a process calculates whether it will partner left or right, it also determines the location of the new data to be received using the same sign as the partner calculation and adding or subtracting from their "bufptr" to create a new pointer called "recvptr". After this, we have all of the data necessary to call MPI_Sendrecv and exchange data between partners. One last trick to keep the logic simple is that if a process partnered left, its "bufptr" is updated to point to the left most piece of data it has received in its final array.

Testing Plan

The testing plan for correctness is as follows: Each process creates an array with data related to their rank and arbitrary data sizes and a receive buffer large enough to hold the data of all of the processes. After the completion of the "allgather", each process asserts that every element of the

final array is correct. For performance, there are multiple iterations testing the "allgather" function for process starting array segment sizes ranging from 32 bytes up to 2^20 bytes. For each segment size, "allgather" is executed 100 times and the maximum time taken by an execution is recorded, with this maximum time average across three total executions (3 * 100 executions for each of the 16 segment sizes). These executions were tested on the ASC DMC clusters using the new AMD Milan CPU nodes, this time ensured that all executions are using the same nodes on the cluster. The results of the performance evaluation is recorded below.

Performance Results (P = Number Of Processes)

Gather & Broadcast Implementation

Num Bytes	P = 1	P = 2	P = 4	P = 8	P = 16
32	2.71e-06 s	4.99e-06 s	6.03e-06 s	1.02e-05 s	2.28e-05 s
64	2.11e-07 s	2.61e-06 s	4.00e-06 s	1.06e-05 s	1.89e-05 s
128	2.35e-07 s	3.17e-06 s	6.59e-06 s	1.03e-05 s	2.85e-05 s
256	2.30e-07 s	4.20e-06 s	6.38e-06 s	1.52e-05 s	5.81e-05 s
512	3.33e-07 s	5.18e-06 s	9.34e-06 s	3.34e-05 s	9.21e-05 s
1024	4.33e-07 s	7.14e-06 s	1.62e-05 s	4.63e-05 s	1.46e-04 s
2048	3.44e-07s	1.36e-05 s	2.74e-05 s	7.48e-05 s	2.71e-04 s
4096	3.95e-07 s	2.48e-05 s	5.09e-05 s	1.53e-04 s	3.59e-04 s
8192	4.73e-07 s	3.74e-05 s	9.03e-05 s	2.81e-04 s	4.96e-04 s
16384	7.12e-07 s	5.94e-05 s	1.60e-04 s	3.37e-04 s	9.64e-04 s
32768	1.03e-06 s	1.11e-04 s	2.98e-04 s	5.00e-04 s	1.91e-03 s
65536	1.74e-06 s	2.16e-04 s	3.39e-04 s	9.86e-04 s	3.78e-03 s
131072	3.33e-06 s	4.23e-04 s	5.19e-04 s	1.93e-03 s	7.48e-03 s
262144	7.33e-06 s	8.37e-04 s	1.01e-03 s	3.80e-03 s	1.50e-02 s
524288	1.95e-05 s	1.73e-03 s	2.00e-03 s	7.63e-03 s	3.00e-02 s
1048576	2.97e-05 s	3.50e-03 s	4.03e-03 s	1.52e-02 s	6.03e-02 s

Pair-Wise Implementation

Num Bytes	P = 1	P = 2	P = 4	P = 8	P = 16
32	2.32e-06 s	3.96e-06 s	4.53e-06 s	5.32e-06 s	8.73e-06 s
64	2.75e-08 s	1.31e-06 s	1.97e-06 s	2.62e-06 s	4.57e-06 s
128	2.84e-08 s	1.35e-06 s	2.05e-06 s	3.23e-06 s	5.44e-06 s
256	2.87e-08 s	1.54e-06 s	2.85e-06 s	3.88e-06 s	6.56e-06 s
512	3.09e-08 s	2.94e-06 s	4.58e-06 s	8.08e-06 s	2.24e-05 s
1024	3.79e-08 s	3.43e-06 s	7.81e-06 s	1.52e-05 s	2.77e-05 s
2048	5.53e-08 s	5.09e-06 s	9.71e-06 s	1.54e-05 s	3.22e-05 s
4096	1.05e-07 s	1.37e-05 s	2.38e-05 s	4.04e-05 s	8.03e-05 s
8192	1.88e-07 s	1.90e-05 s	3.81e-05 s	7.54e-05 s	1.58e-04 s
16384	3.80e-07 s	2.71e-05 s	7.08e-05 s	1.51e-04 s	3.26e-04 s
32768	7.85e-07 s	5.00e-05 s	1.18e-04 s	2.61e-04 s	5.86e-04 s
65536	1.44e-06 s	9.43e-05 s	1.04e-04 s	4.82e-04 s	1.00e-03 s
131072	2.94e-06 s	1.37e-04 s	1.54e-04 s	7.51e-04 s	7.83e-04 s
262144	7.12e-06 s	2.26e-04 s	3.03e-04 s	6.41e-04 s	1.39e-03 s
524288	1.88e-05 s	3.29e-04 s	6.02e-04 s	1.28e-03 s	3.53e-03 s
1048576	2.91e-05 s	6.13e-04 s	1.19e-03 s	2.60e-03 s	8.32e-03 s

Analysis and Conclusions

Overall, both algorithms ran extremely quickly even on the large datasets and large number of processes that were intercommunicating. The pairwise algorithm pulled ahead in performance when significantly increasing the data size and even more so when increasing the number of processes. At the two largest data sizes (bytes = 524299, 1048576) and the largest process count (P=16), the pairwise implementation had a lower execution time by a factor of 10. My initial implementation of the pairwise program was running about 0-5% faster than the gather & broadcast program due to some unnecessary and costly math taking place on all processes on

each phase but reducing the math redundancy improved the performance to show how the pairwise is a significantly faster implementation.

If t is the time taken for any send and any receive communication between two processes, the total time spent in communication is as follows for each implementation. For the gather & broadcast implementation, all processes initiate a send that takes time t at the same time, the root process then sets up non-blocking receives that each take time t (size * t), then starts another set of simultaneous sends t (size * t), and then each other process simultaneously receives with time t, to time spent in communication is equal to 2 * (size * t) + 2 t = **2 * ((size + 1) * t)**. For the pairwise, the time spent in send and receives is now dependent on the number of processes, as each process does log2(size) number of phases that each have a 2T sendrecv call but never all to the same process, so these 2T calls happen simultaneously among pairs. Thus, the total time of the pairwise operation can be modeled as **log2(size)*2T.** It is difficult to accurately compare these two operations with these two models though, as neither accounts for the size of the data being transmitted. In the pairwise transmission, at most half as much data is being transmitted in the largest data transmission phase than the broadcast of the other operation, which is another aspect of the algorithm that instills better performance.

The goals of the assignment were successfully achieved: both the gather/broadcast and pairwise implementations of the "allgather" collective communication primitive were implemented, tested, and analyzed. Determining how and when processes should communicate required thoughtful design and consideration in order to minimize the amount of time wasted among processes.

References

The main sources of information used for this project were the textbook and Dr. Bangalore's scatter and broadcast implementations.

Documentation for MPI_Isend, MPI_Irecv, MPI_Sendrecv
https://www.mpich.org/static/docs/v3.1/www3/MPI_Isend.html
https://www.mpich.org/static/docs/v3.3/www3/MPI_Irecv.html
https://www.mpich.org/static/docs/v3.3/www3/MPI_Sendrecv.html

Github Repository: https://github.com/CCSanders/CS581_HW5