Priority-aware MPI_Reduce-Scatter Operation

Mohsen Gavahi
Department of Computer Science
Florida State University
Tallahassee, FL, USA
gavahi@cs.fsu.edu

***Abstract*—Reduce-Scatter operation is a well-known method in many parallel applications, particularly for using it in reduce-scatter-allgather method. However, this method yields undesirable and unnecessary pending time for many processing data. In this paper, I consider the priority of elements in reduce-scatter operation in MPI library. I design a new scheduling in which elements with higher priority process sooner and eliminate unnecessary delays to handle them. I empirically evaluate new algorithm on both Ethernet and InfiniBand network and the results show on very large messages it outperforms the naïve algorithm up to 7%.**

Keywords—MPI, Reduce-scatter, Priority-aware

I. INTRODUCTION

For the last couple of decades, Message Passing Interface (MPI) [1] has been the dominant programming model for high-performance computing (HPC) applications. The MPI Standard [21] provides primitives for various point-to-point, collective, and synchronization operations. Collective operations defined in the MPI standard offer a very convenient abstraction to implement group communication operations. Owing to their ease of use and performance portability, collective operations are widely used across various scientific domains. The MPI_Allreduce is a fundamental component of a wide range of parallel applications, such as norm calculations in iterative methods and gradient mean calculation in deep neural networks. It is arguably one of the most popular collective operations in use today. The allreduce operation consists of performing a reduction operation over values from all processes, such as a summing value or determining the maximum. Therefore, the cost of the allreduce increases with process count and motivating the need for improved performance and scalability on emerging architectures.

In this paper I design and implement a new version of Reduce-Scatter algorithm with recursive vector halving and distance doubling. The main feature of new design is considering a priority for each segment of processing data. In original Reduce-Scatter (RS) algorithm, there is no priority for data processing. Hence, it performs reduction operation on available data. It causes unnecessary and redundant pending time for the data with higher priority and finally downgrade the overall performance of original Reduce-Scatter.

The rest of the paper is organized as follows. Section II discuss the background in this work. Section III describes the main advantages along with the challenges in designing new RS algorithms optimizations, including memory requirement of new design. In Section IV, I present a priority-aware RS algorithm that reduces the unnecessary pending times. Experimental results are discussed in Section V. Finally, Section VI contains concluding remarks.

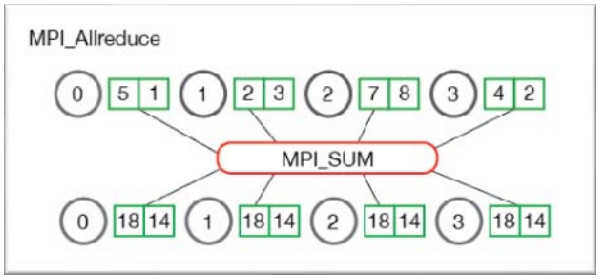

Figure 1: MPI_Allreduce operation with 4 processes

II. BACKGROUND

In this section, I disscuse about some MPI collective operations which we need to understand them before digging to main idea.

A. MPI_Allreduce

The MPI_Allreduce operates upon s sets of p values into s resulting values through operations such as summations or calculating the maximum value. These values are initially distributed evenly across p processes and results are returned to all processes. Figure 1 shows an example of MPI_Allreduce for sum operation with 4 processes.

A reduction requires $(p - 1) \cdot s$ floating point operations if the full reduction is performed on a single process. Therefore, splitting across p processes yields a lower bound of $(p-1)\cdot s/p$ floating-point operations. Furthermore, as data is distributed across all processes,

a minimum of log2 (p) messages must be communicated.

There are several Allreduce algorithms with various levels of optimality dependent on message size s and process count p. The Recursive-Doubling, Ring and Reduce-Scatter-Allgather are the most well-known Allreduce algorithms. In this paper we only focus on Reduce-Scatter-Allgather algorithm.

B. *Reduce-Scatter-Allgather*

This algorithm includes two main phases: 1- Reduce-scatter, 2- Allgather. As Figure 2 shows, processes use scatter operation to share a specific part of data with a specific pair.

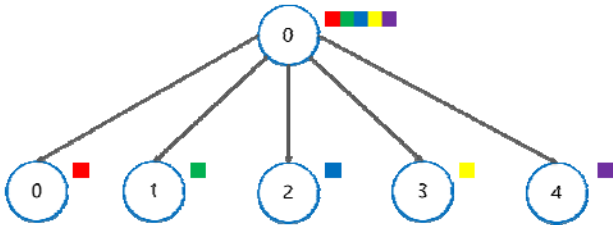

Figure 2: MPI_Scatter operation with 5 processes

In first phase, using Reduce-scatter operation, each process shares own data with other processes and receives new data from them, then it performs the operation on available date to generate partial final data. Finally in second phase, processes call Allgather operation to have all the final result in all the processes. Figure 3 shows a simplified Reduce-Scatter-Allgather algorithm for 4 processes.

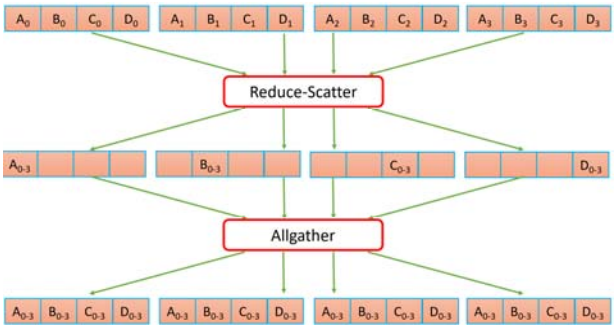

Figure 3: Reduce-Scatter-Allgather algorithm

III. PRIORITIZED REDUCE-SCATTER

In this section, I explore in details of Reduce-Scatter operation's communication pattern and show how we can improve the algorithm by changing communication scheduling in each process. Figure 4 shows the communication pattern for rank 0 which works along with 15 more ranks in reduce-scatter operation. As you can see, in this rank element A remains in the process until last step. It means we can postpone operations on this element until last step without it causes any problem for final result. In the similar way, element B remain in the process until one step before the last one. It means element B must be process sooner than element A in rank 0. In other works, element B has higher priority than element A. However, comparing elements C and D with element B, C and D have higher priority than element B. So, we can postpone the operations on element B until operations on C and D are completed. We can find the similar scheduling for other elements and also in other ranks. This gives us this idea to propose a new design of reduce-scatter method to consider priority of elements and re-schedule operations to reach minimum pending time for high priority elements. In the next subsection I discuss the advantage and challenges to fulfill this idea. In rest of paper, I use RS as abbreviation for Reduce-Scatter operation.

A. *Advantages of prioritized RS*

There are several benefits for new RS methods. First, it shortens pending time for higher priority elements. In each process, some elements must be processed sooner because other pairs need them. When we perform reduction on all available data, other pair must wait longer to complete the processing on the sender rank. By postponing the processing on data with less priority, we eliminate the pending time in each step and finally improve the overall performance.

Second, it decreases the idle time of ranks in communication phase. In original RS method (with no priority), processing units are idle when processes are sending own data and waiting to receive new data from other pairs. However, in new version (with priority), ranks can work on pending calculation. Especially when packet sizes are large, in communication phase, there is a meaningful idle time that can be utilized to perform reduction on postponed data. This benefit also improves the overall performance.

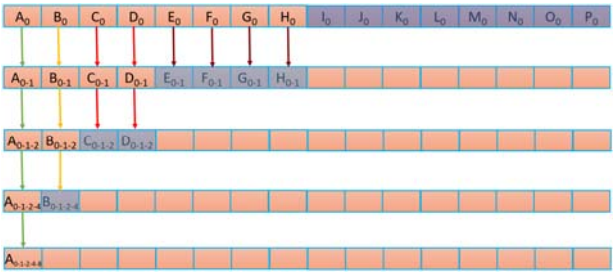

Figure 4: Reduce-Scatter communication pattern in rank 0 of 16

B. *Challenages on desgining prioritized RS*

The RS operation is very well-know MPI method, however we want to implement the features in RS that may cause some challenges for developer. I shortly discuss few of these challenges in this paper.

First, we know that postponing operations related to less priority data is beneficial, however we need to store postponed data somewhere. Otherwise storing

unprocessed data would be a big challenge and we should request extra memory which is not cost effective. In my design, it mathematically is provable that no extra memory is needed. The reason is that, in each step of RS algorithm, half of working buffer is sent (available to re-use) and only half of half received buffer must be saved. Hence, we do not need any extra memory to handle unprocessed data. Figure 5 shows how to arrange data in the user buffer to fix this issue.

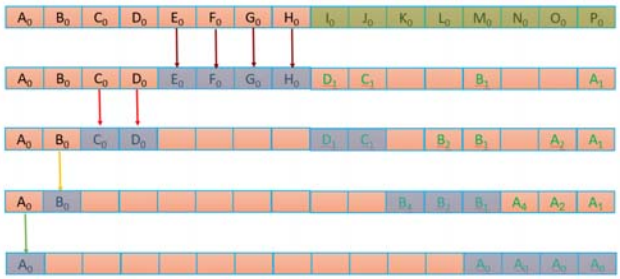

Figure 5: The arrangement of unprocessed data in Reduce-Scatter with Priority at process 0 of 16 ranks

Second challenge is that we must figure out the patter of data priority in each process. In contrast of process 0, finding the patter of data priority in rest of processes is not a straightforward approach. For example, as Figure 6 shows, not only this patter is not linear (from the element with lowest index to highest one), but also it is not necessarily in a same direction. For example, in first step it must send data from lowest to highest index (I to P). But in second step, it must process and send data from highest to lowest index (D to A). Also, in third step, it must change the sending directions again, and send data from lowest to highest index (G to H). To understand why these directions are changing in each step, one should draw other pairs data arrangements to show the priority patters in them.

Third, in prioritized RS we need to send multiple small messages instead of a large one. This imposes more latency which will reduce the performance for small and medium size message sies.

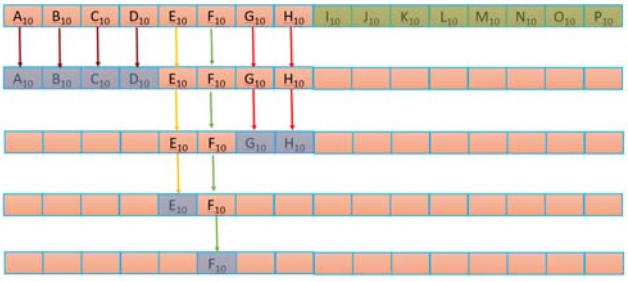

Figure 6: The priority pattern in Reduce-Scatter with Priority at process 10 of 16 ranks

C. *Prioritized method*

In this section, I discuss bout the proposed method with more details and explore how need design can improve throughput of operation. To better understanding of the main idea, I present naïve and proposed methods.

Algorithm 1 Naive Reduce-Scatter

Input: send buffer, Message size, MPI communicator
Output: Reduce_scatter result on send buffer
Initialisation :

```
1:  Rank = MPI_comm_rank;
2:  Size = MPI_comm_Size;
3:  Round = log(Size);
4:  Mask = 1;
5:  chunk_size = Msg_size ÷ 2;

6:  while Mask < Round do
7:    Dst = Rank ^ Mask;
8:    MPI_Sendrecv(SendBuf, TmpBuf, chunk_size, Dst);
9:    MPI_Oper(SendBuf, TempBuf, chunk_size, Op);
10:   chunk_size >> = 1;
11:   Mask << = 1;
12: end while
13: return SendBuf
```

Figure 7-a: Proposed algorithm for priority-aware MPI_Reduce-Scatter

Algorithm 2 Priority-aware Reduce-Scatter

Input: send buffer, Message size, MPI communicator
Output: Reduce_scatter result on send buffer
Initialisation :

```
1:  Rank = MPI_comm_rank;
2:  Size = MPI_comm_Size;
3:  Round = log(Size);
4:  Mask = 1;
5:  chunk_size = Msg_size ÷ Size;
6:  Round_size = Size ÷ 2;

7:  while Mask < Round do
8:    Dst = Rank ^ Mask;
9:    for i = 0 to Round_size do
10:     if (Mask > 1) then
11:       for i = 1 to log(Mask) do
12:         MPI_Wait(ReqRecv[]);
13:         MPI_Oper(SendBuf, TempBuf, chuk_size ,Op);
14:       end for
15:     end if
16:     MPI_Irecv(TempBuf, chunk_size, Dst, ReqRecv[]);
17:     MPI_Isend(SendBuf, chunk_size, Dst, ReqSend[]);
18:   end for
19:   Round_size >> = 1;
20:   Mask << = 1;
21: end while
22: return SendBuf
```

Figure 7-b: Proposed algorithm for priority-aware MPI_Reduce-Scatter

Figure 7-a shows the naïve approach of RS algorithm. It gets send buffer, message size and MPI communicator as input and return the final result in send buffer as output. After initializing variables, in line 8, the MPI_Sendrecv operation is called which is a blocking operation and rank must wait until all data

available in the buffer. In line 9, it performs MPI_Oper on all available data.

As Figure 7-b shows, the proposed algorithm gets send buffer, message size and MPI communicator as input and return the final result in send buffer as output. After initializing variables, in the first iteration it does not perform any operation on data, and only communicate with other pairs via non-blocking operation of MPI_Isend and MPI_Irecv. In the second iteration, it first performs operation (lines 9-15) on received data in last iteration and then move to communication part (lines 16-17). The reason is that we only want to work on the highest priority data in this iteration and do not want to be ideal for undesirable data.

Because we do not use blocking operation for communication, in line 12 we only wait for specific chunk of data which is required in this step. Also, the operation is done of a smaller size of date. All of these cause that high priority data process faster, and unnecessary pending time eliminate from its scheduling.

IV. PERFORMANCE STUDY

In this section, I first provide the information about experimental setup and cluster, and then show the collected results for implemented codes and finally present a analysis about them.

A. *Experiment setup*

The experiments were performed on a local system named Noleland cluster. Noleland is equipped with Intel Xeon Gold 6130 CPUs with 2.10 GHz frequency. Each node has 16 cores, 32 threads, and 187GB DDR4-2666 RAM. This cluster runs CentOS-7, and the underlying network is a 100 Gbps Mellanox MT28908 Infiniband. I allocated nodes manually, and the same nodes were chosen for all measurements on this cluster. All experiments reported on Noleland have p=128, N=8, and PPN=16. I implemented my proposed algorithm in the MVAPICH2-2.3.3 [3] and MPICH-3.3 [4]. The library was compiled with the default compile flags (including the -O2 flag). I have then conducted experiments with the OSU benchmark suite [5].

B. *Results on MPICH*

For running MPICH, it is connected to 10 Gbps ethernnet and run osu_allreduce benchmark. As figure 8 shows, as we expected, the proposed method can not beat niave approach because of high latency imposed in new method. However, figure 9 shows that for messages larger than 256KB, priority approach ouperforms the naïve one. As both figures show, the proposed mehod have similar trend as naïve one. Table 1 shows the overhead of priority method than naïve in varius sizes. We observe up to 10% ovhead in 256B and 16KB, and up to 7% improvement in 512KB and 8MB message sizes.

C. *Results on MVAPICH*

I also conduct the similar evaluation using MVAPICH and connect it to 100 Gbps InfiniBand network and run osu_allreduce benchmark. As we can see in figure 10, the proposed method can not beat niave approach because of high latency imposed in new method. Moreover, there is a big jump in 1KB and lager message sizes than naïve, which is mot observed in MPICH evaluation. Figure 9 shows that for messages larger than 16MB, priority approach ouperforms the naïve one. The proposed mehod has similar trend as naïve one in large message sizes. Fianally, Table 2 shows the overhead of priority method than naïve in varius sizes. There is an 308% overhead in 4KB, and up to 5% improvement in 64MB message sizes.

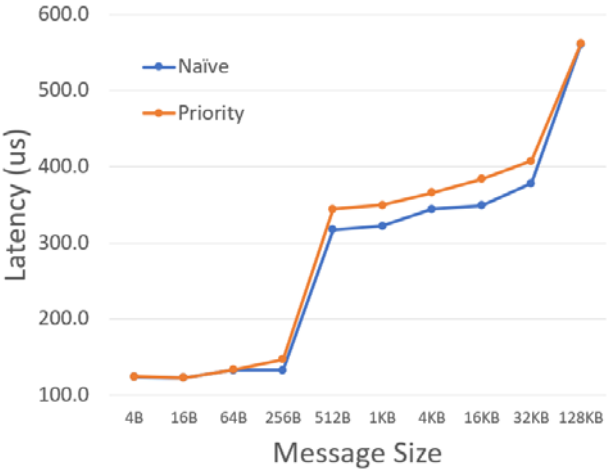

Figure 8: MPICH results on proposed algorithm for small and medium message sizes in osu_allreduce with 128 ranks, 8 nodes

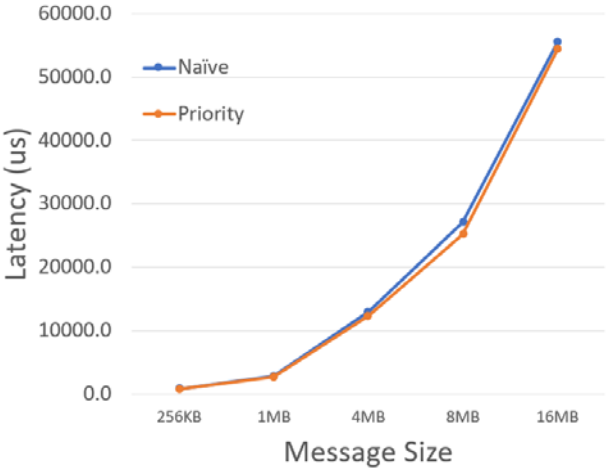

Figure 9: MPICH results on proposed algorithm for large message sizes in osu_allreduce with 128 ranks, 8 nodes

	MPICH		
	Naïve	Priority	Overhead %
4B	123.9	124.5	0.5
16B	122.8	122.6	-0.2
64B	133.0	133.2	0.2
256B	132.9	147.1	10.7
1KB	321.7	349.5	8.6
4KB	344.4	365.7	6.2
16KB	348.7	383.5	10.0
128KB	560.1	561.6	0.3
256KB	825.1	772.6	-6.4
512KB	1424.5	1322.9	-7.1
4MB	12954.1	12287.6	-5.1
8MB	27092.7	25147.3	-7.2
16MB	55519.6	54344.3	-2.1

Table 1: Overhead of proposed algorithm than naïve on MPICH for 128 ranks, 8 nodes, and 10 Gbps Ethernet

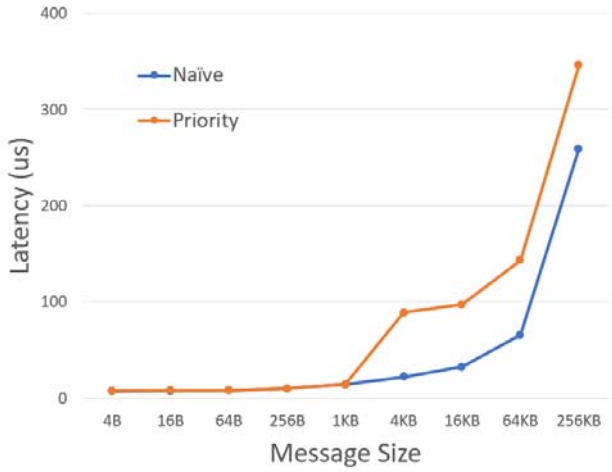

Figure 10: MVAPICH results on proposed algorithm for small and medium message sizes in osu_allreduce with 128 ranks, 8 nodes

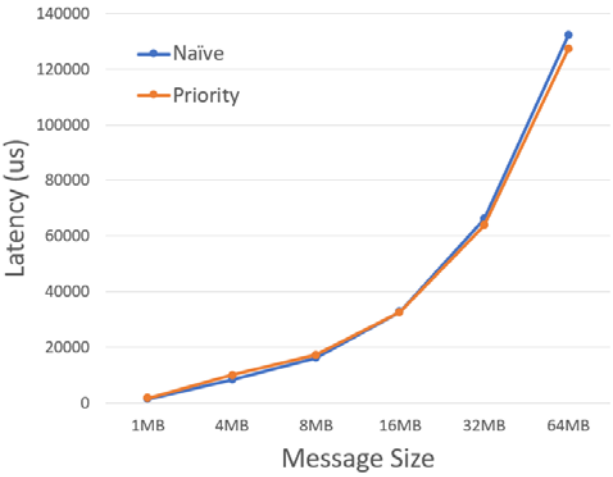

Figure 11: MVAPICH results on proposed algorithm for large message sizes in osu_allreduce with 128 ranks

	MVAPICH		
	Naïve	Priority	Overhead %
4B	7.1	7.2	2.1
16B	7.5	7.5	0.7
64B	7.6	7.7	1.0
256B	9.8	9.9	1.0
1KB	13.8	14.0	1.6
4KB	21.8	88.9	308.4
16KB	32.2	97.2	201.5
64KB	65.6	143.0	117.9
256KB	258.9	345.9	33.6
1MB	1285.8	1603.6	24.7
8MB	16141.7	17108.5	6.0
16MB	32604.5	32532.2	-0.2
32MB	66072.89	63784.81	-3.5
64MB	132259.0	125283.4	-5.3

Table 2: Overhead of proposed algorithm than naïve on MVAPICH for 128 ranks, 8 nodes, and 100 Gbps IB

V. Conclusion

In this paper, I present an optimized version of reduce-scatter algorithm to eliminate undesirable delay in processing data and shorten the critical path in all the ranks. I give the priority to each element in the ranks and generate a new communication scheduling to handle the element with higher priority faster. The proposed method imposes a high latency for small and medium size messages because of the increase in number of communication rounds in there. I implemented priority approach in MPICH and MVAPCH and observed that it outperforms up to 7% in Ethernet and 5% in IB. As future work, I am working to define a new priority for a group of elements which have the higher priority than other elements and send/receive them in a single round. This approach will not hurt the priority concept in proposed method but reduce the number of rounds and consequently reduce the communication latency.

References

[1] Message Passing Interface Forum 1994. MPI: A Message-Passing Interface Standard. Message Passing Interface Forum.

[2] MPI3 2012. MPI-3 Standard Document. http://www.mpi-forum.org/docs/mpi-3. 0/mpi30-report.pdf. (2012).

[3] MVAPICH2: High Performance MPI over InfiniBand, iWARP and RoCE. http://mvapich.cse.ohio-state.edu/, accessed Apr 9, 2021.

[4] MPICH: High-Performance Portable MPI, https://www.mpich.org, accessed Sep 9, 2020.

[5] DK Panda. OSU micro-benchmark suite, 2011.