Implementation and Analysis of MPI Primitives

Chahak Tharani, Kaustubh Khare, Abhinav Agarwal
Group AKC (P16)

1 Introduction

Message Passing Interface (MPI) is a standard for the exchange of messages between processes. MPI is one of the most significant and widely used modes of communication amongst distributed systems. Today several implementations of MPI interfaces, like Gloo and OpenMPI are available which are optimized for certain hardware and applications.

The performance of HPC (High-performance computing) and distributed applications is dominated by these communication primitives. Multiple widely used frameworks and applications like PyTorch [6], TensorFlow [1] depend on MPI for their intracluster communication. TeraSort [5] is one of the most popular map-reduce sort used in distributed systems which makes use of MPI primitives. Considering such widespread usage of MPI primitives, having a high-performance implementation and in-depth understanding of these operations is of vital importance.

Some popular features of MPI are the collective communication primitives like broadcast, (all) reduce, (all) scatter, and (all) gather which are used for communication amongst a group of processes. Each of these primitives could be implemented in a variety of ways using different algorithms. For example, All_Reduce could be implemented using $TreeReduction$ or $Gather-Reduce-Scatter$ which will lead to different performances. $TreeReduction$ would be better if the data size is huge whereas $Gather$, $Reduce$, and $Scatter$ would be better for smaller data that could be handled by a single node. Thus, depending on the use case it is important to take into consideration the implementation to use and the underlying algorithm which the primitive uses.

In our work, we focus on developing efficient and optimized implementation of two of these communication primitives while comparing the different frameworks, various algorithmic implementations of each primitive with varying workloads on different topologies. We implement reduce and broadcast primitives from scratch in C++ using only basic send and receive function calls on Gloo and OpenMPI and evaluate them on ring and mesh topology. We compare our implementation against standard library primitives and provide an analysis for the difference in the performance. We perform a mathematical evaluation of different algorithms that we have implemented using the $\alpha-\beta$ cost model.

2 Background

2.1 Cost Model

We use the $\alpha-\beta$ cost model to estimate the cost of different collective communication primitives in terms of latency and bandwidth.
The time taken to send a message between two nodes A and B is approximated as $O(\alpha+\beta n)$ (shown in fig. 1) where α is the latency term (startup time) which is independent of message size n and β is the bandwidth cost per byte of message.

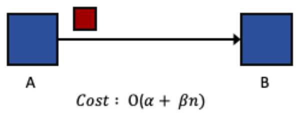

Figure 1: Sending a message of n bytes from A to B

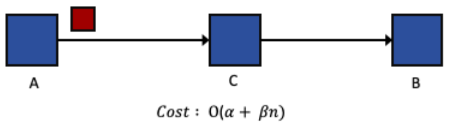

Figure 2: Sending message from A to B through C

The communication links are full duplex and the cost remains the same if a message is sent and received on the same link simultaneously.

If k messages are transferred in the same direction on a link, the cost of communication increases to $O(\alpha+k\beta n)$ due to link congestion.

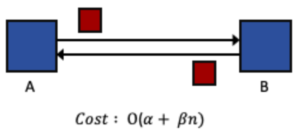

Figure 3: Sending message from A to B and B to A concurrently

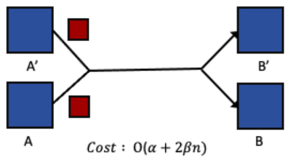

Figure 4: Communication with link congestion

2.2 Network Topologies

Number of links (**λ**) is the total number of connections in the network between any two pair of nodes. A network with higher number of links is more expensive to build and maintain.

Bisection bandwidth (**B**) is the bandwidth available between two partitions in the network. It gives true estimate of the available bandwidth in the system. Some topologies can have a large bisection bandwidth but at the cost of high number of links which increases the overall cost and complexity of the system.

Diameter (**Δ**) is the length of the maximum shortest path between any two nodes in the network. A network with shorter diameter will have lower latency for point-to-point communication. Below we discuss a few important network topologies with their bisection bandwidth, diameter and the number of links.

- **Ring topology** - All nodes are connected in circular ring fashion. This topology has $n - 1$ number of links, diameter of $n - 1$, and bisection width of 2.

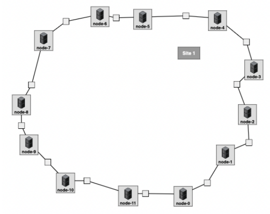

Figure 5: Ring Topology

- **2-D mesh** - Nodes are laid out in 2-D grid. This topology has $O(2n)$ number of links, $2(\sqrt{n} - 1)$ diameter, and bisection width of $O(\sqrt{n})$.

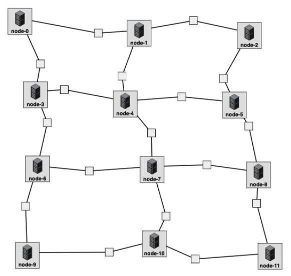

Figure 6: Mesh Topology

3 Related Work

Since the nineties, many algorithmic implementations and frameworks for MPI primitives have evolved for different architectures. Early work on collective communication has focused on communications patterns like ring, tree, etc. One relevant example is performing communication in a logarithmic fashion to minimize latency. Thakur et. al. [11] perform recursive-doubling for allgather primitive and devise different algorithms depending on whether the reduction operation is commutative or non-commutative. For reduce-scatter primitive, they implement recursive-halving for commutative operations and recursive-doubling for non-commutative ones. Geijn et. al. [9] devise hybrid collective communication algorithms for varying input sizes for mesh architectures. These works are limited in scope of a particular framework and specific network topologies.

A comparative study by Sinha et. al. [10] focuses on the comparison of broadcast and point-to-point communication with a data size of few KBs. This neither covers the other collective MPI primitives nor does it compare the results with different data sizes. Another comparison of Cray MPI and OpenMPI done by Graham et. al. [3] provides a detailed view of point-to-point primitives and only one collective communication primitive ((all) reduce).

Other studies aim at optimizing collective communication operations for gradient synchronization in distributed training of data-parallel deep learning models. Castello et. al. [2] implemented a pipelined optimization with nonblocking reduce primitive to achieve better overlap of communication with computation. This approach is widely used in deep learning frameworks but is specific to ML workloads. Research has also been performed to evaluate the quality of implementation by measuring how close it is to the cost of optimal point-to-point communication [8].

Further studies by Hoefler et. al. [4] and by Liu et. al. [7] propose new and improved methods for implementing broadcast. However, neither of them gives a holistic and practical comparison of different implementations of MPI primitives either individually or on real-world examples.

Previous comparison surveys have focused on individual aspects like data size, different backends, and various algorithmic implementations. However, there is a dearth of holistic comparative studies. Our project provides a comprehensive analysis and comparison of the various algorithmic implementations, on different network topologies and backends for varying workload sizes for primary collectives. Our study also attempts to make performance improvements over multiple MPI interface libraries like Gloo and OpenMPI.

4 Design

4.1 Algorithms

We have analyzed different MPI primitives and their cost models. In our project, we implement **Reduce** and **Broadcast** primitives using different algorithms. Below is the description of the algorithmic implementations we have developed in this project.

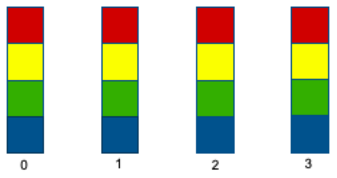

Figure 7: Initial configuration for communication

- **Rabenseifner's Reduce**: This is performed by Reduce-Scatter followed by Gather as shown in fig. 8 and 9. The total communication cost is $O(\alpha logP + \beta n)$. It provides better performance for long messages as the cost becomes optimal in terms of both bandwidth and latency.

- **Tree reduce**: There will be $O(logP)$ rounds of communication (fig. 10 and 11), each with a communication cost of $O(\alpha + \beta n)$. The overall cost of this algorithm is $O(\alpha logP + \beta nlogP)$. We notice that while this algorithm is optimal in latency (α term), it is not bandwidth optimal.

- **Tree broadcast**: Similar to tree reduce, the cost of this algorithm is $O(\alpha logP + \beta nlogP)$, which is optimal in the latency term but not in the bandwidth term (fig. 12 and 13).

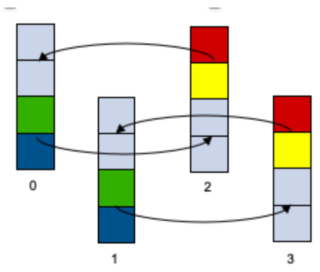

Figure 8: Rabenseifner's Reduce Scatter

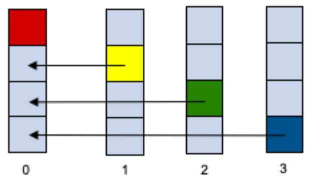

Figure 9: Rabenseifner's Reduce Gather

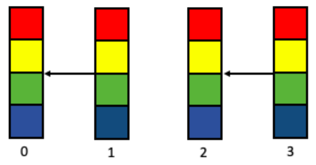

Figure 10: Tree Reduce first round

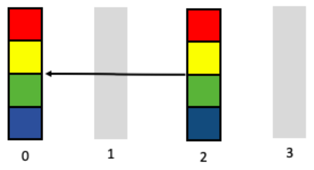

Figure 11: Tree Reduce last round

- **SAG broadcast**: This is implemented as scatter (fig. 14) followed by all-gather. The cost of the scatter phase is $O(\alpha logP + \beta n)$. This is followed by a ring all-gather (shown in fig. 15 and 16) which has a cost of $O(\alpha P + \beta n)$. Hence, the total cost for SAG Broadcast becomes $O(\alpha P + \beta n)$. This is bandwidth optimal but not latency optimal.

5 Experiments

We implemented the broadcast and reduce primitives from scratch in C++ (using only basic send and receive calls). We have used Cloudlab machines with Ubuntu 18.04.1 (Linux kernel 4.15.0), GCC version 7.5.0, Open-

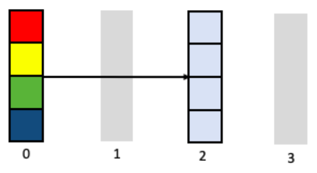

Figure 12: Tree Broadcast first round

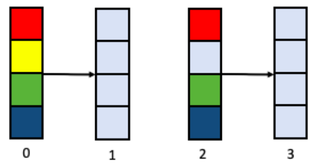

Figure 13: Tree Broadcast last round

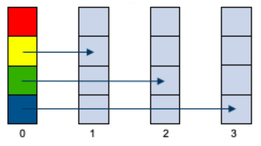

Figure 14: SAG Broadcast Scatter Phase

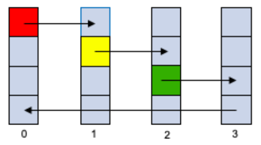

Figure 15: SAG Broadcast Gather first round

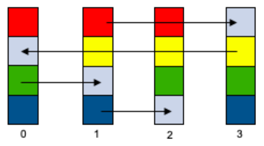

Figure 16: SAG Broadcast Gather last round

MPI version 4.1.1, Gloo head from github at commit $9c1876a$ running on Intel Xeon CPU E5-2630 v3 with 64GB RAM per node. We implemented the following algorithms in both Gloo and MPI:

- Broadcast
  1. Tree Broadcast
  2. Scatter All-Gather (SAG) Broadcast
- Reduce
  1. Tree Reduce
  2. Rabenseifner's Reduce

We compared these algorithms against each other as well as against the standard library implementations in Gloo and MPI. We experimented and contrasted their performance on Ring and Mesh network topologies. We analyzed the communication cost on int arrays with sizes ranging from 8 to 140k. We used Cloudlab's profiles to create the different network topologies as shown in fig 5 and 6. Before running any experiment we run the primitive 10-15 times to warm up the network and system caches and then collect the median time after 1000 iterations.

```
init()
warmup()

maxDurations = []
for i in 0 -> 1000
    startTime = getTime()
    run_test_primitive()
    endTime = getTime()
    // Barrier to wait for all procs
    Barrier()
    duration = endTime - startTime
    // Gather times at rank 0
    allDuration = gatherDurationsFromAll()
    maxDuration = max(allDurations)
    maxDurations.add(maxDuration)

timeTaken = median(maxDurations)
```

6 Results and Evaluation

We have focused on creating high-performance minimal implementations of the proposed primitives in C++ that matches or beats the standard library implementations in some cases. While our implementations are not as rich in features as the standard library, we demonstrate that by careful and performance oriented programming, we can implement efficient implementations of MPI primitives.

6.1 Broadcast Primitive

We compare tree and SAG implementation of broadcast primitive on Gloo and OpenMPI and evaluate them on mesh and ring topology. We expect SAG broadcast to perform better than tree broadcast for large vector sizes because it is optimal in the bandwidth term while tree broadcast is optimal in the latency term.

6.1.1 Performance on Gloo

From fig. 20 we see that for smaller vector sizes, tree broadcast is better while SAG broadcast is better for larger vector sizes as expected. We also see that our tree broadcast performance closely follows standard Gloo library broadcast as both of them use the same tree algorithm.

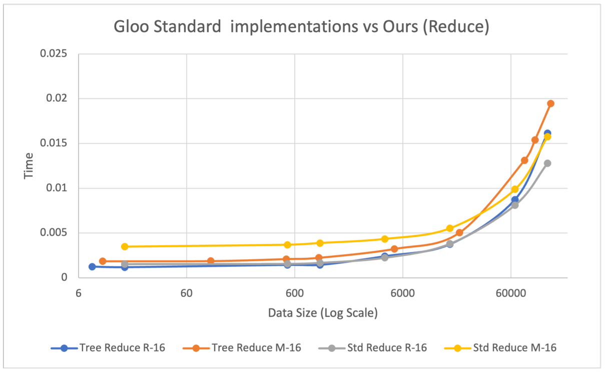

Figure 17: Std. Gloo Reduce vs Our Implemenation

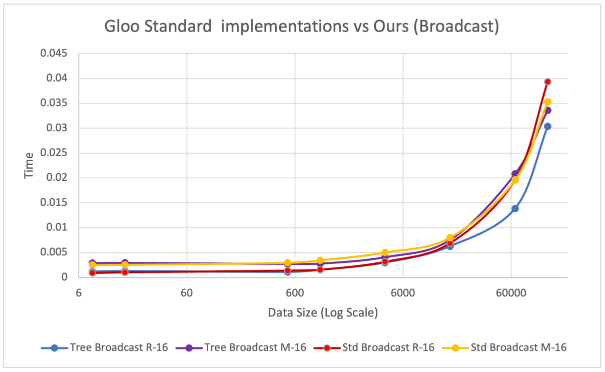

Figure 18: Std. Gloo Broadcast vs Our Implemenation

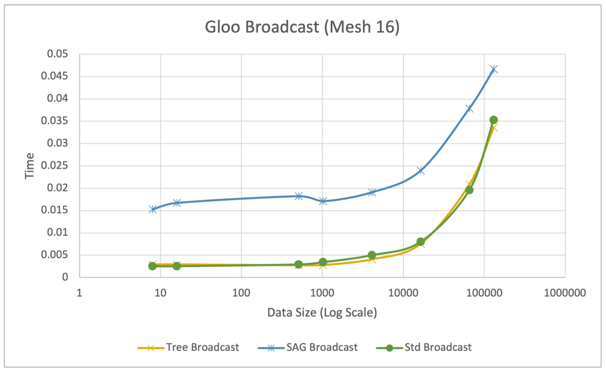

Figure 19: Gloo Broadcast on 16-Mesh Topology

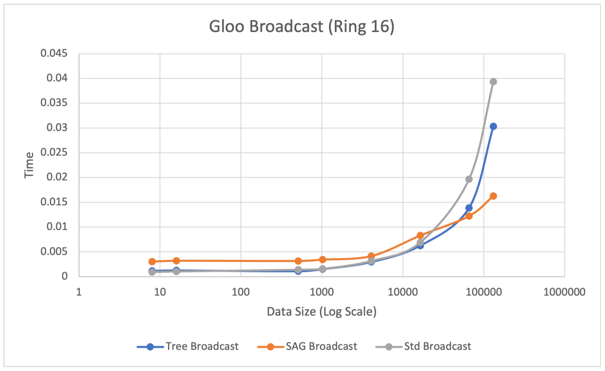

Figure 20: Gloo Broadcast on 16-Ring Topology

6.1.2 Performance on OpenMPI

From fig 21, we see similar results from OpenMPI as Gloo. Our implementation is able to slightly outperform the standard implementation at larger vector sizes which should be because our implementation does not do strict asserts checks as in standard library and also caches temporary buffers to avoid malloc/delete calls in subsequent primitive invocations.

6.1.3 Ring vs Mesh topology

Fig. 31 shows the comparative performance of tree broadcast on Gloo for Ring vs Mesh of size 16. We observe that broadcast performs better on the ring topology than mesh, which is counter-intuitive as mesh has lower diameter and higher bisection bandwidth. We suspect that this might be due to the layout of nodes in the mesh.

6.1.4 Gloo vs OpenMPI

Fig. 22 shows the comparative performance of tree and standard broadcast on OpenMPI vs Gloo backends. We see that OpenMPI performs slightly better than Gloo using both standard library and our custom implementation on larger vector sizes while the difference is not significant on small vector sizes.

6.1.5 Impact of increasing number of nodes

To measure the effect of distributing the data over larger number of nodes, we evaluate the performance impact of broadcast by doubling the number of nodes while halving the the data per node to keep total data size constant. Fig 24 and 23 shows comparative performance analysis for this. We see that for larger vector sizes, ring-16 has better performance than ring-8. The cost for ring-8 vs ring-16 will be as follows:

$$C_{tree-r8} = \alpha logP + n\beta logP$$

$$C_{tree-r16} = \alpha log2P + (n/2)\beta log2P$$

6.2 Reduce Primitive

We compare Tree and Rabenseifner implementation of reduce primitive on Gloo and OpenMPI and evaluate them on mesh and ring topology. We expect Rabenseifner reduce to perform better than tree for large vector

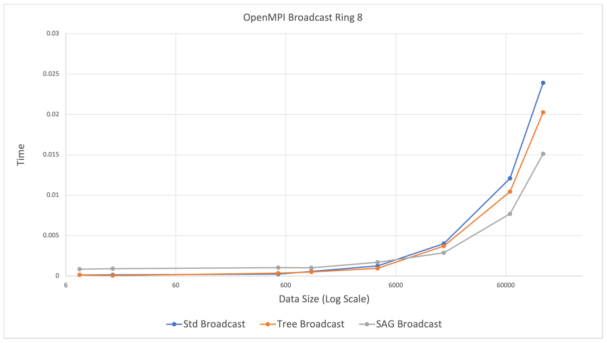

Figure 21: OpenMPI Broadcast on 16-Ring Topology

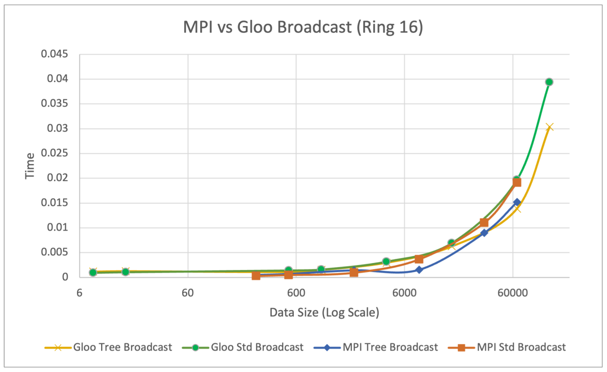

Figure 22: OpenMPI vs Gloo Broadcast on 16-Ring Topology

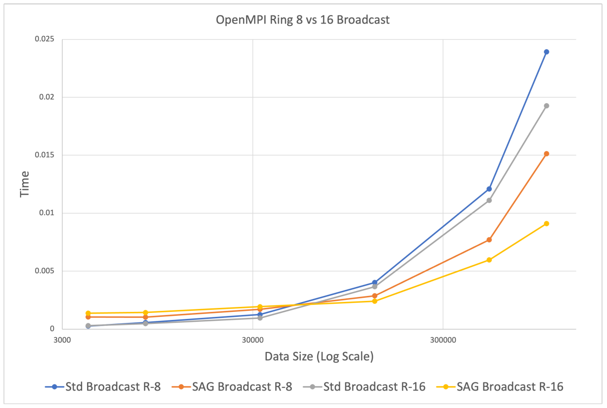

Figure 23: OpenMPI Broadcast : 8-Ring vs 16-Ring

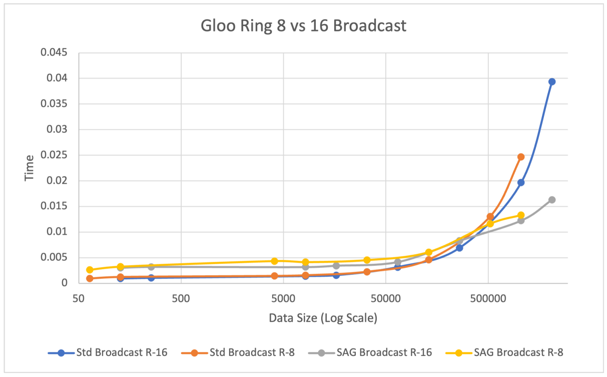

Figure 24: Gloo Broadcast : 8-Ring vs 16-Ring

sizes because it has optimal cost but with a large constant factor. Standard ring reduce of Gloo had very bad performance, so we implemented Rabenseifner using standard reduce scatter and gather. Hereinafter, we refer to this as standard Gloo reduce.

6.2.1 Performance on Gloo

From fig. 26 we see our Rabenseifner's implementation is worse than tree reduce, which is pretty close to standard libraries performance. This could be due to some implementation bottleneck in our Rabenseifner's implementation.

6.2.2 Performance on OpenMPI

From fig. 28 we see that all three implementations have similar performance for smaller vector sizes while Rabenseifner has worse performance for bigger sizes. We suspect the reason is same as mentioned for Gloo.

6.2.3 Ring vs Mesh topology

Fig. 32 shows the graphs for tree reduce on both ring and mesh topologies with 16 nodes. As seen in the figure, reduce performs better on the ring topology than mesh, which is counter-intuitive as mesh has lower diameter and higher bisection bandwidth. We suspect that this might be due to the layout of nodes in the mesh.

6.2.4 Gloo vs OpenMPI

Fig. 27 shows that OpenMPI and Gloo have similar performance and outperform each other at certain points. Looking at the trend, it seems that OpenMPI might be better for larger vector sizes.

6.2.5 Impact of increasing number of nodes

From fig. 29, we see that similar to broadcast, increasing the number of nodes leads to better performance for larger vector sizes.

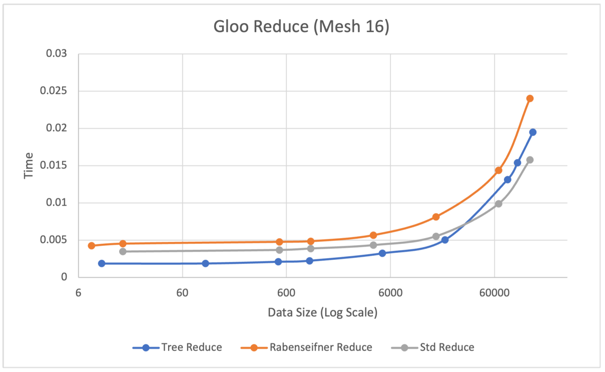

Figure 25: Gloo Reduce on 16-Mesh Topology

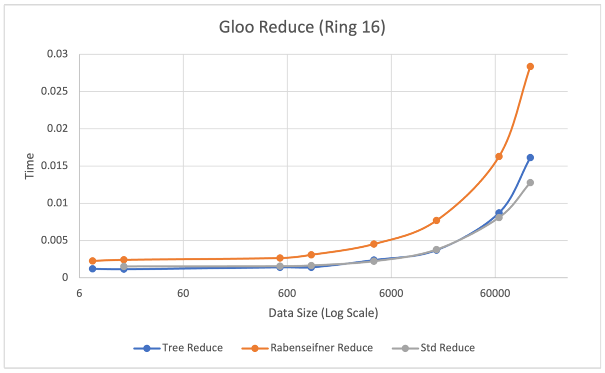

Figure 26: Gloo Reduce on 16-Ring Topology

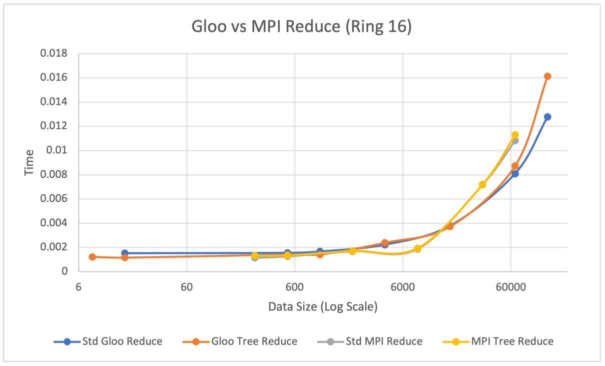

Figure 27: OpenMPI vs Gloo Reduce on 16-Ring Topology

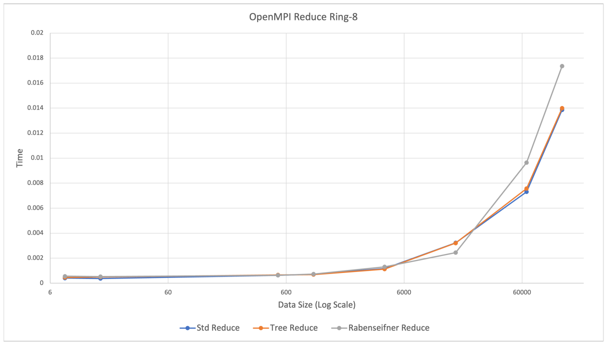

Figure 28: OpenMPI Reduce on 8-Ring Topology

6.3 Takeaways

- SAG broadcast is faster than tree broadcast for larger vector sizes while tree broadcast is faster for smaller sizes. This is because SAG broadcast is optimal in bandwidth term while tree broadcast is optimal in latency term.
- Rabenseifner reduce is expected to be faster than tree reduce for larger vector size, but we were unable to see this in our evaluations.
- We saw worse performance on mesh compared to ring, which was not expected. Also, we were unable to run OpenMPI on Mesh topology due to multi-interface binding errors in the library.
- OpenMPI is almost always faster than Gloo.
- Distributing data over larger number of nodes helps in improving the performance for larger input sizes.
- We show that with careful and specific implementation, we can beat the standard library implementations of MPI primitives.

7 Future work

- Cache temporary buffers more effectively to reduce time spent in memory management after the first call to the primitive.
- Use multi-threading to send-receive data. This should be beneficial especially for large sizes.
- Do a similar analysis on more complicated topologies like $d-$dimensional Torus and Hypercubes.

8 Individual Contributions

This project has been a great learning opportunity for all of us. Browsing through the library code and studying the standard implementations of some of the primitives was enlightening. We learned a lot of smart coding and implementation techniques used for executing various functionalities. Debugging and working on distributed programs was more challenging than we anticipated, but we managed to make significant progress by careful task divisions and collaboration among all the team members. While this project has been a collective team effort, we

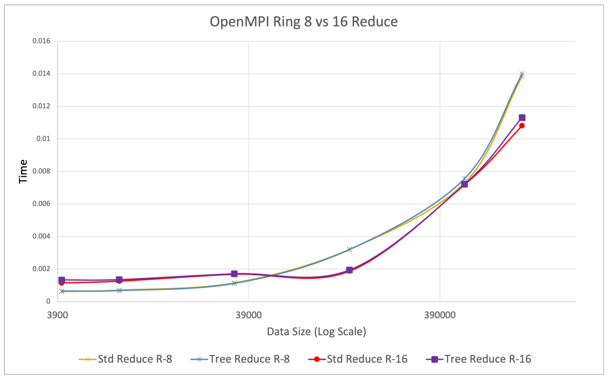

Figure 29: OpenMPI Reduce : 8-Ring vs 16-Ring

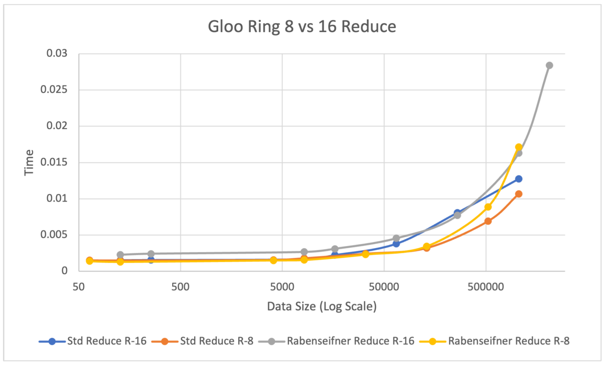

Figure 30: Gloo Reduce : 8-Ring vs 16-Ring

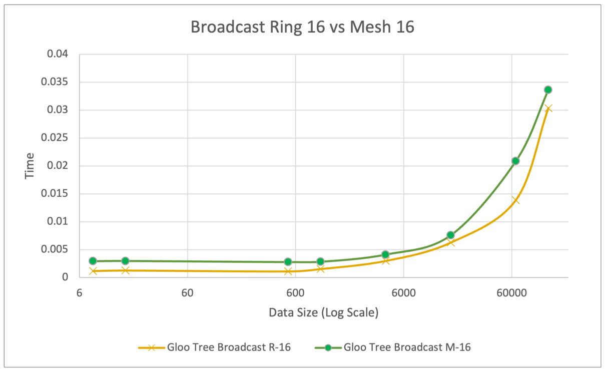

Figure 31: Broadcast: 16-Ring vs 16-Mesh

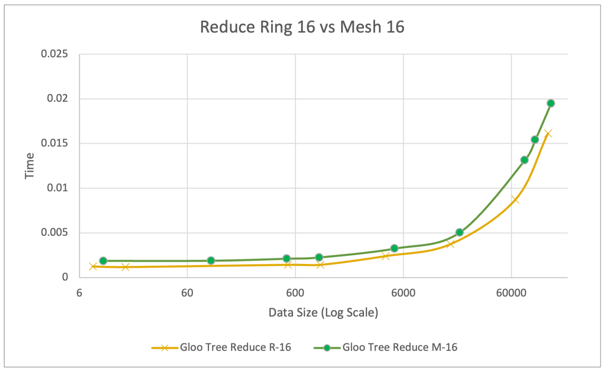

Figure 32: Reduce: 16-Ring vs 16-Mesh

note the individual contributions of the team members as follows:

- Chahak - implemented Rabenseifner's reduce and SAG broadcast algorithm in Gloo and Tree reduce on OpenMPI. She also implemented common framework to collect and analyse statistics for all algorithms that we have presented in this report and benchmarked standard library implementations.

- Kaustubh - implemented Tree broadcast in Gloo, SAG broadcast and Tree reduce in OpenMPI. He also managed the whole cloudlab infrastructure including creating different network topologies and automation scripts to quickly install the setup and get evaluation results.

- Abhinav - implemented Tree reduce in Gloo and Rabenseifner's reduce in OpenMPI. He created C++ project setup in CMake, did mathematical $\alpha-\beta$ cost model analysis of different algorithms, helped other team members in troubleshooting C++ and other issues, and coordinated all team meetings.

For the report and poster, Chahak generated all the final evaluation graphs and wrote a detailed analysis and discussion for them. Kaustubh worked on creating system diagrams and explaining the background and design of our project. Abhinav worked on introduction and related work section along with formatting and proofreading the charts, diagrams, and text in the poster and report.

Finally, we would like to thank the TA Yien Xu and Prof. Shivaram Venkataraman for helping us setup clouldlab infrastructure and providing valuable feedback throughout the project.

9 Final Remarks

This project has been a great learning experience for all of us and we feel that such collaborative projects are a great way of learning which brings out creativity and teaches the value of teamwork. All the members of our group have learned a lot working with each other, which we believe, will be valuable beyond this course as well.

References

[1] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, et al. Tensorflow: A system for large-scale machine learning. In *12th {USENIX} symposium on operating systems design and implementation ({OSDI} 16)*, pages 265–283, 2016.

[2] J. D. Adria'n Castello, Enrique S. Quintana-Ortı. Accelerating distributed deep neural network training with pipelined mpi allreduce. *Cray Users Group (CUG'07)*, 2021.

[3] R. L. Graham, G. Bosilca, and J. Pješivac-Grbovic. A comparison of application performance using open mpi and cray mpi. *Cray Users Group (CUG'07)*, 2007.

[4] T. Hoefler, C. Siebert, and W. Rehm. A practically constant-time mpi broadcast algorithm for large-scale infiniband clusters with multicast. In *2007 IEEE International Parallel and Distributed Processing Symposium*, pages 1–8. IEEE, 2007.

[5] S. Li, S. Supittayapornpong, M. A. Maddah-Ali, and S. Avestimehr. Coded terasort. In *2017 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 389–398. IEEE, 2017.

[6] S. Li, Y. Zhao, R. Varma, O. Salpekar, P. Noordhuis, T. Li, A. Paszke, J. Smith, B. Vaughan, P. Damania, et al. Pytorch distributed: Experiences on accelerating data parallel training. *arXiv preprint arXiv:2006.15704*, 2020.

[7] J. Liu, A. R. Mamidala, and D. K. Panda. Fast and scalable mpi-level broadcast using infiniband's hardware multicast support. In *18th International Parallel and Distributed Processing Symposium, 2004. Proceedings.*, page 10. IEEE, 2004.

[8] R. A. v. d. G. Mohak Shroff. Collmark: Mpi collective communication benchmark. *International Conference on Supercomputing 2000*, 1999.

[9] R. A. v. d. G. Prasenjit Mitra. Fast collective communication libraries, please. *International Conference on Supercomputing 2000*, 1999.

[10] A. Sinha, N. Das, and A. Unit. A comparative study of the mpi communication primitives on a cluster. In *IEEE International Conference on High Performing Computing*. Citeseer, 2004.

[11] R. Thakur and W. Gropp. Improving the performance of collective operations in mpich. *Cray Users Group (CUG'07)*, 2007.