OPTIMIZATION OF DISTRIBUTED SUM OF OUTER PRODUCTS (DSOP)

David Enderlin, Noé Heim, Roy Schubiger, Elwin Stephan, Patrick Ziegler

Department of Computer Science
ETH Zürich
Zürich, Switzerland

1. ABSTRACT

Computing the outer products of pairs of vectors, each pair stored on different nodes, and distributing their sum lies at the core of optimizing distributed deep learning. This paper proposes the *Generalized Rabenseifner* for distributed sum of outer products (DSOP), an algorithm that computes the DSOP using multiple participants. The proposed algorithm can be used to optimize distributed training of deep neural networks using a data-parallel approach by replacing the gradient averaging, typically using an *allreduce*, with the proposed algorithm. Compared to a baseline allreduce algorithm, our *Generalized Rabenseifner* algorithm achieves a median speedup of up to 3x.

2. INTRODUCTION

Machine Learning is rapidly developing into an essential tool in modern systems. Deep Learning is a popular model used in modern machine learning systems. Optimizing the training of deep neural networks becomes an essential aspect as datasets become increasingly larger. A popular approach to accelerate training deep neural networks is using a data-parallel approach. This approach replicates the neural network across multiple computing nodes. The individual nodes can compute the forward and backward propagation steps independently. However, there needs to be a synchronization step to synchronize the gradients across the individual nodes. This synchronization step induces an allreduce operation. [1]

Recent work by Bakker et al. showed that the gradient of feedforward and recurrent neural networks exhibit an outer product derivative structure in [2]. This project aims to exploit this structure by studying the problem of the DSOP. Algorithms that solve the DSOP can then be used as a component to synchronize gradients across multiple nodes in distributed deep learning.

In the DSOP, there are P hosts $p_1, ..., p_P$, each host p_i having two vectors a_i, b_i. They engage in a computation such that all of the P hosts end up with a local copy of the matrix G:

$$G = \sum_{i=1}^{P} a_i \cdot b_i^\top \tag{1}$$

This project aims to evaluate, design, and implement different efficient algorithms to compute G. First, we evaluate different implementations using procedures provided in MPI to compute the DSOP efficiently. Furthermore, we propose a novel algorithm to solve the DSOP, which optimizes bandwidth and performance. Moreover, we identify extensions to the proposed algorithm that can take the network heterogeneity into account and address the sparsity of inputs. All source code and benchmarks results are publicly available on GitHub[1].

3. RELATED WORK

Deep Learning has become an integral part of modern science. Improving the training time of deep neural networks has become increasingly important due to the growth of datasets, the complexity of problems they solve, and network sizes. Ben-nun and Hoefler provide a good overview of current and future parallelization strategies in [1].

The demand for distributed training of deep neural networks has led to various systems that leverage the message passing interface (MPI). For example, Horovod [3] introduces an open-source library that uses ring reduction to enable distributed training in TensorFlow. Vishnu et al. [4] also propose a modification to TensorFlow to enable TensorFlow on clusters. In addition, there is a myriad of papers that try to apply MPI to distributed machine learning [4, 5, 6].

An approach for data-parallel training at scale is provided in Blink [7], a collective communication library that tries to optimize communication in the face of hardware heterogeneity. A further approach is provided in Nguyen et al. [8]: The authors propose a sparse-ring-ring allreduce algorithm, which uses topology-aware and data compression techniques.

[1] https://github.com/elwin/dphpc

4. BACKGROUND

One can group algorithms that compute the DSOP into two groups: allgather and allreduce, as illustrated in Figure 1. The most important collective operations in MPI used in these algorithms are allgather, broadcast, scatter, allreduce, and reduce-scatter. The following paragraphs describe the most important algorithms used in allgather and allreduce and present the cost model used to model our algorithms.

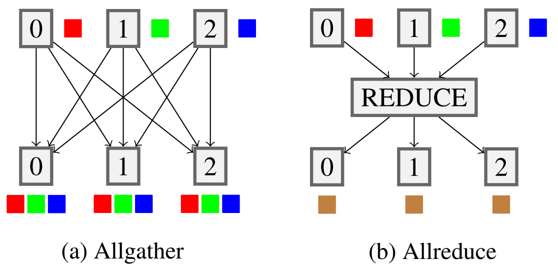

Fig. 1: Illustration of the most commonly used collective operations.

4.1. Allgather and Allreduce

Allgather represents an operation where each participating process contributes some data and receives the data contributed by all other processes. This notion of allgather suggests that an allgather algorithm that computes the DSOP contains a step where the data from all processes are distributed to every other process. The most important algorithms used for implementing allgather in Open MPI are Bruck [9], recursive doubling, neighbor, and the ring algorithm. [10, 11, 12]

Compared to allgather, the allreduce operation is composed of data movement and incorporates an operation executed on the data. The canonical operation fitting the setting of computing the DSOP is addition. The most important algorithms used to implement the allreduce operation in Open MPI are recursive doubling, the ring algorithm, and Rabenseifner's algorithm. [13, 10, 11, 14]

Multiple algorithms are required to implement collective communication operations in MPI efficiently. Different algorithms provide different tradeoffs. Therefore, one has to decide on an algorithm based on the amount of data movement and the number of processes in the collective communication operation. Open MPI uses a decision tree to select a suitable allreduce or allgather algorithm based on the total number of nodes, the size of the data, and, for allreduce, whether the reduction operation is commutative [10, 11]. An extension to hierarchical networks introduces further complexity in choosing and designing such algorithms. [15]

4.2. Cost Model

The cost model under which we evaluate our algorithms considers three dimensions: communication cost, the number of additions and multiplications per node. The network communication is modeled using the LogGP model. The addition and multiplication is indicated by the number of floating-point additions/multiplications performed in a node. With these three entities, the cost model can capture the tradeoff that the individual approaches incur when computing the DSOP.

The LogGP model uses the following standard definitions to model the communication: latency L, host overhead o for sending and receiving, the gap g between messages, number of processes P, and the gap-per-byte G (inverse of the bandwidth). We simplify the model by assuming that $g < o$. Furthermore, due to the problem at hand, the amount of data being transferred depends on the input sizes N and M, and the algorithms used in the MPI-collective operations depend on the decision tree used in Open MPI 4.0.2, as can be seen in [10].

5. IMPLEMENTATION

This section presents all algorithms we implemented that are relevant to the results. Other approaches that turned out to be less effective than the baseline were left out in the interest of brevity. We asymptotically model their costs alongside the algorithms using the previously described cost model.

5.1. Basic Implementation

As a first step, we present our baseline algorithms for solving the DSOP problem based on allgather and allreduce.

In our basic *allgather* algorithm, all of the P processes first distribute their two input vectors to all other processes using `MPI_Allgather`. Then, after a process has received all vectors from the other nodes, it will compute the outer product $G_i = a_i \cdot b_i^\top$ for each received vector pair and sum up all the computed outer products to get the final matrix G. The cost for this baseline implementation using allgather can be modeled as follows:

$$\begin{array}{rl} \text{Communication} & \mathcal{O}\left(P \cdot (o + L + (N+M) \cdot G)\right) \\ \text{Addition} & \mathcal{O}(P \cdot NM) \\ \text{Multiplication} & \mathcal{O}(P \cdot NM) \end{array}$$

The communication part is entirely determined by the `MPI_Allgather` implementation. Here, we considered the neighbor exchange algorithm implemented in Open MPI as described in [12]. Due to the chosen input parameters (vector size and the number of processes), this is the algorithm Open MPI selects for all allgather operations in our benchmarks. [10]

Our other baseline algorithm is based on an allreduce approach. All P processes first calculate the outer product $G_i = a_i \cdot b_i^\top$ of their vector pair and then use a single `MPI_Allreduce` to compute the sum of all matrices on all nodes.

The allreduce implementation used by Open MPI in all our benchmarks is a segmented ring allreduce. The chosen allreduce implementation is responsible for both the network cost and the cost of adding all intermediate matrices. The multiplications are due to the initial outer-product calculation. Here, another parameter S is introduced, representing the number of elements in each segment. The segment size used by Open MPI is 1MB and thus $S = 2^{17}$. [10]

$$\begin{aligned} \text{Communication} \quad & \mathcal{O}\big(\tfrac{NM}{S}\,(L + o + S \cdot G) \\ & \quad + P\,\big(L + o + \tfrac{NM}{P} \cdot G\big)\big) \\ \text{Addition} \quad & \mathcal{O}(NM) \\ \text{Multiplication} \quad & \mathcal{O}(NM) \end{aligned}$$

The cost models already show the main difference between the allgather and allreduce approach. An allgather approach performs more local computations while transferring fewer data over the network. In contrast, an allreduce approach trades fewer local computations for larger network transfers and more communication.

5.2. Allreduce Ring

One of the better performing algorithms is the *allreduce-ring*. First, all of the P processes compute the local outer product $G_i = a_i \cdot b_i^\top$ of their vectors, similar to the *allreduce* baseline, but instead of using the Open MPI implementation `MPI_Allreduce`, the algorithm uses a custom reduction algorithm using a ring structure.

The algorithm divides the local matrix into P chunks, and each process sends a different chunk to the next process in the ring. Each process then receives a chunk from the previous process, adds its matching chunk, and sends it to the next process. After $(P - 1)$ rounds, each process possesses one chunk of the final result matrix. An allgather using the ring structure concludes the algorithm by each process sending the final chunk through the ring and forwarding any received chunks. This takes another $(P-1)$ steps, after which all processes possess the full final matrix G.

This is a reimplementation of the ring allreduce algorithm in Open MPI. We present it here because it performs better than using `MPI_Allreduce` for the allreduce operation in most cases. We discuss this in Section 7.

$$\begin{aligned} \text{Communication} \quad & \mathcal{O}\big(P\,\big(o + L + \tfrac{NM}{P} \cdot G\big)\big) \\ \text{Addition} \quad & \mathcal{O}(NM) \\ \text{Multiplication} \quad & \mathcal{O}(NM) \end{aligned}$$

The communication cost is only asymptotic, but in reality, there is a constant factor of 2 because the algorithm performs the $(P - 1)$ rounds twice, once for calculating a chunk for the final matrix and once for exchanging those final chunks.

5.3. Generalized Rabenseifner for DSOP

The Open MPI *allreduce-rabenseifner* algorithm is a combination of a reduce-scatter implemented with recursive vector halving and recursive distance doubling, followed by an allgather implemented with recursive doubling (butterfly). [14]

This algorithm is the basis for the proposed algorithm we call the *Generalized Rabenseifner*. *Rabenseifner's* first butterfly is replaced with an allgather round to distribute all vectors to each process. Then, each process calculates its local chunk of the final matrix, namely a continuous region of the final matrix stored in row-major order. This partitioning can be achieved by partitioning the rows into contiguous rows and assigning them to individual processes. Then, using an `MPI_Allgather`, all the final chunks are distributed, such that in the end, each process has a local copy of the complete final matrix G.

$$\begin{aligned} \text{Communication} \quad & \mathcal{O}\big(P \cdot (o + L + (N + M) \cdot G) \\ & \quad + P \cdot \big(o + L + \tfrac{NM}{P} \cdot G\big)\big) \\ \text{Addition} \quad & \mathcal{O}(NM) \\ \text{Multiplication} \quad & \mathcal{O}(NM) \end{aligned}$$

The communication cost of the *Generalized Rabenseifner* is based on the underlying allgather implementation by Open MPI, which uses the neighbor exchange algorithm for our experimental configurations. The *Generalized Rabenseifner* algorithm performs two allgather operations, once with $N+M$ elements to exchange all vector pairs and once with $\frac{NM}{P}$ elements, the number of elements in the process' chunk of the final matrix. The addition and multiplication costs are from performing one outer product for the process' chunk for each vector pair, resulting in $P \cdot \frac{NM}{P} = NM$ additions and multiplications.

5.4. Further Approaches

In the process of refining the previously mentioned approaches, we implemented several other algorithms as part of the evaluation process. The corresponding results are less relevant, as they do not perform as well.

allgather-async and *bruck-async* were attempts to take existing allgather algorithms for exchanging the vector pairs and perform outer product computations while some data was still in flight using non-blocking send and receive operations.

The *allreduce-butterfly* is a simple reimplementation of *allreduce* using a butterfly structure to send the data around. The matching implementation is similar to the *recursive-doubling* algorithm of Open MPI.

Similar to pipelining inside allgather, we tried to pipeline the *allreduce-butterfly* and the allreduce ring algorithm to perform the reduction step (matrix addition) while data was in flight. We also pipelined the first allgather step in the *Generalized Rabenseifner* to perform the outer product computations in parallel with the data transfers. All these attempts showed at most marginal improvements in runtime since the majority of the runtime can be attributed to communication, so we do not gain much when cutting down on calculation time. Nonetheless, there is still more room for pipelining as we did not pipeline all that was possible (see Section 8).

6. EXPERIMENTAL RESULTS

6.1. Benchmark Setup

All benchmarks were run on the Euler III cluster with a varying number of physical nodes. Each node contains a quad-core Intel Xeon E3-1585Lv5 processor, clocked between 3.0 and 3.7 GHz and 32GB of DDR4 memory, clocked at 2133 MHz. The cluster is arranged as a two-layer fat tree: 45 nodes are connected together via a 10G ethernet switch in a single bay; 27 of such bays are connected via 4x40G ethernet switches to one of Euler's core switches. Even though each node contains 4 cores, for our benchmarks, only 1 process runs per node.

All our code is compiled using gcc version 9.3.0 with the flags `-O3 -std=c++17 -mavx -mavx2 -mfma` and Open MPI 4.0.2. Note that the Rabenseifner algorithm is not present in the decision tree of this particular version of Open MPI, it has only been added in version 4.1 [13, 10, 11].

The experiments are run on multiple input configurations. An input configuration is composed of the sizes N, M of the input vectors a_i and b_i respectively, and the number of processes P. The experiments are restricted to the following ranges of parameter values:

$$N = M \in \{1000 \cdot i \mid 1 \leq i \leq 8,\ i \in \mathbb{N}\}$$
$$P \in \{8, 16, 32, 48\}$$

The range for the number of processes P is mainly motivated by the limited access to the cluster, as students can only request up to 48 nodes at a time. Furthermore, the range of the input sizes N, M is limited by the provided memory, since for larger matrices, the jobs run into memory limits. Moreover, the constraint that our experiments only consider $N = M$ is driven by limiting the number of configurations our experiments explore. Limiting the number of configurations enabled us to do more repeated measurements and, therefore, increased the results' repeatability. Finally, it has to be noted that all algorithms work independently of N and M. However, for the experiments at hand, the algorithms were optimized for the case where $N = M$, i.e., for square matrices. But all the algorithms can easily be generalized for arbitrary values for N, M.

Each configuration is repeated 25 times with the same node assignment, meaning that each MPI process runs the algorithm 25 times. Before each iteration, all processes are synchronized using `MPI_Barrier`. The results of the first iteration is discarded as a warm-up iteration. The remaining iterations are aggregated into a single observation by only looking at the median of these measurements within one node assignment.

Different jobs have potentially different node assignments, possibly leading to varying performance results across otherwise identical input parameters. Therefore, each job is scheduled between 109 and 150 times to get a distribution over observations. To avoid confusion, we will call each of these jobs a "repetition" while each of the 25 repetitions within the same job is an "iteration". Hence, for any given set of input parameters, we have at least 109 observations. Due to long queueing times for jobs with many nodes, the jobs running on 32 or 48 nodes tend to have fewer repetitions, while the ones with 8 and 16 nodes all have 150.

Runtimes are measured as differences in the timestamp returned by `MPI_Wtime`. The runtime for a process is measured from when the `MPI_Barrier` call returns to the point where the process has completed its participation in the algorithm. This point is after the process has the final matrix but may include additional data transfers to other processes that are still part of the algorithm. After the experiment is finished, it results in one runtime value per process; the runtime for the experiment is the maximum of these values.

The evaluation considers statistics based on percentiles as we look at a distribution of observations. However, due to the variability in measurements and non-normally distributed observations, the statistics are estimated using a nonparametric 95% confidence interval, which uses the same method to the *percentile confidence interval* using bootstrap, as used in [16, Chapter 5.7].

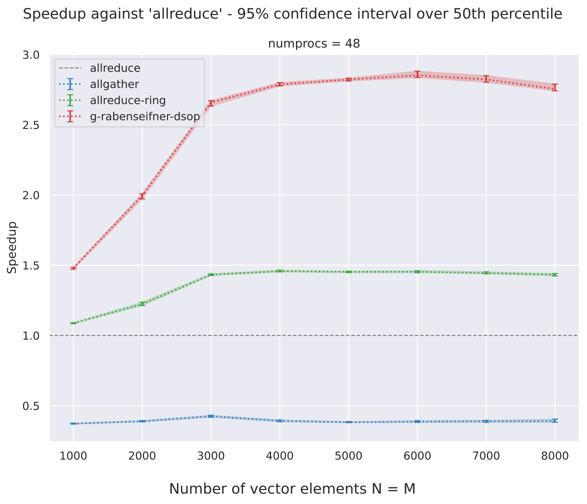

Fig. 2: Median speedup against base allreduce implementation with 95% confidence interval, run with 48 nodes

6.2. Results

Here, we present the key results from our experiments. For the sake of brevity, we only show selected plots. More plots can be found in the appendix in Supplementary Figures as well as in the GitHub repository linked in Section 2.

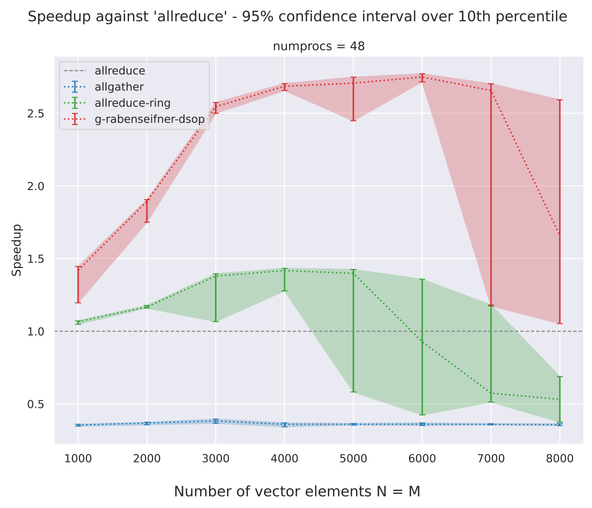

Fig. 3: 10th percentile of speedups against base allreduce implementation with 95% confidence interval, 48 nodes

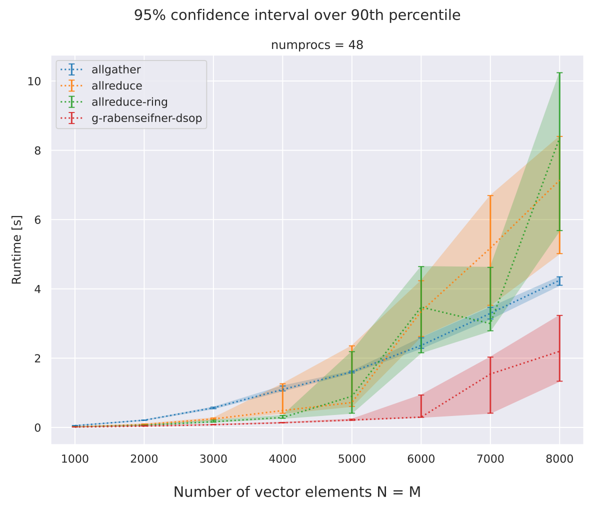

Fig. 4: 90th percentile of runtimes with 95% confidence interval, run with 48 nodes

In the plot legends, *allreduce* and *allgather* refers to the two baseline algorithms described in Section 5.1. The label *allreduce-ring* refers to our allreduce ring algorithm from Section 5.2 and *g-rabenseifner-dsop* refers to the *Generalized Rabenseifner*. Figure 2 shows the median speedup of the benchmarked implementations against the baseline allreduce implementation with a 95% bootstrap confidence interval. In Figure 3, we see the same speedup plot, but for the 10th percentile. For each observation, the speedup is calculated, giving us a distribution of speedups over all repetitions.

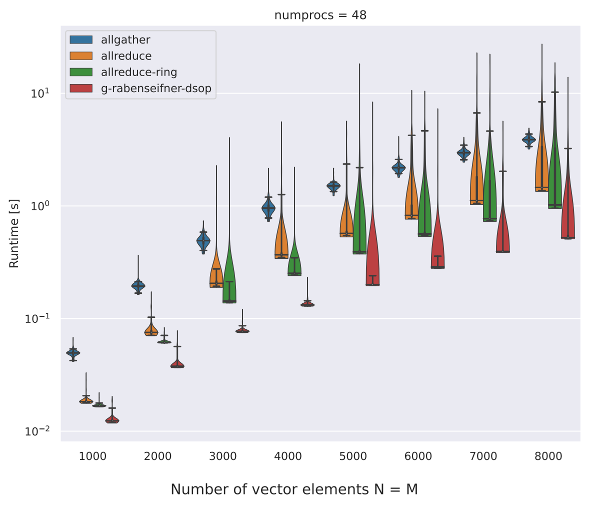

Fig. 5: Runtime distributions per implementation on 48 nodes.

In Figure 4, we see the absolute runtime for the same experiment. This graph shows the 95% bootstrap confidence interval around the 90th percentile of observations, meaning that 90% of the jobs had a faster runtime than indicated by the dotted lines.

Because runtimes can vary between observations, we also present the distributions of runtimes over observations as violins in Figure 5. The violins show all observed values and the boxplots (without the central box) inside the violins show the 95th and 5th percentile as horizontal lines and the median as a slightly longer line. Because the violins for the *Generalized Rabenseifner* algorithm are quite bottom-heavy, the median line often overlaps the 5th percentile line at the bottom of the violin.

7. DISCUSSION

In Figure 2, we see a more than 2.5x median speedup of the *Generalized Rabenseifner* algorithm compared to the allreduce baseline. Even considering the 10th percentile of speedups in Figure 3, meaning 90% of observed speedups were higher, shows a clear advantage except for $N \in \{7000, 8000\}$. Therefore, our algorithm still outperforms existing algorithms in a near worst-case scenario with reasonable certainty.

Additionally, the improvement of our algorithm compared to the existing methods is confirmed by comparing the 90th percentile of speedups to the allreduce baseline. This shows for 16, 32, and 48 processes, the speedup is larger than 3x in most cases (plots not shown). It even reaches values between 6x and 14x for a few cases. Therefore, the proposed algorithm outperforms the existing methods by a large margin in a best-case scenario.

In general, the experiments show a higher variability for larger inputs, especially for $N \in \{7000, 8000\}$. This variability can be attributed to network congestion from other jobs running on the cluster and the effect of node assignment. The relative location of nodes corresponding to a job seems to have a much bigger effect on runtimes when larger amounts of data are sent over the network.

The same behavior can be observed in Figure 5. The spread of runtimes increases, starting at $N = 5000$, with outliers over an order of magnitude slower than the median. Furthermore, the plot suggests that the *Generalized Rabenseifner* sends fewer data over the network than the two allreduce implementations because they start showing a large spread in runtimes at $N = 3000$.

All plots also include measurements for the implementations *allgather* and *allreduce-ring*. The *allgather* implementation is the other of our two baseline implementations. In the median case, the *allgather* is inferior to all other shown algorithms due to the large amount of local computation, which is performed redundantly at every node. However, because the algorithm only requires small data transfers, the overall runtime performance is stable across all experiments and outperforms allreduce on 16 nodes or fewer.

The *allreduce-ring*, which just reimplements the allreduce ring algorithm used in Open MPI, surprisingly, outperforms the allreduce baseline in many cases. However, the speedup advantage is not as clear cut as with the generalized Rabenseifner because the 95% confidence intervals do overlap the allreduce baseline in the worst-case scenarios in Figures 3 and 4. Nevertheless, there is a significant median speedup as seen in Figure 2. We suspect this improvement over the allreduce-baseline may be due to compiler optimizations. Our code is compiled with AVX, AVX2, and FMAs enabled, while Open MPI has none of these enabled. Furthermore, in our *allreduce-ring* implementation, the outer-product step and the allreduce step are implemented in the same function, which could help the compiler better optimize the whole implementation. Because the allreduce runtime is dominated by the send and receive operations, these differences in local computations should not impact the total runtime to this extent. Further investigation is needed to find the root cause of these differences.

8. FUTURE WORK

While the *Generalized Rabenseifner* already provided excellent results, combining the proposed solution with further approaches can lead to additional improvements.

Topology awareness has huge potential to reduce the total runtime of the algorithm. The proposed *Generalized Rabenseifner* can easily be extended to multiple variants using subgroups. In these variants, one can optimize the communication by using topology-aware information. An instantiation of such a variant is motivated by the approach proposed by Gong et al. in [17]. One can adapt their network performance aware operations to the *Generalized Rabenseifner* by instantiating a network hierarchy as proposed in [17] to perform the allreduce operation.

The resulting outer products will also be sparse when working with sparse vectors. Therefore, one can expect substantial performance improvements by optimizing the amount of data sent over the network for sparse input vectors, as most of the runtime arises from communication and not calculation. In [18], the authors are pruning RNNs to get a sparse network which could then be combined with an algorithm that exploits this sparsity during DSOP.

Interleaving computation with communication can lead to further improvements of the overall runtime. However, the relative improvements are expected to be minor because the runtime of local computations bounds the possible reduction in runtime. Initial experiments with pipelining show marginal improvements in runtime in exchange for significant code complexity. However, the effect of pipelining on the overall performance requires further investigation.

As the processes in the *Generalized Rabenseifner* only require a part of the information in the vectors, one can expect improvements by only sending the relevant vector information to the individual processes in the first stage of the algorithm. As an all-scatter algorithm does not exist in MPI, exploring such custom algorithms for further improvements can be beneficial.

9. CONCLUSION

This paper proposes the *Generalized Rabenseifner*, a novel distributed algorithm to compute the DSOP on an arbitrary number of nodes using MPI. The algorithm borrows concepts from the Rabenseifner algorithm but adapts the approach to our problem statement, achieving a good compromise between the number of local computations and the size of exchanged data. The algorithm outperforms a baseline implementation using `MPI_Allreduce` by a factor of 2.5 to 3 in the median case. Furthermore, it has been shown that even in the 90th percentile of runtime observations, the *Generalized Rabenseifner* outperforms the baseline with sufficient confidence. Even though we benchmarked inputs vector pairs of the same size and only up to 48 nodes, we believe that the results hold up for non-square result matrices and for a larger number of nodes. However, to explore the scaling properties of the proposed algorithm, future evaluation on a larger number of nodes is required.

10. REFERENCES

[1] Tal Ben-Nun and Torsten Hoefler, "Demystifying parallel and distributed deep learning: An in-depth concurrency analysis," 2018.

[2] Craig Bakker, Michael J. Henry, and Nathan O. Hodas, "The outer product structure of neural network derivatives," 2018.

[3] Alexander Sergeev and Mike Del Balso, "Horovod: fast and easy distributed deep learning in tensorflow," 2018.

[4] Abhinav Vishnu, Charles Siegel, and Jeffrey Daily, "Distributed tensorflow with mpi," 2017.

[5] He Ma, Fei Mao, and Graham W. Taylor, "Theano-mpi: a theano-based distributed training framework," 2016.

[6] Kailai Xu, Weiqiang Zhu, and Eric Darve, "Distributed machine learning for computational engineering using mpi," 2020.

[7] Guanhua Wang, Shivaram Venkataraman, Amar Phanishayee, Jorgen Thelin, Nikhil Devanur, and Ion Stoica, "Blink: Fast and generic collectives for distributed ml," 2019.

[8] Truong Thao Nguyen, Mohamed Wahib, and Ryousei Takano, "Topology-aware sparse allreduce for large-scale deep learning," in *2019 IEEE 38th International Performance Computing and Communications Conference (IPCCC)*, 2019, pp. 1–8.

[9] J. Bruck, Ching-Tien Ho, S. Kipnis, E. Upfal, and D. Weathersby, "Efficient algorithms for all-to-all communications in multiport message-passing systems," *IEEE Transactions on Parallel and Distributed Systems*, vol. 8, no. 11, pp. 1143–1156, 1997.

[10] Open MPI Team, "Openmpi decision tree for allreduce and allgather, version 4.0.2," https://github.com/open-mpi/ompi/blob/v4.0.2/ompi/mca/coll/tuned/coll_tuned_decision_fixed.c, 2019.

[11] Open MPI Team, "Openmpi decision tree for allreduce and allgather, version 4.1.2," https://github.com/open-mpi/ompi/blob/v4.1.2/ompi/mca/coll/tuned/coll_tuned_decision_fixed.c, 2020.

[12] Jing Chen, Linbo Zhang, Yunquan Zhang, and Wei Yuan, "Performance evaluation of allgather algorithms on terascale linux cluster with fast ethernet," in *Eighth International Conference on High-Performance Computing in Asia-Pacific Region (HPCASIA'05)*, 2005, pp. 6 pp.–442.

[13] Open MPI Team, "Openmpi github," https://github.com/open-mpi/ompi/tree/v4.1.2, 2021.

[14] Rajeev Thakur, Rolf Rabenseifner, and William Gropp, "Optimization of collective communication operations in mpich," *Int. J. High Perform. Comput. Appl.*, vol. 19, no. 1, pp. 49–66, feb 2005.

[15] Rajeev Thakur and William D. Gropp, "Improving the performance of collective operations in mpich," in *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, Jack Dongarra, Domenico Laforenza, and Salvatore Orlando, Eds., Berlin, Heidelberg, 2003, pp. 257–267, Springer Berlin Heidelberg.

[16] Bill Venables and B Ripley, *Modern Applied Statistics With S*, 01 2002.

[17] Yifan Gong, Bingsheng He, and Jianlong Zhong, "Network performance aware mpi collective communication operations in the cloud," *IEEE Transactions on Parallel and Distributed Systems*, vol. 26, pp. 3079–3089, 2015.

[18] Sharan Narang, Erich Elsen, Gregory Diamos, and Shubho Sengupta, "Exploring sparsity in recurrent neural networks," 2017.

A. SUPPLEMENTARY FIGURES

A.1. Speedup

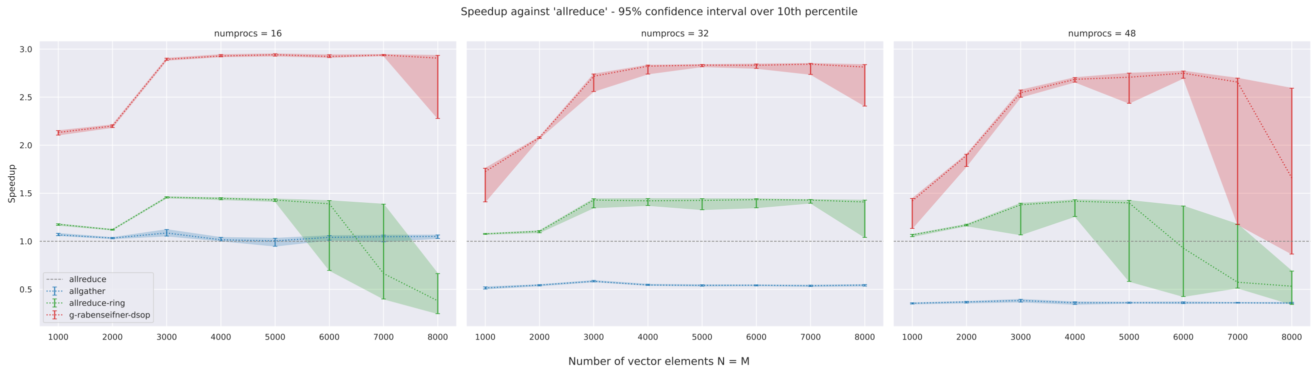

Fig. 6: 10th percentile of speedups against base allreduce implementation with 95% confidence interval, run with 16, 32, and 48 nodes

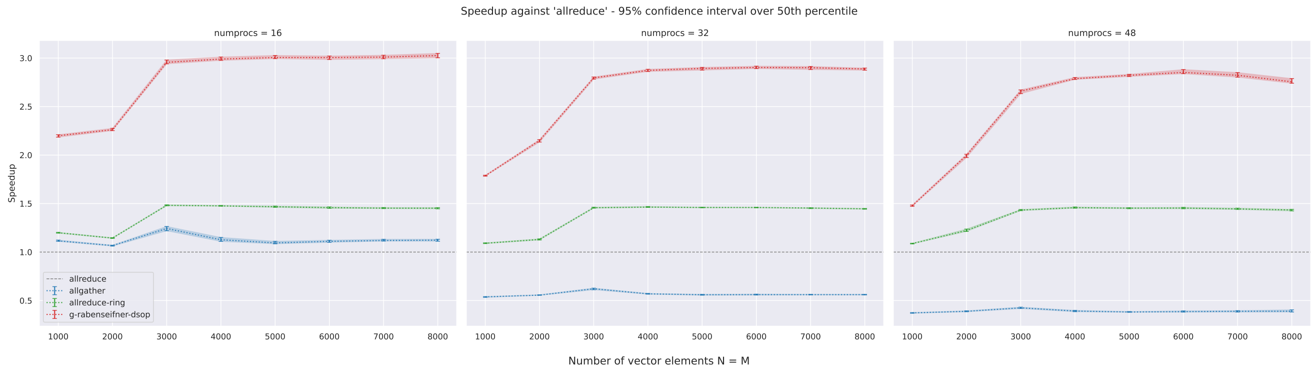

Fig. 7: Median speedup against base allreduce implementation with 95% confidence interval, run with 16, 32, and 48 nodes

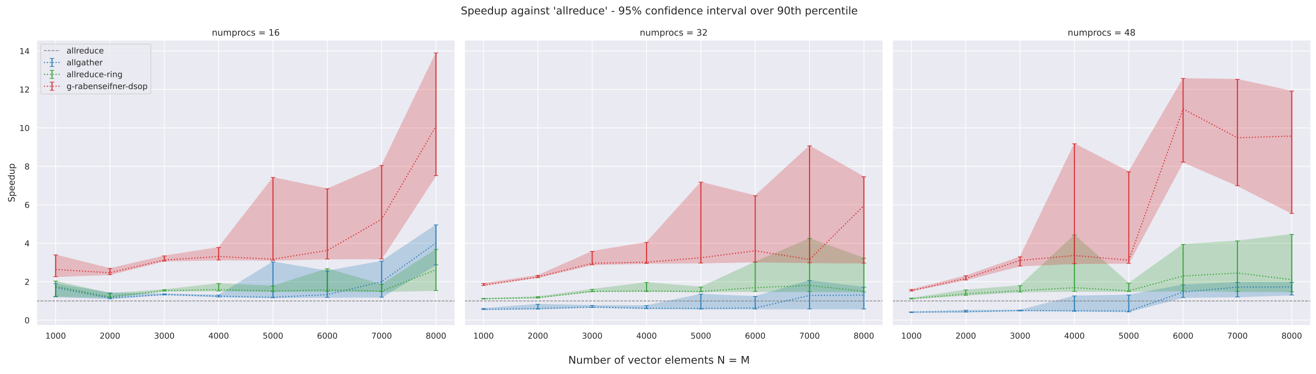

Fig. 8: 90th percentile of speedups against base allreduce implementation with 95% confidence interval, run with 16, 32, and 48 nodes

A.2. Runtimes

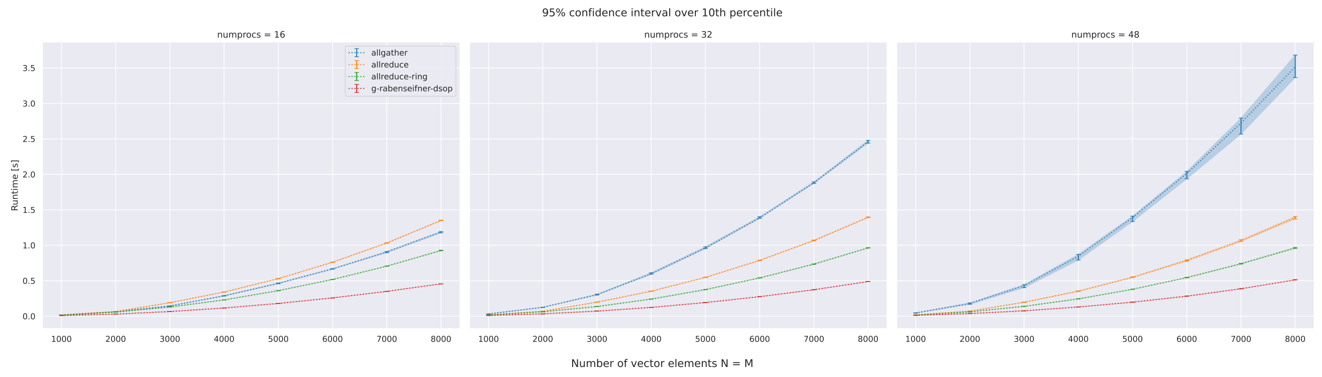

Fig. 9: 10th percentile of runtimes with 95% confidence interval, run with 16, 32, and 48 nodes

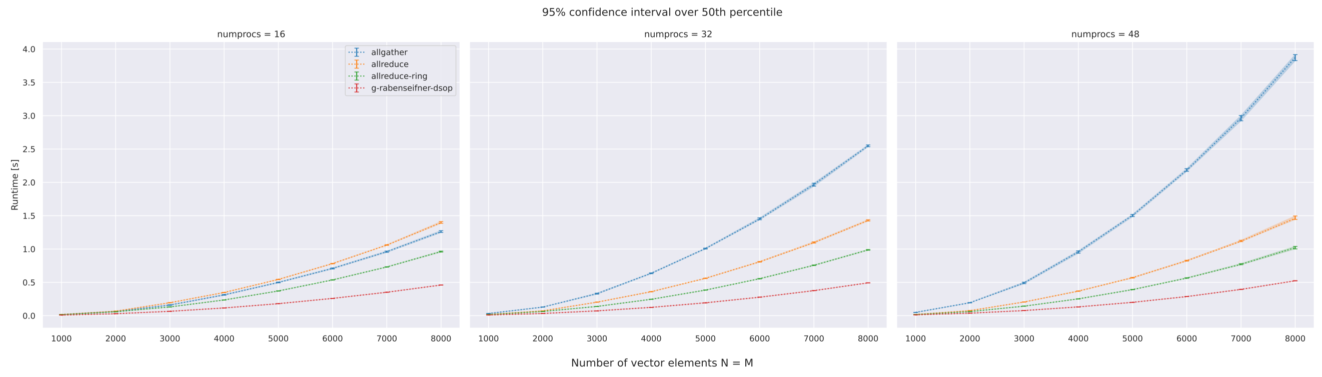

Fig. 10: Median runtimes with 95% confidence interval, run with 16, 32, and 48 nodes

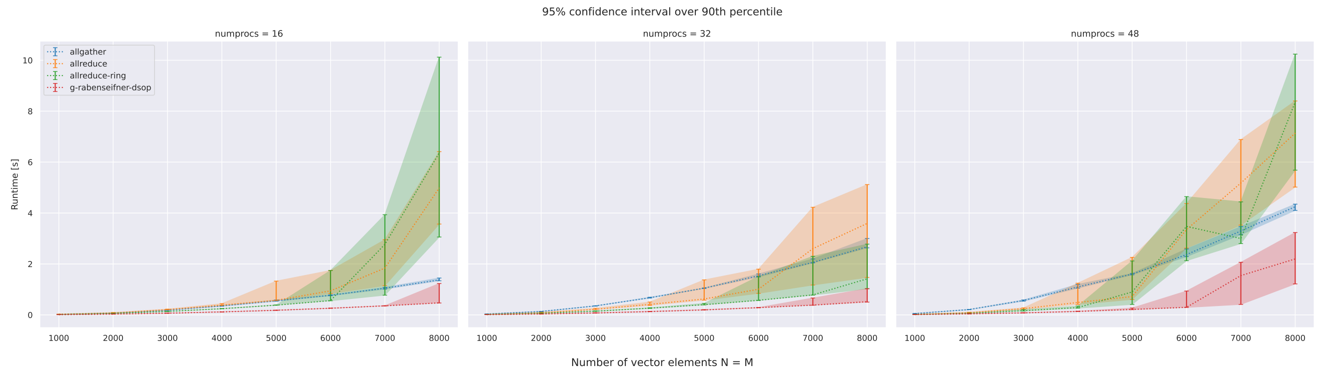

Fig. 11: 90th percentile of runtimes with 95% confidence interval, run with 16, 32, and 48 nodes

A.3. Runtime Distributions

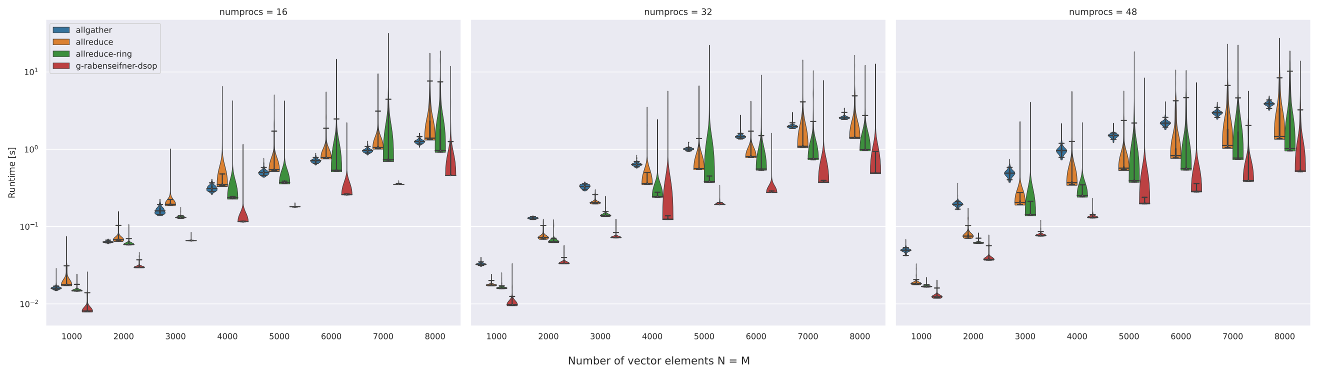

Fig. 12: Runtime distributions per implementation on 16, 32, and 48 nodes.