SAND2021-10681C

Low-Communication Asynchronous Distributed Generalized Canonical Polyadic Tensor Decomposition

1st Cannada Lewis
Sandia National Laboratories
Livermore, California
canlewi@sandia.gov

2nd Eric Phipps
Sandia National Laboratories
Albuquerque, New Mexico
etphipp@sandia.gov

***Abstract*—In this work, we show that reduced communication algorithms for distributed stochastic gradient descent improve the time per epoch and strong scaling for the Generalized Canonical Polyadic (GCP) tensor decomposition, but with a cost, achieving convergence becomes more difficult. The implementation, based on MPI, shows that while one-sided algorithms offer a path to asynchronous execution, the performance benefits of optimized allreduce are difficult to best.**

***Index Terms*—Tensor Decomposition, Canonical Polyadic, Stochastic Gradient Descent, Asynchrony, Federated Learning**

I. Introduction

We consider a tensor to be a multidimensional array of data (dense, sparse, or scarce [1]) that is a natural representation of many data sets [2]. Tensor decompositions are a group of methods that seek to represent tensors via an ansatz that provides one or more of the following: reduced storage, interpretability of the data, dimensionality reduction, or a separable approximation useful for further transformations. The Canonical Polyadic (CPD) and Tucker decompositions [2] are two decompositions that have been heavily investigated and provide similar utility to the singular value decomposition for matrices. One advantage of the CPD over the Tucker decomposition is that the CPD decomposition does not suffer from the curse of dimensionality. In this work, we focus on solving the Generalized form of the Canonical Polyadic tensor decomposition (GCP) [1]. GCP, cannot be solved with the traditional CPD method alternating least squares, instead we use stochastic gradient decent (SGD).

In the last decade, the machine learning community has heavily invested in the scaling of distributed stochastic gradient decent (dSGD) [3], we test two approaches for reducing the communication in dSGD: Local SGD (LSGD) [4] and Elastic Averaging (EA) [5] and compare them to a synchronous distributed implementation of ADAM [6]. Other work investigates alternatives to SGD [7] and efficient distribution and performance optimization [8], [9], but to our knowledge reduced communication strategies for dSGD have not been explored for GCP tensor decompositions.

Sandia National Laboratories is a multimission laboratory managed and operated by National Technology and Engineering Solutions of Sandia, LLC, a wholly owned subsidiary of Honeywell International Inc., for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA0003525. This paper describes objective technical results and analysis. Any subjective views or opinions that might be expressed in the paper do not necessarily represent the views of the U.S. Department of Energy or the United States Government. Sand Number: TODONEEDTOFINISHTHIS

II. Algorithms

Algorithmic details for the parts of our algorithm that are independent of multi-process distribution are covered in [10]. A major (maybe dominate) motivation of our distribution strategy was to reuse as much of this single process implementation as possible, this is one place where we differ from [11].

A. Anatomy of Our SGD algorithm

We follow the standard strategy for SGD popular in the ML community, with a few caveats. We set large units of work, called epochs, to use the number of non-zero (nnz) tensor elements and an equal number of zeros, via semistratified sampling [12]. To do this, we compute gradients using a fixed number of samples, called a mini-batches, such that the number of mini-batches times the size of the mini-batches is approximately the number of non-zeros in the tensor. For implementation purposes, we break from tradition in two main ways: 1) we do sampling with replacement, 2) for the non-zero sampling, each distributed worker, called a rank, will sample the minimum of either the batch size divided by the number of ranks or the number of local nnz. When the distribution of the tensor nnz is not well balanced this could introduce bias into our sampling, but it makes the implementation of the distributed algorithm simpler.

For timing purposes our SGD algorithm can be broken into three phases:

1) *Gradient*: the calculation where the local contributions to the gradient are computed.
2) *Communication*: Measures any communication steps used in the algorithm
3) *Update*: where the local factor matrices are updated using the factor gradients, surprisingly to the authors (but in hindsight obvious) this step can be expensive

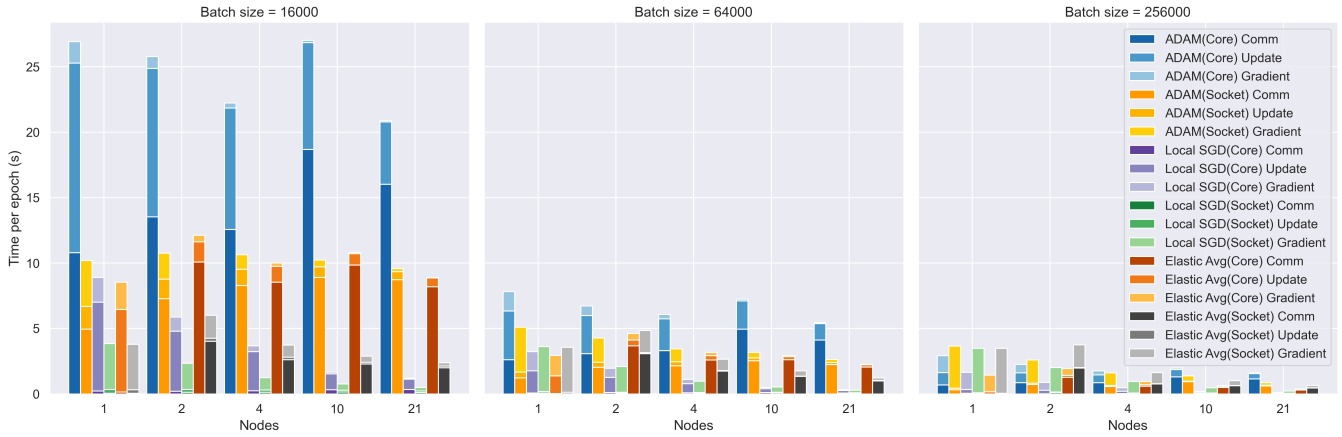

Fig. 1. Stacked average times for each phase of dSGD (the difference between the sum of these three phases and the total epoch time was negligible). The decomposition rank was 16 and the communication period was fixed at 128 mini-batches for the LSGD and EA methods.

depending on the amount of factor replication and thread parallelism choices in the algorithm.

B. Tensor and Factor Distribution

To explain the distribution strategy it will be helpful to know what our goals are. We are motivated by the following, not universally true, assumptions:

1) The gradient calculation is the most expensive step of each mini-batch
2) The implementation of `MPI_Allreduce` will be efficient, we assume that most HPC style clusters and MPI implementations will provide a high performance algorithm.

with these goals in mind we initially distribute the tensor as follows. We will replicate any row of a factor matrix that might be used in local gradient computations and we will distribute the tensor in blocks that correspond to contiguous regions of each mode. To do this, we create a MPI Cartesian grid with the same dimensionality as the tensor that minimizes factor storage. We then give the user the option (the approach used in this work) of testing small perturbations to this grid with actual `MPI_Allreduce` calls to factor matrices with random data. Allowing users to trade faster communication for increased storage, if storage reduction is a priority this step can be skipped.

This distribution strategy will partially replicate the factor matrices, but in a way that the storage required for factor matrix replication grows more slowly than the total memory for the system, under the assumption that adding another rank always adds additional memory. Because factors are always dense minimizing their storage does not take into account the distribution of nonzero data in the tensor. For some tensors this can become an issue when the non-zeros are clustered, either naturally or by construction.

Finally, in the EA implementation we store a sharded center variable in a `MPI_Window` distributed over the subgrid that corresponds to those particular rows of the factor matrices. The window is distributed such that each rank in the subgrid holds an equal number of rows of the center. The sharded center represents the consensus view of the factors, that is used to gauge accuracy and convergence. The authors suggest thinking of this `MPI_Window` as convenient way to implement a data structure that functions as a parameter server in MPI.

C. dSGD Algorithms

We implemented three strategies for dSGD:

1) Synchronous allreduce with ADAM [6],
2) LSGD, in which the factors are averaged after a predetermined number of mini-batches (τ) using `MPI_Allreduce`,
3) EA, with independent ranks that asynchronously read and write a sharded center after a predetermined number of mini-batches (τ), using MPI one-sided communication.

In the algorithms 1, 2, and 3 the method $sampler.fusedGradient(F)$, is an implementation detail of Genten, that consumes the created element of the tensor gradient immediately. This allows $fusedGradient$ to avoid the two step process of creating and storing a gradient tensor and calling a separate Matricized tensor times Khatri-Rao product routine.

Algorithm 1 ADAM SGD mini-batch iteration with AllReduce

```
procedure ADAMSGDITERATION(F, sampler)
    G = sampler.fusedGradient(F)
    Allreduce(G)    ▷ Calls MPI_Allreduce on each gradient block
    F.update(G)
end procedure
```

In algorithm 1, we see that all that is required to distribute ADAM SGD is to unconditionally allreduce the gradient before the factors are updated. This ensures that the factors on different ranks see the same gradients, but also requires communication for every single mini-batch.

Algorithm 2 Local SGD mini-batch iteration

procedure LOCALSGDITERATION($F, sampler, e, \tau$)
  if $e\%\tau == 0$ **then** ▷ If τ divides the epoch iteration, e, synchronize
    $Allreduce(F)$▷ Calls `MPI_Allreduce` on each replicated factor block
    $F = F/numMPIRanks$
  end if
  $G = sampler.fusedGradient(F)$
  $F.update(G)$
end procedure

In algorithm 2, instead of ensuring that all ranks receive the same gradient we simply average the factor matrices with a period of τ. So every τ iterations we ensure that all the ranks have the same factor matrices. While communicating less frequently than algorithm 1, this is still synchronous, since all ranks must all reach the allreduce method before any rank can proceed.

Algorithm 3 Elastic Averaging SGD mini-batch iteration

procedure LOCALSGDITERATION($F, sampler, e, \tau, C$)
  if $e\%\tau == 0$ **then** ▷ If τ divides the epoch iteration, e, communicate
    $C = readCenter()$ ▷ Calls `MPI_Get_accumulate`
    $D = 0.9(F - C)$
    $F = F - D$
    $writeCenterUpdate(D)$ ▷ Calls `MPI_Accumulate`
  end if
  $G = sampler.fusedGradient(F)$
  $F.update(G)$
end procedure

Finally, in algorithm 3, we see that Elastic Averaging requires doing an asynchronous read of a center variable (initially the center and all the factors are initialized to the same values), then a local update based on the difference between the local factors and the center, which is followed by an asynchronous accumulation of that difference back to the center. Importantly, none of these operations depend on explicit synchronization with any other ranks. The read and write operation are implemented with `MPI_Get_accumulate` and `MPI_Accumulate` allowing safe, but arbitrary interleaving of reads and writes.

III. IMPLEMENTATION

We extended the Genten [10] tensor decomposition package to make use of distributed computing (see similar work at Sandia [11]) using MPI. Genten employs the Kokkos [13] library to provide on node parallelism and the flexibility of MPI+X.

Hardware: All tests were run on the Sandia Blake cluster with two Xeon Platinum Skylake 8160 CPUs per node (48 cores/node), 28 nodes, and a 100Gbs OmniPath network.

Software: The code for this project can be found at `https://gitlab.com/tensors/genten/` on the mpi-sgd branch. The external dependencies used were Boost (1.75), Open MPI (4.0.5), Kokkos (commit 1fb0c28, `OpenMP` backend), all compiled with Intel 2021.2.0 compilers.

IV. RESULTS

The sparse tensor used was NELL-2 [14], [15] with dimensions [12092, 9184, 28818] and 76879419 non-zeros (nnz). The size of each epoch was chosen to sample all non-zeros (76879419) and an equal number of zeros with replacement using semistratified sampling [12]. We used a Poisson loss function. Each epoch was divided into equal sized mini-batches with three phases: 1) gradient computation, 2) factor update, and 3) communication. Reported timings are an average of 50 epochs for the ADAM results and 8 epochs for the others. EA and LSGD are sensitive to hyper parameter selection, so convergence and time to solution comparisons will be reported in a future work. The all results, except rank scaling, used a decomposition rank of 16. Finally, we ran two different configurations: one MPI rank per core (Core) and one MPI rank per socket (Socket), the latter used 24 OpenMP threads on each rank.

A. Strong Scaling and Batch Size Effects

In Figure 1, we see communication dominate ADAM times at smaller mini-batch sizes and large node counts, this confirms that our method is very effective at meeting its goal of strong scaling the gradient computation. As expected, communication as a percentage of the epoch time is larger when the mini-batch size is small due to more communication events. The communication avoiding methods—especially LSGD, which exploits optimized `MPI_Allreduce` routines—show the effects of reduced communication overhead when the synchronization period is 128 mini-batches. We also see that the batch size has a large effect on the epoch times for all methods, the main reason for this is that larger batches reduce both the number of update and communication steps per epoch, while the work for the gradient calculations stays approximately constant (once the mini-batch is large enough to effectively hide the work required to set up and tear down a mini-batch). The effect is most pronounced when analyzing the ADAM(Core) method on a single node, with a batch size of 16000 communication and update steps dominate, while with a batch size of 256000 they are much more closely balanced.

Here we also can see that the time spent in the update step can become significant, especially when distributing by core. The reason for this is that each rank must apply gradient updates to it's local copy of the factors. When the amount of replication is high this leads to significant redundant work, in the 21 node case there are 1008 MPI ranks for example, each which must do some amount of redundant work.

While for many cases the jobs distributed by socket are slightly to significantly faster than the jobs distributed by cores, there are times when that is not the case. So even though

distributing by socket may be a good default recommendation, there are configurations where by core will be better.

Finally, we can see that for this tensor on this system, it is very hard for the MPI one-sided communication used in Elastic Averaging to match the performance of LSGD, when the frequency of communication is the same, we believe this is partly due to the fact that `MPI_Allreduce` is a highly optimized routine and NELL-2's non-zeros are distributed in a way that leads to acceptable load balance in the configurations we tested. That is not to say that LSGD is a more efficient method than EA with respect to time to solution or convergence accuracy, we simply didn't compute those outcomes, but intend to do so in future work.

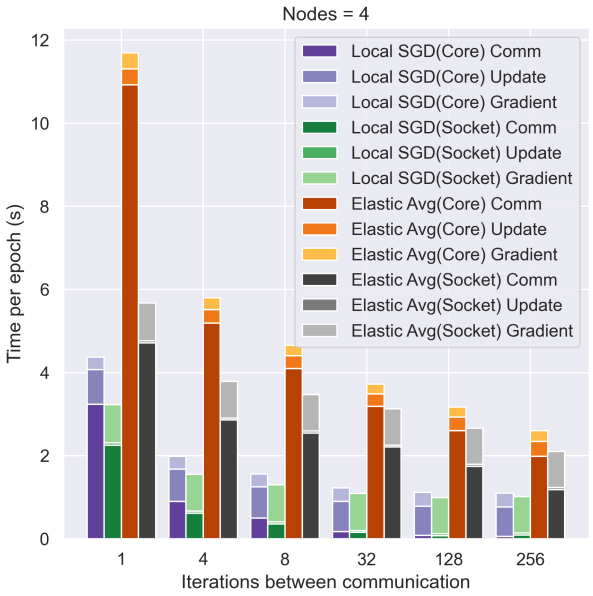

Fig. 2. Test, on four nodes, of the impact of communication period on dSGD phases. The decomposition rank was 16 and the mini-batch size was 64000

B. Frequency of Communication

Figure 2, shows the average epoch time decreases as the frequency of communication decreases. This is expected and without convergence results it is hard to make any deep insights or recommendations. This plot does make it clear that the EA method is bottlenecked by communication for this configuration with NELL-2, suggesting that the benefits of asynchrony may be hard to realize on HPC systems, with fast reliable networks.

C. Effect of Decomposition Rank

As expected we see that time per epoch increases with increasing rank. The 128 rank jobs also highlight a trend that the jobs distributed by cores have an advantage over the jobs distributed by socket, with regards to computing the gradients. This is due to the fact that the single core jobs do not need to do any synchronization for writing the gradient elements, while the socket based jobs need to use atomics (or other methods) to avoid race conditions. There don't seem to be

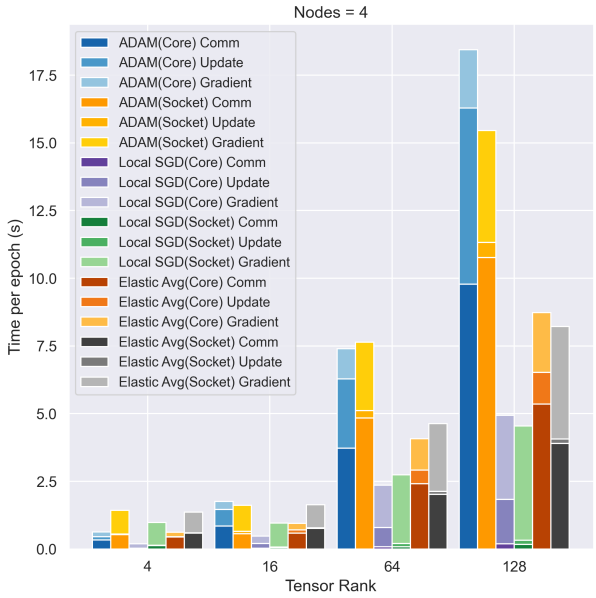

Fig. 3. Test, on four nodes, of the impact of decomposition rank with respect to epoch times. The communication period was every 128 mini-batches and the mini-batch size was 256000

any obvious take aways except that as expected all phases of computation increase in cost as the rank of the decomposition increases and the relative performance of distributing by cores or sockets may depend on the rank of the decomposition.

V. Conclusions

A. Insights

We showed that communication avoiding dSGD algorithms lead to faster epoch times and improved strong scaling relative to synchronous ADAM. The best scaling is obtained when the gradient is the dominant cost of each epoch, favoring large mini-batch sizes and infrequent communication. The trade-offs between different configurations (Core versus Socket) were mixed, warranting more investigation. These results are promising, but challenges remain: namely to prove that low communication algorithms can match synchronous ones in time to solution and accuracy.

B. Questions Left Unanswered and Future Avenues of Investigation

The big questions that we didn't address in this work is which method achieves a certain quality of solution in the fastest time. LSGD and EA both were very sensitive to the annealing schedule (a detail, not discussed in this work) and the learning rate, making it difficult to compare convergence and time to solution. Future work will be to find strategies to make their convergence more robust so a true comparison to synchronous ADAM can be made.

While the choice to use a distribution that minimizes factor replication/communication in this work may not be obvious when using a HPC cluster with ample memory and fast networks, this decision was made with GPUs in mind. One

of our next goals is to port this work to GPU clusters with few nodes and multiple GPUs per node. We chose to minimize factor matrix replication with the idea that we would want to fit our factors and tensors on 10s of GPUs as oppose to 1000s of cores. Allreduce based methods will also be valuable when porting to GPUs because of the existence of optimized routines and the fact that asynchronous MPI operations are not well supported at this point in time.

Finally, we need to investigate more tensors and different loss functions. Some of the trends seen in this work may not hold for tensors of different dimensions, sparsity, sparsity patterns, or size.

REFERENCES

[1] D. Hong, T. G. Kolda, and J. A. Duersch, "Generalized canonical polyadic tensor decomposition," *SIAM Review*, vol. 62, no. 1, pp. 133–163, 2020.

[2] T. G. Kolda and B. W. Bader, "Tensor decompositions and applications," *SIAM review*, vol. 51, no. 3, pp. 455–500, 2009.

[3] J. Dean, G. Corrado, R. Monga, K. Chen, M. Devin, M. Mao, M. Ranzato, A. Senior, P. Tucker, K. Yang *et al.*, "Large scale distributed deep networks," *Advances in neural information processing systems*, vol. 25, pp. 1223–1231, 2012.

[4] S. U. Stich, "Local sgd converges fast and communicates little," *arXiv preprint arXiv:1805.09767*, 2018.

[5] S. Zhang, A. Choromanska, and Y. LeCun, "Deep learning with elastic averaging sgd," in *Proceedings of the 28th International Conference on Neural Information Processing Systems - Volume 1*, ser. NIPS'15. Cambridge, MA, USA: MIT Press, 2015, p. 685–693.

[6] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," *arXiv preprint arXiv:1412.6980*, 2014.

[7] T. M. Ranadive and M. M. Baskaran, "Large-scale sparse tensor decomposition using a damped gauss-newton method," in *2020 IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE, 2020, pp. 1–8.

[8] S. Smith, N. Ravindran, N. D. Sidiropoulos, and G. Karypis, "Splatt: Efficient and parallel sparse tensor-matrix multiplication," in *2015 IEEE International Parallel and Distributed Processing Symposium*. IEEE, 2015, pp. 61–70.

[9] M. Baskaran, T. Henretty, and J. Ezick, "Fast and scalable distributed tensor decompositions," in *2019 IEEE High Performance Extreme Computing Conference (HPEC)*, 2019, pp. 1–7.

[10] E. T. Phipps and T. G. Kolda, "Software for sparse tensor decomposition on emerging computing architectures," *SIAM Journal on Scientific Computing*, vol. 41, no. 3, pp. C269–C290, 2019.

[11] K. D. Devine and G. Ballard, "Gentenmpi: Distributed memory sparse tensor decomposition." Sandia National Lab.(SNL-NM), Albuquerque, NM (United States), Tech. Rep., 2020.

[12] T. G. Kolda and D. Hong, "Stochastic gradients for large-scale tensor decomposition," *SIAM Journal on Mathematics of Data Science*, vol. 2, no. 4, pp. 1066–1095, 2020.

[13] H. C. Edwards, C. R. Trott, and D. Sunderland, "Kokkos: Enabling manycore performance portability through polymorphic memory access patterns," *Journal of parallel and distributed computing*, vol. 74, no. 12, pp. 3202–3216, 2014.

[14] S. Smith, J. W. Choi, J. Li, R. Vuduc, J. Park, X. Liu, and G. Karypis. (2017) FROSTT: The formidable repository of open sparse tensors and tools. [Online]. Available: http://frostt.io/

[15] A. Carlson, J. Betteridge, B. Kisiel, B. Settles, E. R. Hruschka Jr., and T. M. Mitchell, "Toward an architecture for never-ending language learning." in *AAAI*, vol. 5, 2010, p. 3.