WITS UNIVERSITY

School of Computer Science and Applied Mathematics
UNIVERSITY OF THE WITWATERSRAND, JOHANNESBURG

Distributed Deep Learning

Chris Fourie,358183, MSc

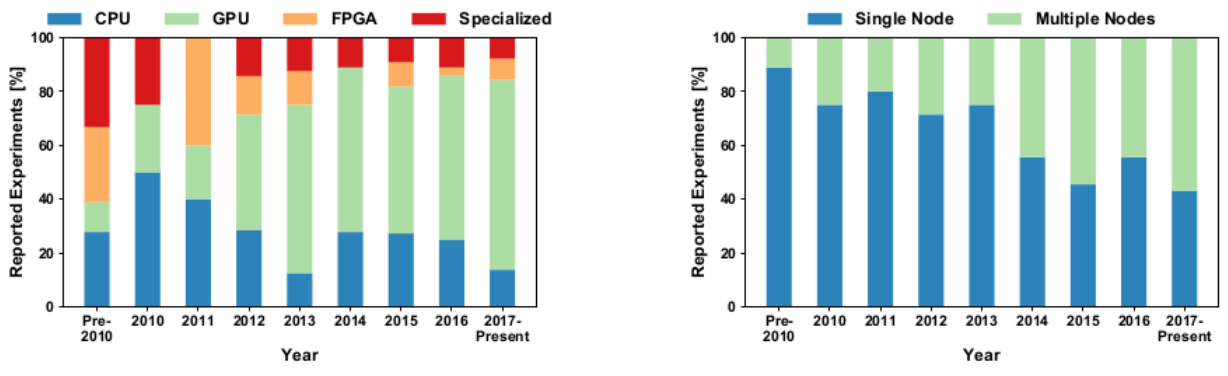
13 April 2019

Abstract

Deep learning techniques are an increasingly popular tool for solving a broad set of problems and showing signs that this will continue into the foreseeable future [1]. At the core of deep learning is the neural network architecture which to meet the requirements of general industrial performance standards, needs to be implemented at scale. Here we investigate recent popular methods of scaling fundamental neural networks components, with implementation and comparison of parameter sharing methods and potential application implications to healthcare research in Africa.

History and relevance

Popularity has been growing since the success of the deep learning neural network AlexNet in 2012 at ImageNet Large Scale Visual Recognition Competition (ILSVRC). Many traditional techniques for regression, classification and clustering have since been replaced by deep learning [9]. Parallel to this there has been a rise in the use of GPUs owing to the streaming multiprocessor (SMP) architecture that provides an order of magnitude more processes than traditional CPUs on an individual piece of hardware or node. Additional available processes provide opportunity for scaling the training of deep neural networks. There has also been an increase in how many nodes are used to train the neural networks. The figures 1 and 2 reflect this progression from 2010.



(a) Hardware Architectures

(b) Training with Single vs. Multiple Nodes

Figure 1: Parallel Architectures in Deep Learning [1]

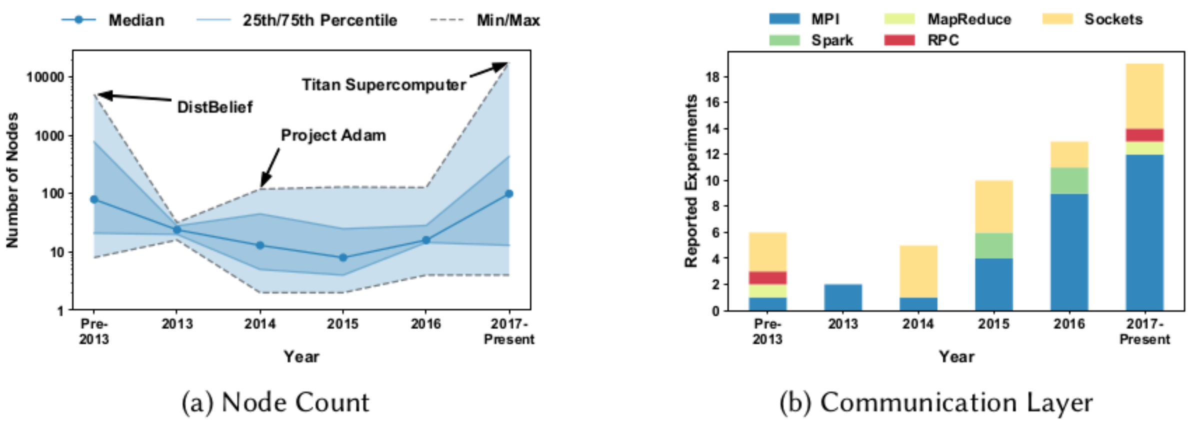


Figure 2: Characteristics of Deep Learning Clusters [1]

The management of communication between nodes is an important aspect to consider. Here MPI (Message Passing Interface) has been used in this implementation however there are a number of other frameworks and standards, however MPI is the most popular by a significant margin. The utility of using popular tools is seen in the documentation and online support from the community using it, with a large set of custom solutions to frequently asked questions as well as extremely niche questions. These can normally be found in scattered between forums, wikis and questions and answer websites.

Parallel Strategies

Parallel Algorithms [1]

DAG and average parallelism
Every computation on a computer can be modeled as a directed acyclic graph (DAG). The vertices of the DAG are the computations and the edges are the data dependencies (or data flow). The computational parallelism can then be characterized by two main parameters, the graphs work W, corresponding to the total number of vertices and the graph's depth D as the number of vertices along the longest path in the DAG. This allows one to characterize the computational complexity on a parallel system.

If we assume one operation per time unit, then the time needed to process the graph on a single processor (single vertex) is $\textit{number of processors} = T = 1 = \mathbf{W}$ and the time needed to process the graph of an infinite number of processes is $\textit{number of processors} = T = \infty = D$.

Often a *good* measure of how many processes to execute a graph with is the average parallelism expressed as (W/D). The execution time of such a DAG on p processors is bounded by $\min\{\mathbf{W}/p, \mathbf{D}\} \leq T_p \leq O(\mathbf{W}/p + \mathbf{D})$ [2] [3]

Reductions
Many of the basic operation associated with neural networks can already be modeled optimally in parallel using tensors, which is one of the reasons for their popularity. However summation or reduction operations introduce data dependencies (edges in the DAG). This makes reduction operations a target for novel parallel algorithms.

In a reduction, a series of binary operator $\oplus$ are used to combine n values into some $< n$ value and in most practical cases into a single value. $y = x_1 \oplus x_2 \oplus x_3 \cdots \oplus x_{n-1} \oplus x_n$. With the assumption that the operation $\oplus$ is associative, then the DAG can be changed from a linear-depth structure to a base 2 logarithmic-depth tree structure as shown in figure 3.

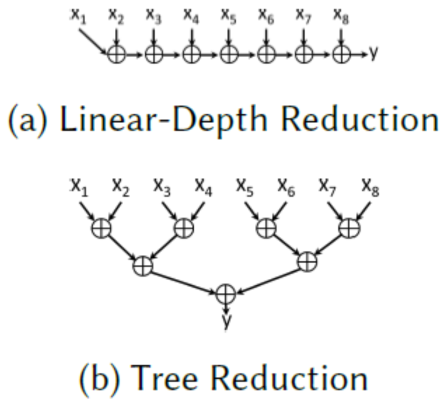

x_1 x_2 x_3 x_4 x_5 x_6 x_7 x_8
y
(a) Linear-Depth Reduction
x_1 x_2 x_3 x_4 x_5 x_6 x_7 x_8
y
(b) Tree Reduction

Figure 3: Reduction scheme [1]

Using reductions in such a way lets one model work and depth as $\mathbf{W} = n-1$ and $\mathbf{D} = \lceil \log_2 n \rceil$ [1].

AllReduce
Deep learning often needs to reduce large tensors of m independent parameters and then return or broadcast the resultant individual value of this reduction to all of the processes. Generally one tries to assign one processes per independent parameter. MPI provides the *AllReduce* specification / function to achieve this, as a combination of a reduce and broadcast operation. As GPU libraries such as NVidias CUDA generally provided much better performance on individual hardware nodes (machines), MPI as the communication layer between nodes. The communication bandwidth between nodes is relatively low when compare to local communication bandwidth within a node. This makes the *AllReduce* operation one of the most critical operations for distributed deep learning. Using a *LogP* model [5],similar to an $\alpha - \beta$ model, where $L = \alpha$ the point-to-point latency in the network, $G = \beta$ the cost per byte and $P \leq p$ the number of networked nodes. Combining this with the DAG model, the lower bound for the reduction time is $T_r \geq L\log_2(P)$. As each element has to be sent at least once, the second lower bound is $T_r \geq \gamma mG$ where γ represents the size of the single data value sent and m is the number of values sent. We can therefore consider the size of a message being sent between nodes to be γm and the cost per byte of that message to be G.

There are a number of algorithms for the parallel *AllReduce* that show differing utility with various *environments*, *message sizes* and *number of processes*[4] [6]. The simplest key algorithm is to combine two trees, the first tree to perform the reduction, the second tree to perform the broadcast, see figure 4. It's complexity is $T_{\text{tree}} = 2\log_2(P)(L + \gamma mG)$. This can be optimized with a butterfly pattern using a recursive halving reduce-scatter followed by a recursive doubling all-gather[7], improving the time complexity to $T_{bfly} = \log_2(P)(L + \gamma mG)$, see figure 4. The butterfly algorithm is near-optimal for small messages, i.e small γm.[1]. It is not clear exactly which *AllReduce* algorithm is used by MPI as there are multiple versions of implementations that vary, OpenMPI version 4.0.1 was used for this implementation. No specific *AllReduce* algorithm is evident in the OpenMPI documentation and no specific algorithm could be found in the OpenMPI source code. It is also possible that some implementations use adaptive algorithms according to their network environment and the message properties [8].

AllReduce Ring[7]
The problem with the butterfly communication pattern is that it can cause network contention in many contemporary clusters, such as the previously mentioned SMP or multi-core clusters. That is in a first come-first served network environment, when two nodes try to transmit at the same time, also known as a collision. In contrast the ring-based algorithm only requires a tree topology to be bandwidth optimal and can achieve contention-free communication in almost all contemporary clusters, including SMP/multicore clusters and Ethernet switched clusters with multiple switches. The ring based structure also requires less working memory and can be applied to clusters with non-power-of-two numbers of nodes.

A limitation of the ring algorithm is that it is only optimal in terms of bandwidth but not latency. The number of

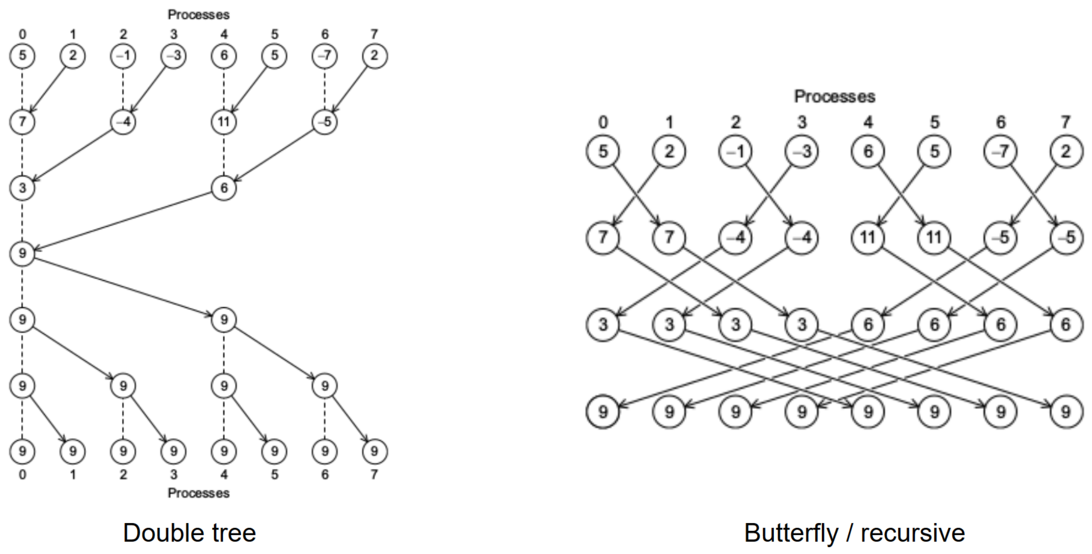


Figure 4: *AllReduce* parallel algorithms, Credit: University Witwatersrand

communication rounds is proportional to the number of processes. Additionally in ring-based algorithms the reduction results are computed with different "bracketing" which may cause problems in the presence of rounding errors. See figure 5 for an example of how the broadcast step of the all reduce is implicit in the ring structure.

Cost of computing

Using the Work-Depth(W-D) model one can formulate the costs of computing the forward and backpropagation of different layers types. Figure 6 shows a summary of these costs when considering images as inputs, where $N =$ the number of samples in a minibatch, $C =$ the number of channels (usually RGB), H and W are the height and width of the images.

This shows that the work W performed in each layer asymptotically dominates the maximal operation dependency path D this is at most logarithmic in parameters. This reaffirms that parallelism is a major consideration in the feasibility of evaluating and training when designing deep neural networks.

Partitioning Strategies

Data Parallelism

As most of the operators are independent with respect to N the size of the minibatch samples, the samples can be distributed among multiple nodes and cores. Initially called pattern parallelism, this was one of the first ways distributed deep learning was achieved. This is currently still the most popular form of paritioning.

The scaling of this approach is defined by the minibatch size N. Backpropagation can be carried out in parallel once receiving outputs from the forward pass, which is also carried out in parallel. The weight / parameter update phase requires that the resulting parameters of all the partitions to be averaged with respect to the entire minibatch and sent to each participating node as each has a replicated model of the entire neural network. This requires an *AllReduce* operation.

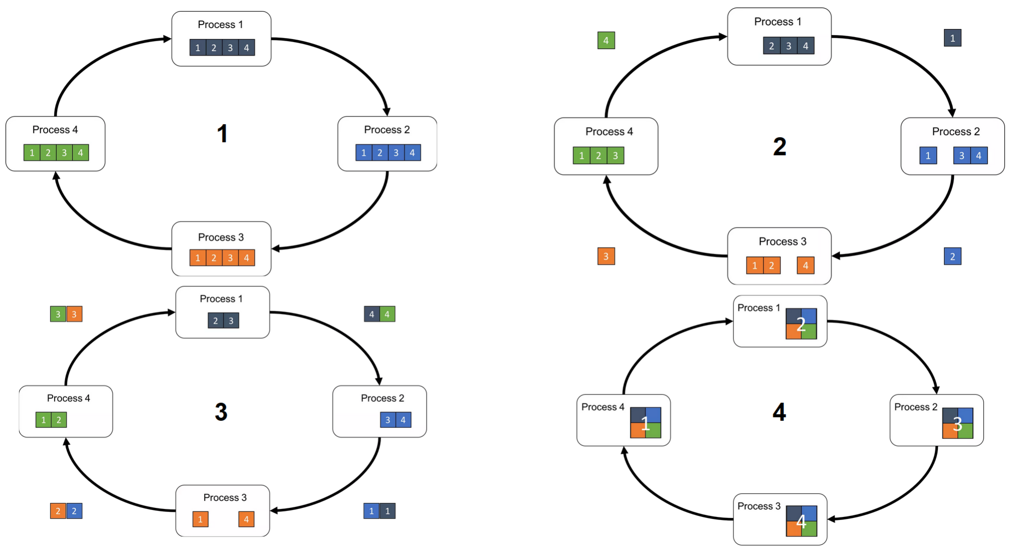
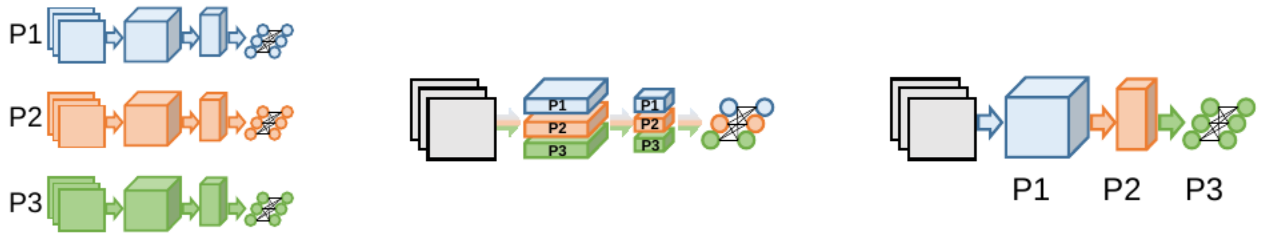


Figure 5: *AllReduce* ring-based structure

Operator Type	Eval.	Work (W)	Depth (D)
Activation	y	$O(NCHW)$	$O(1)$
	∇w	$O(NCHW)$	$O(1)$
	∇x	$O(NCHW)$	$O(1)$
Fully Connected	y	$O(C_{out} \cdot C_{in} \cdot N)$	$O(\log C_{in})$
	∇w	$O(C_{in} \cdot N \cdot C_{out})$	$O(\log N)$
	∇x	$O(C_{in} \cdot C_{out} \cdot N)$	$O(\log C_{out})$
Convolution (Direct)	y	$O(N \cdot C_{out} \cdot C_{in} \cdot H' \cdot W' \cdot K_x \cdot K_y)$	$O(\log K_x + \log K_y + \log C_{in})$
	∇w	$O(N \cdot C_{out} \cdot C_{in} \cdot H' \cdot W' \cdot K_x \cdot K_y)$	$O(\log K_x + \log K_y + \log C_{in})$
	∇x	$O(N \cdot C_{out} \cdot C_{in} \cdot H \cdot W \cdot K_x \cdot K_y)$	$O(\log K_x + \log K_y + \log C_{in})$
Pooling	y	$O(NCHW)$	$O(\log K_x + \log K_y)$
	∇w	–	–
	∇x	$O(NCHW)$	$O(1)$
Batch Normalization	y	$O(NCHW)$	$O(\log N)$
	∇w	$O(NCHW)$	$O(\log N)$
	∇x	$O(NCHW)$	$O(\log N)$

Figure 6: Complexity of various neural network layers [1]



(a) Data Parallelism (b) Model Parallelism (c) Layer Pipelining

Fig. 14. Neural Network Parallelism Schemes

Figure 7: Partitioning strategies [1]

Traditionally this was carried out using a parameter server to gather, average and broadcast the parameters to all nodes. More recently *AllReduce* tree and butterfly methods have been used. Most recently ring-based methods have been shown to have superior scalability. Industry examples of this method can be seen in Nvidas NCCL"Nickle" libary and Horovod[10], a distributed training framework for TensorFlow, Keras, PyTorch, and MXNet. The implementation that is carried out here attempts to reproduce the comparison of the improved performance from using a ring based *AllReduce* operation versus a tree/butterfly based *AllReduce* operation.

One problem that persists in the data parallelism strategy is that of the batch normalisation, which requires a full synchronization when carried out. Weight normalization has been suggested as an alternative which decreases the depth D of the operator from $O(\log N)$ to $O(1)$, removing inter-dependencies within the minibatch.

Significant gains are also seen from fragmenting minibatches into microbatches that are decomposed on individual nodes to take advantage of hybrid CPU-GPU methods as a form of heterogeneous parallelism.

Model Parallelism

This divides model longitudinally allowing for entire forward and backward passes of a longitudinal segment to be carried out on an individual node. This model is effective but not as widely use used as it is significantly harder to implement and can vary according to the specific architecture of the network, for example a convolutional neural network and a recurrent neural network models will be partitioned very differently. This makes it difficult for researches to 'drop in' new models or perform hyper parameter searches.

Pipelining

This divides the model into cross sections by layer, allowing for a section of the forward or backwards pass to be carried out on an individual node. This is a fairly new technique but suffers from similar problems as model parallelism when it comes carrying out rapid variations.

1 Experiment and implementation

Experiment

The goal of the experiment was to, using a data parellism approach, reproduce elements of the following results published by Steffen Rochelat at Apache MXNet.

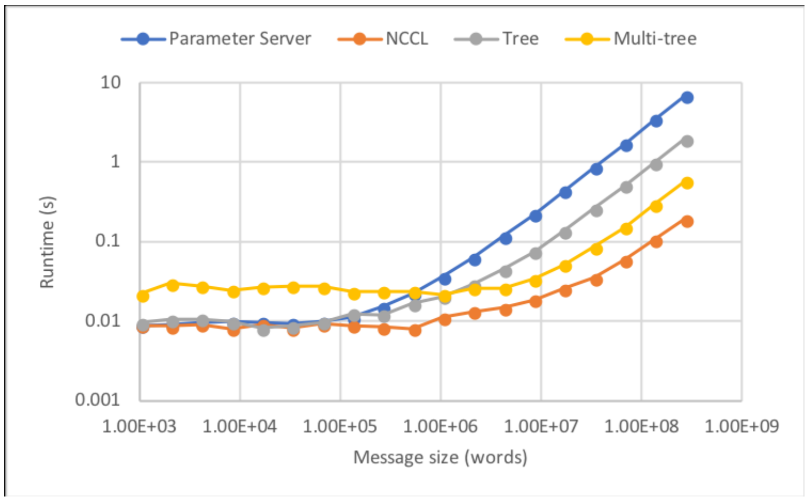


Figure 8: Results to reproduce

Specifically reproducing the divergence of the Tree-based *AllReduce*(Tree) and the ring-based *AllReduce*(NCCL). That is to show that the ring-based *AllReduce* scales more effectively by measure of runtime T with increasing size of messages γm sent between nodes.

Above and beyond reproducing these results, further hetrogenous parallelism with CUDA on individual nodes was attempted, concurrently for each node run on the cluster. The matrix multiplication of the forward pass was made parallel, assigning 1 GPU process per multiplication operation. While the program executed and the the CUDA kernel produce superficially reasonable outputs, only an accuracy of approximately 10% was achieved, that is as good as random.

OpenMP was considered for increasing speed of dataloading by breaking the data up into parts for each of the cpu cores to load into local memory independently.

Implementation

A 4 layer feedforward neural network, was implemented using the Genann neural network library for C. The neural network was train on the MNist dataset with 60000 images in the training set and 10000 images in the testing set. Training was carried out for 20 epochs. The communication layer between nodes was OpenMPI version 4.01 The message being passed between nodes was that of the weights of the network.

The size of the second hidden layer of the network was parameterized to provide variability in network size, that is layer breadth and therefore message size.

A standard OpenMPI AllReduce, using a tree/butterfly-based reduced, from here on referred to as "allreduce" was implemented. Additionally a synchronus and an asynchronus ring-based AllReduce was implemented "allreduce ring" and "allreduce ring async" respectively. The synchronus ring using MPI Send and MPI Recv operations and the asynchronus using MPI Isend and MPI recv operations.

Experiments were carried out on the University of the Witwatersrand computational cluster across 10 machines each with 8 cores, providing 80 nodes to pass messages between. The number of nodes was also varied for consistency. Each machine comprises an Intel core i7 CPU, Nvidia 1060 GTX GPU, 8 GB RAM.

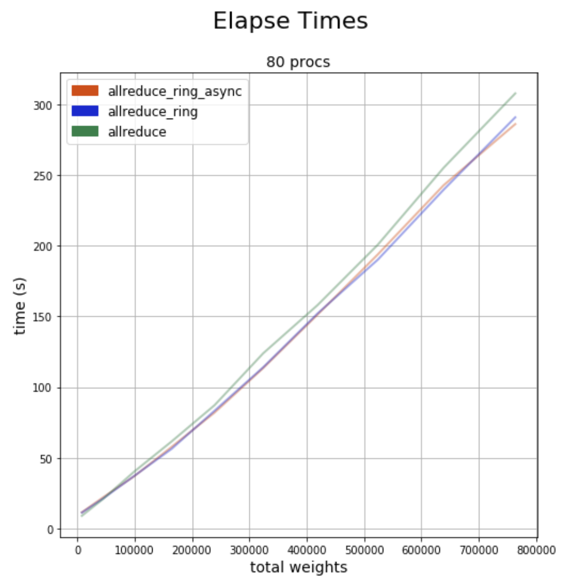


Figure 9:

2 Results

Interpretation
The average across multiple experiments was used for results.

It can be seen that across multiple nodes / processes ($procs$) that "allreduce" tree/butterfly (green) operation scales worse than the "allreduce ring"(blue) and the "allreduce ring async" (orange) operations.

As the number of processes decreases this advantage diminishes. For experiments with using more proces than that available from the cluster, that is more than 80, scheduling / queuing of processes had a negative effect on performance in terms of runtime and accuracy. This effect was more pronounced for the "allreduce" tree/butterfly operation.

Figure 16 shows the average accuracies for experiments run on 80 processes. Across all experiments the upperbound for accuracy of the ring-based algorithms was consistently 7-10% lower than that of the tree/butterfly-based algorithm.

The reason for this is not entirely clear, however it is possible that a it was produced by a flaw in the code, however it is also possibly caused by aforementioned "bracketing" and rounding errors. This needs to be investigated.

3 Application

Ring-based *AllReduce* are specifically well suited for highly distributed network structures. This could be used in conjunction with federated distributed deep learning in the context of African health care. One could imagine a system where health practitioners across the continent using mobile devices carry out local data collection and training, sharing only the parameters. This has benefits for patient privacy preservation and low cost severless research collaboration across the continent.

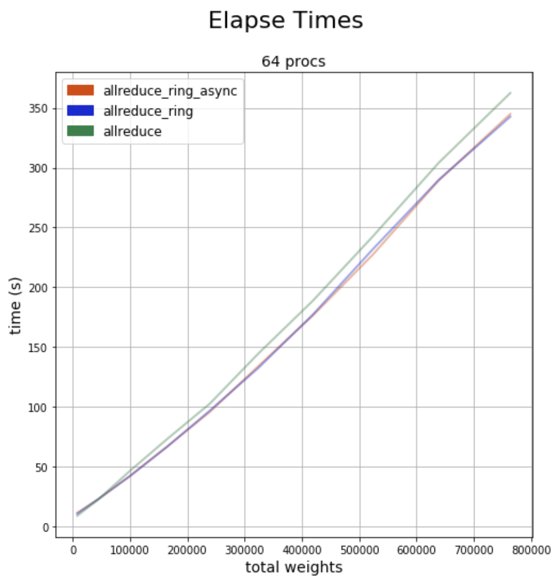


Figure 10:

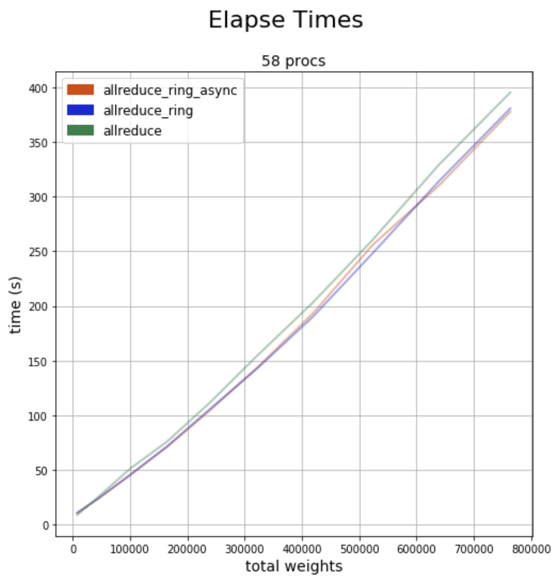


Figure 11:

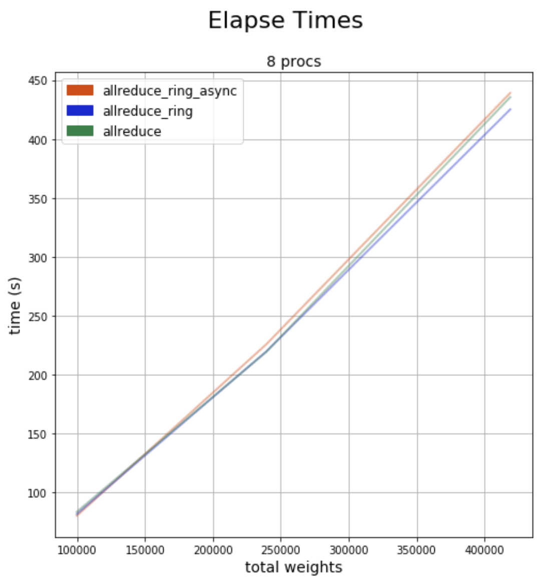


Figure 12:

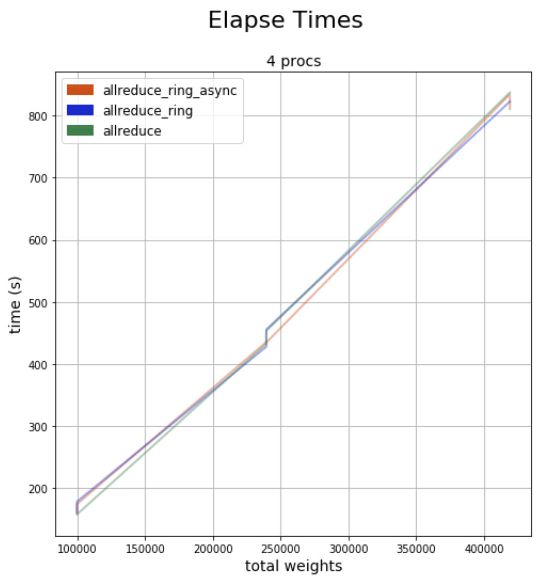


Figure 13:

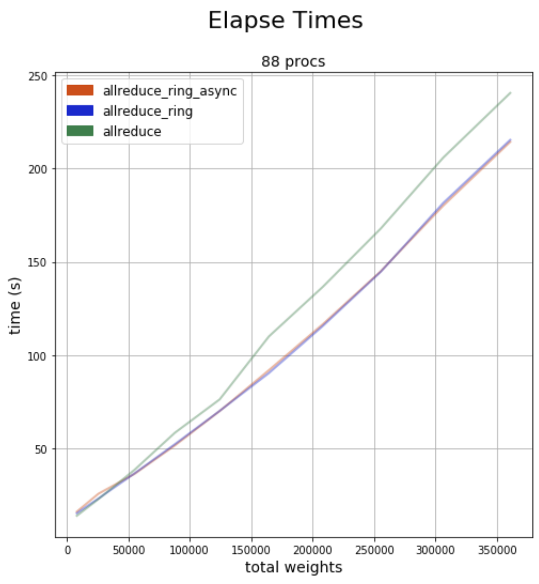


Figure 14:

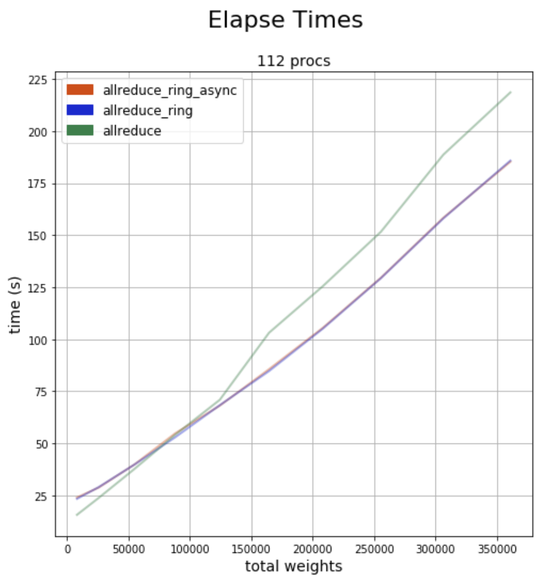


Figure 15:

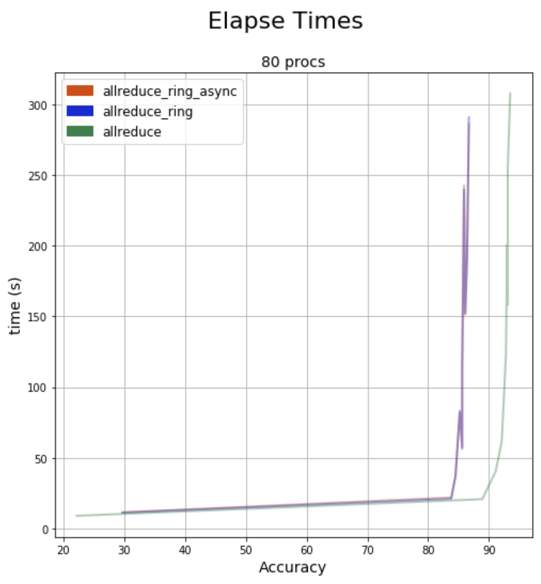


Figure 16:

4 Conclusion

The general principal of the experiments by Steffen Rochelat et al. were reproduced with data parallelism using MPI successfully carried out. There was however a 7-10% variation in accuracy, with the ring-based *AllReduce* algorithms performing consistently worse than the tree/butterfly *AllReduce* algorithm.

Hetrogenous parallelism combing MPI and CUDA was partially successful, as the program ran concurrently on multiple nodes and produced outcomes, however the accuracy was that of random after training.

References

[1] T. Ben-Nun and T. Hoefler. Demystifying parallel and distributed deep learning: An in-depth concurrency analysis. *arXiv:1802.09941 [cs]*, Feb 2018. arXiv: 1802.09941.

[2] R. D. Blumofe and C. E. Leiserson. Scheduling multithreaded computations by work stealing. *Journal of the ACM (JACM)*, 46(5):720–748, 1999.

[3] R. P. Brent. The parallel evaluation of general arithmetic expressions. *Journal of the ACM (JACM)*, 21(2):201–206, 1974.

[4] E. Chan, M. Heimlich, A. Purkayastha, and R. Van De Geijn. Collective communication: theory, practice, and experience. *Concurrency and Computation: Practice and Experience*, 19(13):1749–1783, 2007.

[5] D. Culler, R. Karp, D. Patterson, A. Sahay, K. E. Schauser, E. Santos, R. Subramonian, and T. Von Eicken. Logp: Towards a realistic model of parallel computation. In *ACM Sigplan Notices*, volume 28, pages 1–12. ACM, 1993.

[6] T. Hoefler and D. Moor. Energy, memory, and runtime tradeoffs for implementing collective communication operations. *Supercomputing frontiers and innovations*, 1(2):58–75, 2014.

[7] P. Patarasuk and X. Yuan. Bandwidth optimal all-reduce algorithms for clusters of workstations. *Journal of Parallel and Distributed Computing*, 69(2):117âĂŞ124, Feb 2009.

[8] J. Pjesivac-Grbovic. Towards automatic and adaptive optimizations of mpi collective operations. page 154.

[9] J. Schmidhuber. Deep learning in neural networks: An overview. *Neural Networks*, 61:85âĂŞ117, Jan 2015. arXiv: 1404.7828.

[10] A. Sergeev and M. D. Balso. Horovod: fast and easy distributed deep learning in tensorflow, 2018.