15-418 2021 Spring

Diego San Miguel (dsanmigu)
Nathan Ang (nathanan)

Distributed Machine Learning Using MPI and CUDA

Final Project Report

May 12th 2021

Summary

We sped up the training of a logistic regression machine learning model. Our implementation utilized MPI and CUDA to achieve 6x speedup when using 8 cores on the GHC machine.

Github link: https://github.com/diegofinni/distributed-ml-mpi

Background

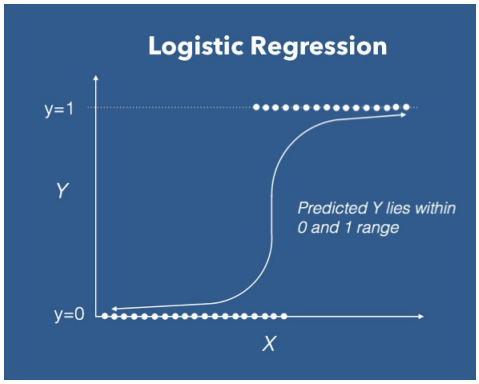
One of the most significant challenges of deep learning is the very large amount of time necessary to train models. Distributed machine learning is the practice of using multi-node computer systems to train machine learning algorithms in order to improve performance, accuracy, and the size of data inputs the system is capable of handling. At its core, distributed machine learning seeks to partition and parallelize the training of a machine learning algorithm.

In learning algorithms, we aim to minimize a loss function (function that indicates how well our model is doing, lower error is better) by slowly changing our models parameters each iteration until the algorithm converges, usually when we are happy with our error or are not improving anymore. Each iteration is dependent on the outputted parameters from the last one.

This last piece of information highlights the main problem in distributed machine learning which we tackle in our project. How can we parallelize a learning algorithm that requires every iteration to be done sequentially? There are two high-level general approaches that an engineer can take when solving this problem: the centralized method, and the decentralized method.

Logistic Regression

Logistic regression models are used to model the probability of a binary event occurring, meaning that there are only two labels possible, yes or no. We chose this model since it is simple to implement and trivial to speedup. We implemented our own basic logistic regression training algorithm (lr.cpp) from scratch. We decided to do this because the libraries we researched during the Checkpoint made it difficult to access the

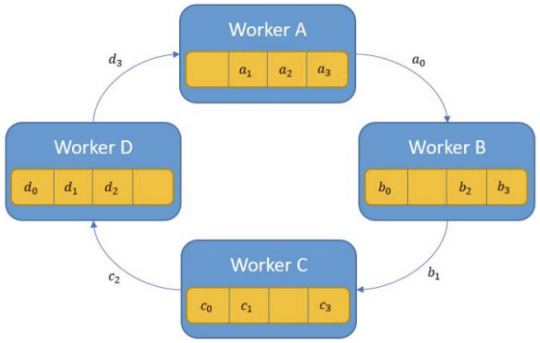


parameters in an easy way. We will be training our model on large datasets to accurately predict if the next day will have rain, if a movie review is positive or negative, etc.

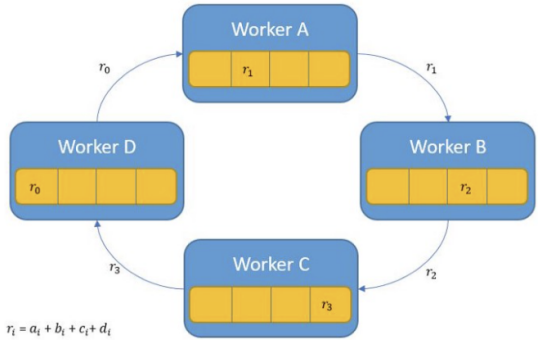
Decentralized Method - Ring all-reduce

In the decentralized method, all nodes in the system contribute equally to the work. If we have N nodes, we shard the dataset into N pieces and each node is responsible for training its model on that piece of the dataset. In order to faster learn the correct parameters and converge faster, we have to share the gradients learned during each iteration between all nodes using an algorithm called all ring reduce.

Ring all-reduce is a decentralized communication algorithm that allows our nodes to efficiently communicate and reduce all the gradients in the model with the least amount of information transmitted across the system. It works in two phases:



1. Each process p sends data to the process (p + 1) % P (P = num of nodes). The array of data of length N being shared is divided by p, where each process is responsible for sending one of the data chunks. This process of sharing occurs p - 1 times and each time a process receives a piece of data it applies the reduce operator to it.



2. Once each process holds a complete reduction of their share of the data, the share-only step begins. Each process forwards the data it has in a ring-like fashion p - 1 times such that each process will finish with all of the reduced data.

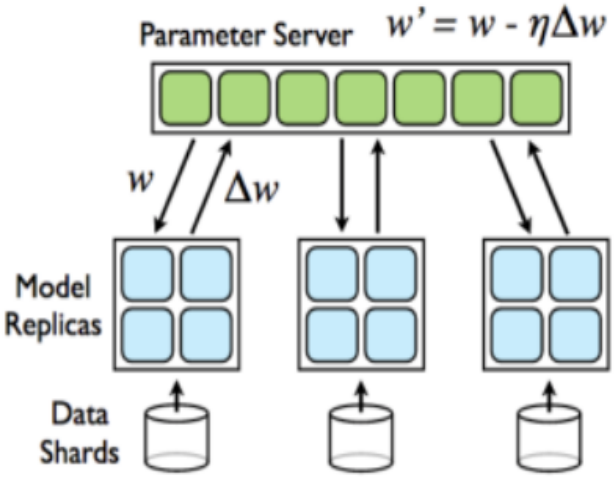
```
grad = gradient(net, w)

for epoch, data in enumerate(dataset):
  g = net.run(grad, in=data)
  gsum = comm.allreduce(g, op=sum)

  w -= lr * gsum / num_workers
```

The high level pseudocode showing how we use ring all reduce in the training of the model is shown on the left.

The decentralized method allows all N workers to contribute equally to the workload, however, it does have one drawback. Given that ring all reduce requires all workers to cooperate together, we have created a hard barrier at every iteration. All workers must complete an iteration before any can start the next one, we have essentially made computation during communication impossible, and vice versa. This drawback is addressed in the centralized method.

Centralized

In the centralized method we have one master parameter server and N worker servers. The N worker servers shard the dataset into N pieces and are each responsible for training on one piece. During each iteration, a worker server will request the most recent parameters from the master server, compute the gradients for that iteration, and then send back the gradients. When the master server receives gradients from a worker, it applies the product of the gradient and the learning rate to its list of parameters.



Since workers only have to communicate with the master server when moving from one iteration to the next, we can allow for some workers to be on different iterations than each other, essentially allowing for stragglers to exist without slowing down the progress of the whole system. This is called the stale synchronous parallel model of computation [2] and it has 3 main tenets:

1. Bounded staleness: Data that is read by a node may be stale.
2. Read-my-writes: If a node updates a value, all subsequent read( ) operations by that thread will see that update.
3. Soft synchronization: At the end of each iteration, nodes execute a “soft barrier.” A soft barrier blocks a node until all nodes are within a specified range of the current iteration.

Approach

Parallel Programming Framework

We chose MPI to be our parallel programming framework that would implement the aforementioned centralized and decentralized architectures. We chose MPI because originally our intention was to use multiple machines in the GHC cluster. This would allow us to have access to multiple GPUs which could speed up our training process significantly. However, due to technical difficulties that we and the course staff were not able to solve, using multiple machines became impossible.

Despite this setback, MPI was still the most viable parallel programming framework for our project for one main reason. The centralized and decentralized architectures were designed assuming that nodes did not share an address space, and that they would communicate through messages. This made MPI the easiest framework to use in implementing our architectures, increasing our programmer productivity.

Decentralized

The decentralized program, lr_decentralized, is compiled and run on the command line like so:

```
$ mpic++ main.cpp decentralized.cpp lib.cpp lr.cpp
$ mpirun -np N ./lr_decentralized num_epoch, learning_rate
```

main.cpp: Main program where processes init their MPI environments, discover their rank, launch the decentralized algorithm, then terminate once finished

decentralized.cpp: Contains all the code surrounding ring-all reduce and MPI communication

lib.cpp: Contains declarations of some types and global variables used by the workers. Most notably it declares the reduce function type like so:

```
typedef void(*ReduceFunction)(double* dst, const double* src, int n);
```

This allows us to pass in pointers to a programmers own defined reduce function when calling the ring-reduce function

lr.cpp: Contains our parallelizable implementation of a logistic regression model

num_epoch: Number of iterations, or epochs, that each node will compute

learning_rate: The learning rate of the logistic regression model (also known as alpha)

The main function in decentralized.cpp that is called by main is ringreduce( ). When this function is called a single round of ring allreduce is held so that nodes can share their data. Every worker node first completes an iteration of the logistic regression model and then populates their buffer with the gradients calculated on that iteration. Then they hit an MPI_Barrier, this ensures that no nodes start the communication phase until they have all finished the computation phase. Then they communicate synchronously with their neighbors using MPI_Send( ) and MPI_Recv( ) functions. Once the reduction phase is over, they use the reduced gradients in their buffer and apply it to their parameters using the learning rate specified in the command line. This process occurs num_epoch times.

Centralized

The decentralized program, lr_decentralized, is compiled and run on the command line like so:

```
$ mpic++ main.cpp master_node.cpp worker_node.cpp lr.cpp
$ mpirun -np N ./lr_centralized num_epoch, n_bound, learning_rate, file
```

main.cpp, lr.cpp, num_epoch, and leearning_rate are the same as before. main.cpp is slightly changed so that the process with rank 0 uses the master_node functions, and every other rank uses the worker_node functions.

master_node.cpp: Contains logic that master_node executes to manage and communicate with all workers

worker_node.cpp: Contains code that worker_node executes to communicate with the master server and run iterations of the logistic regression model

n_bound: Bound of staleness that the system will tolerate.

In the main function, all processes except the one with rank 0 will become worker_nodes. Worker_nodes will read in their shard of data and initialize their parameters the same as everyone else. After each iteration that they compute they will send synchronously send their gradients to the master_node using MPI_Send( ) and then synchronously wait for a response from the server containing the new parameters using MPI_Recv( ).

The master_node at all times has asynchronous receive requests open for all worker_nodes using MPI_Irecv( ). In its main while loop, the master_node( ) continuously waits to see which requests have been fulfilled using MPI_Testany( ).

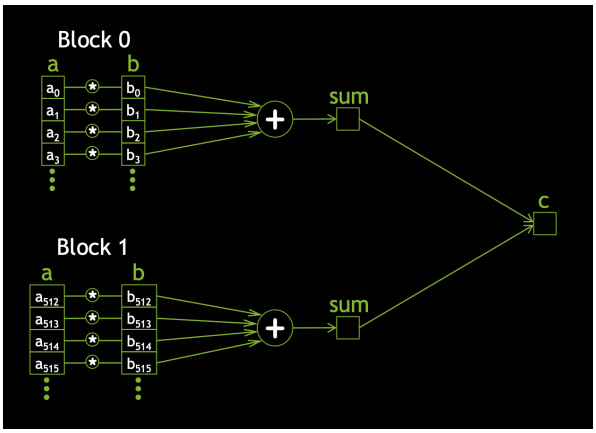
When a request is fulfilled, the master_node will read in the new gradients and apply them to its parameters. After this the master_node will update its global variable `int *iters`. This is an array of ints that keeps track of which iteration every worker_node is on. The master_node can use this array to determine which workers can be unhalted after the latest request was fulfilled and if the worker that completed the request must be halted. Halted and unhalted workers are stored in sets, workers currently working are in neither set.

When a worker has completed all num_epoch iterations, the master_node will decrement its global variable `int active_workers`. Once this variable has reached 0, the master_node will print out its parameters to stdout and then terminate. All workers also terminate immediately once they have finished all their iterations.

Parallelizing At A Finer Grain, With CUDA

We have parallelized the training of the model by essentially dividing up work and solving the complications that arise, as mentioned. However, there are also sequential portions of training at a finer grain that we can parallelize. One major portion is a dot product. To make a prediction with our parameters, we must take the dot product between the parameters and a data row. This means that every data row, we iterate through the entire size of our parameter vector, which is not time efficient when we have thousands of parameters.

Thus, we can parallelize this process using our GPU with CUDA. GPU's are good for this process due to the advantage of having more cores which will result in higher throughput during our computation. Using CUDA's C++ library *Thrust*, we implemented this dot product part of the process in parallel.



Results

Overall, our project was successful. We defined performance via runtime and speedup.

For our experimental setup, we held independent variables, such as the number of cores, and measured how it affected our dependent variable performance. These variables will be clear given the following tables and graphs.

The datasets that we used were found on Kaggle:

- Large sized, 14000 parameters, 1200 data points predicting if movie review is positive or negative.
- Medium sized, 12 parameters, 14,000 data points predicting next day rain.
- Small sized, 8 parameters, 800 data points, predicting if a patient has diabetes.

During testing, each node would read in the data file, shared the portion that it needed, start the timer, train the model, and then stop the timer. All benchmarks were run on a single GHC node which has the following specifications: 8-core, 3.0 GHz Intel Core i7 processor and a NVIDIA GeForce RTX 2080 B GPU

Decentralized

In order to be fair, our sequential implementation is not simply the decentralized architecture with 1 node, it is a fully sequential program that does not use MPI. Overall the speedup was less than linear and we see that as more cores are added the change in speedup decreases as well. The results from decreasing the number of iterations were uneventful, the decentralized method more or less had the same performance relative to the sequential version with any workload that wasn't tiny. Interestingly, the speedup was closer to linear when compared to running the decentralized method on one node.

Weather Dataset: 12 parameters, 14k data points, 1000 epochs

Number of Cores	Sequential (s)	Decentralized (s)	Speedup	Comm Time (ms)
1	86.937	88.958	0.989	N/A
2	86.937	44.431	1.957	3.007
3	86.937	30.206	2.878	4.164
4	86.937	23.094	3.764	6.500
5	86.937	18.740	4.639	6.946
6	86.937	15.573	5.583	9.312
7	86.937	14.734	5.900	16.213
8	86.937	13.998	6.211	25.312

Figure 1

At first we suspected that a majority of the time lost would be due to communication between nodes and the MPI_Barrier. In order to test this theory, we placed timers around the parts of our code where nodes would call the ringreduce function. From figure 1, you can see that we found that around 0.01% of our time was spent communicating between nodes, meaning this was clearly not our bottleneck.

This also helped us confirm that data transfer and memory bandwidth were not our bottleneck. Our nodes were only sending 12 ints across the bus for each message in ringreduce. Given the bandwidth of the intel core i7 the GHC is equipped with, it is safe to say this was not problematic.

In order to confirm our findings, we modified our code to remove the MPI_Barrier and all ringreduce calls, effectively making it such that each node trains its own model on a piece of data 1/N the size of the original. After that, we placed timers before and after each call to an iteration. We then printed out the average amount of time that it took for each node to complete an iteration after it completed all the epochs.

What we found was that the time it took to do an iteration did not scale exactly linearly with how large the dataset was. Dividing the size of the dataset by 2 would actually only create an iteration speedup of about 1.9, and this trend repeats as N grows. Data shown in Figure 2.

Data size	N/1	N/2	N/3	N/4	N/5
Time (ms)	85.3	44.870	31.36	23.448	18.652
Speedup	1	1.90	2.72	3.63	4.57

Figure 2: Ran weather dataset on decentralized and sequential algorithm with 20 epochs. Time represents the average amount of time it took for each node to complete an iteration. Data size indicates the size of the data used in each iteration where N is the entire dataset. (Implicitly you can see that the number N is divided by is the number of nodes in the system)

At this point we are not fully certain why the time for each iteration does not scale perfectly linearly with the size of the data set, and this is where our speculation begins.

If we breakdown the algorithm for an iteration of gradient descent, it looks like the following:

```
for each row:
            t = dot_product(params, row_x)
            sig = Sigmoid prediction(t)
            update_gradient(sig, row_y)
update_theta(gradient)
```

We can see that if we decrease the size of our data, we are increasing the number of times that we call update_theta( ), thus increasing the total amount of computation that a single node does. Thus, we are aggregating some more computations for the sake of doing many more in parallel, causing us to gain nonlinear speedup.

Centralized

Just like in the decentralized version, we made sure to compare our centralized architecture with an implementation that is purely sequential and does not use MPI. Our speedup was very similar to that of the decentralized method, so long as you accounted for the fact that there was always one less worker in the system. Again, we were unable to achieve a perfectly linear speedup, and instead followed a line closer to 0.9N, where N is the number of workers in the system.

We placed timers around communication code in order to sum up total time spent waiting on synchronous messages and sending data, but like in the decentralized case, also found that MPI communication only took a trivial amount of time. This makes sense when we think in terms of arithmetic intensity. The number of floating point operations required for a single iteration of

gradient descent on a dataset with 14,000 points outweighs the number of gradients and parameters sent through MPI in each iteration by several orders of magnitude.

Weather Dataset: 12 parameters, 14k data points, 1000 epochs

Number of Cores	Sequential Runtime (s)	Centralized Runtime (s)	Centralized Speedup
1	86.937	N/A	N/A
2	86.937	85.296	**1.01**
3	86.937	46.116	**1.885**
4	86.937	31.367	**2.771**
5	86.937	24.484	**3.550**
6	86.937	20.38	**4.265**
7	86.937	17.915	**4.853**
8	86.937	15.704	**5.536**

Figure 3. Number of Cores vs Centralized Speedup on medium sized dataset

What we noticed as well was that the advantage of masking stragglers was barely being utilized in our test scenarios. The differences in computation time between having a bound of 10, to a bound of 0 were so small that we found they were not worthwhile to include, and were likely within margin of error.

After further research, we found that this was due to the fact that we had no real stragglers. Our work pieces are a single iteration of gradient descent on a data set size that is exactly equal for each node. For this reason, stragglers were extremely uncommon and nearly impossible to reproduce. The bounded delay paper mentioned in the background sections notes how, in order to prove the usefulness of bounded delay in completing computations quicker, the researchers spoofed stragglers, since creating them naturally in a single node environment was very difficult.

We decided to replicate this the exact same way they described doing so in their paper. Each node created its own seed and would pull a random number from it. If the number were divisible by 5, the node would sleep for 1 second. After incorporating this fake straggler code, we found that increasing the bound had a significant effect on total computation time, see figure 4.

Dataset: 12 parameters, 14k data points, 100 epochs

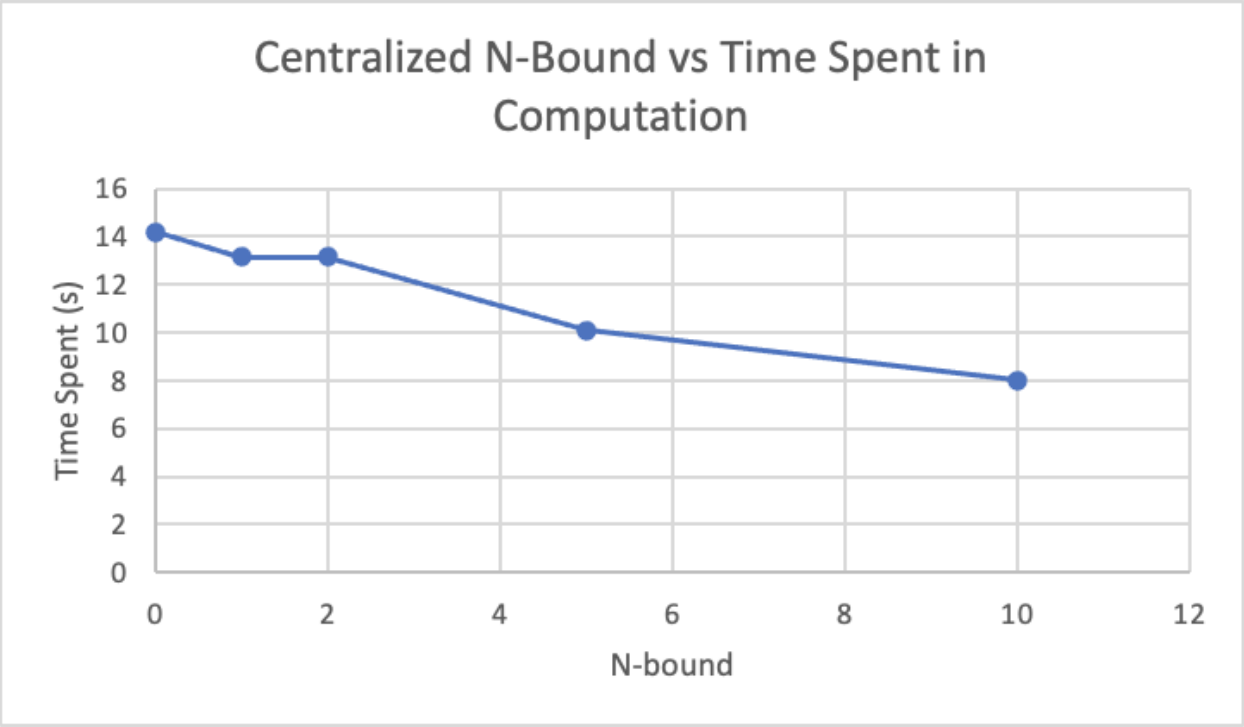


Figure 4.

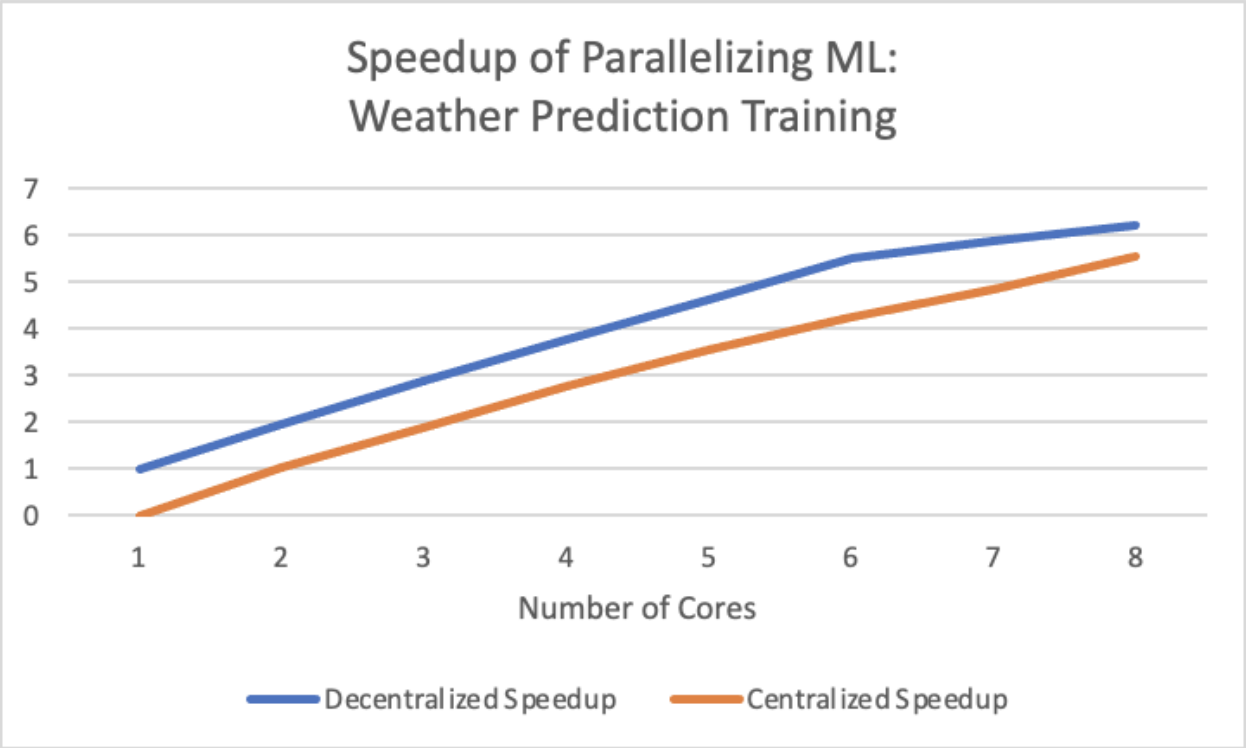


Figure 5: Speedup on training model based on medium sized dataset. We can see that parallelization through MPI gives us better performance with more cores.

We observed relatively good speedup on performance by parallelizing the Machine Learning process. Now, how can we improve upon this?

As mentioned before, we implemented parallelization at a finer grain, with our machine's GPU with CUDA. Specifically, we will parallelize the dot product that occurs when processing each data point.

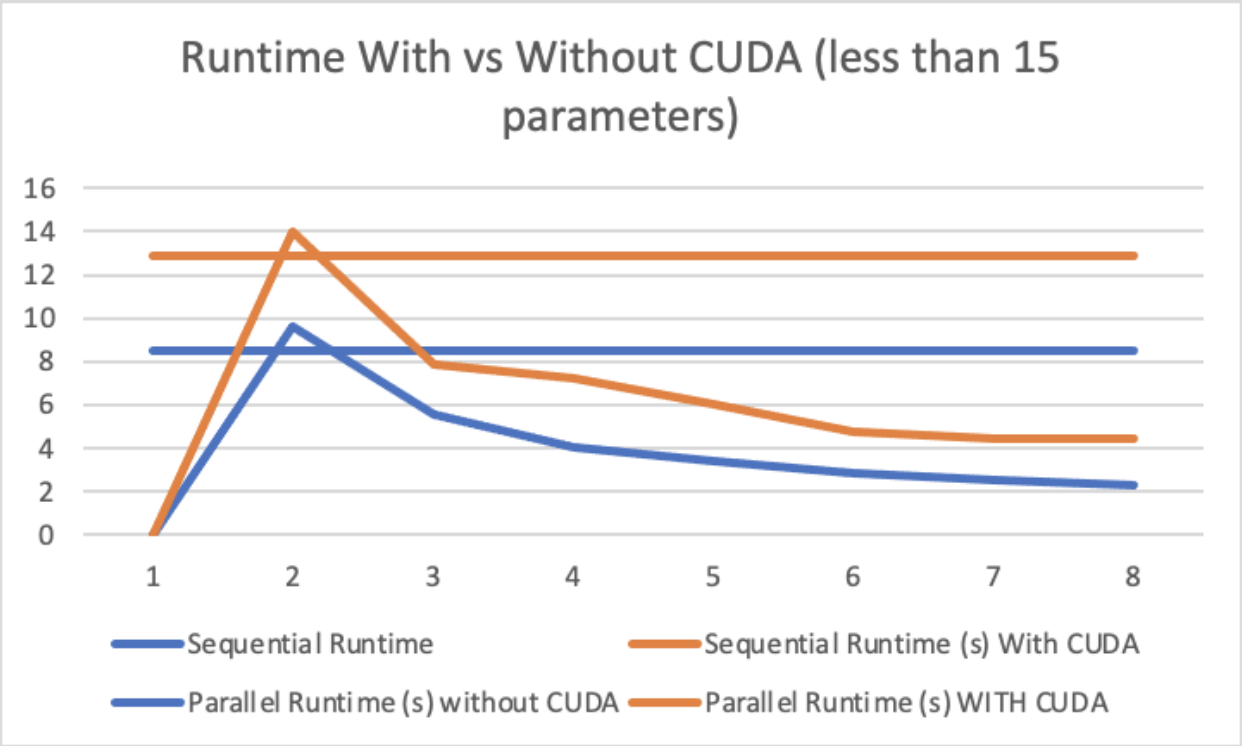


Figure 6: Runtime With vs Without CUDA shows slowdown when parameters are too few for GPU utilization

Here we observe that CUDA actually makes our performance worse. This is where understanding how parallelization fits into the problem as well as **problem size**, a topic we have covered extensively in 15-418, comes into play. For each data point, we are using the GPU to parallelize the computation of less than 15 parameters. This is inefficient, where the overhead of using the GPU and CUDA makes our performance worse than the sequential version.

In order for this parallelism to be effective, we need to be using more parameters. Thus, here are our results on a dataset we parsed into a bag-of-words model with 13,194 parameters.

Training Model for Movie Review Polarity Prediction

Dataset: 13,194 parameters, 1200 data points,

1000 epochs

Number of Cores	Sequential Runtime	Sequential Runtime (s) with CUDA	Parallel Runtime (s)	Parallel Runtime (s) with CUDA	**Total Speedup**
1	76.458	65.153	59.509	49.087	**1.558**
2	76.458	65.153	39.694	34.772	**2.199**
3	76.458	65.153	27.439	23.805	**3.212**
4	76.458	65.153	21.391	18.715	**4.085**
5	76.458	65.153	18.268	15.818	**4.834**
6	76.458	65.153	16.75	14.558	**5.252**

Figure 7: Data of Runtimes with and Without CUDA on dataset with thousands of parameters

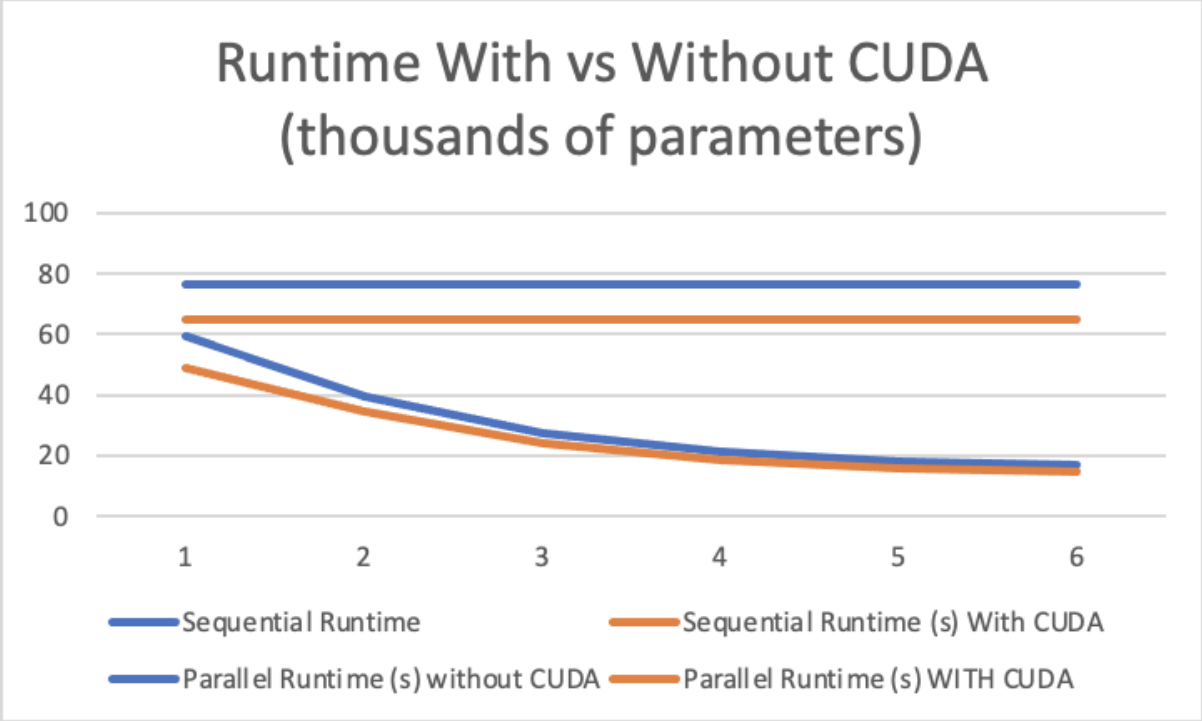


Figure 8: Runtime With vs Without CUDA shows speedup when number of parameters is sufficient for GPU utilization

Here, we observe that now CUDA improves our performance due to the large amounts of parameters.

These are our final results on datasets of varying sizes with CUDA.

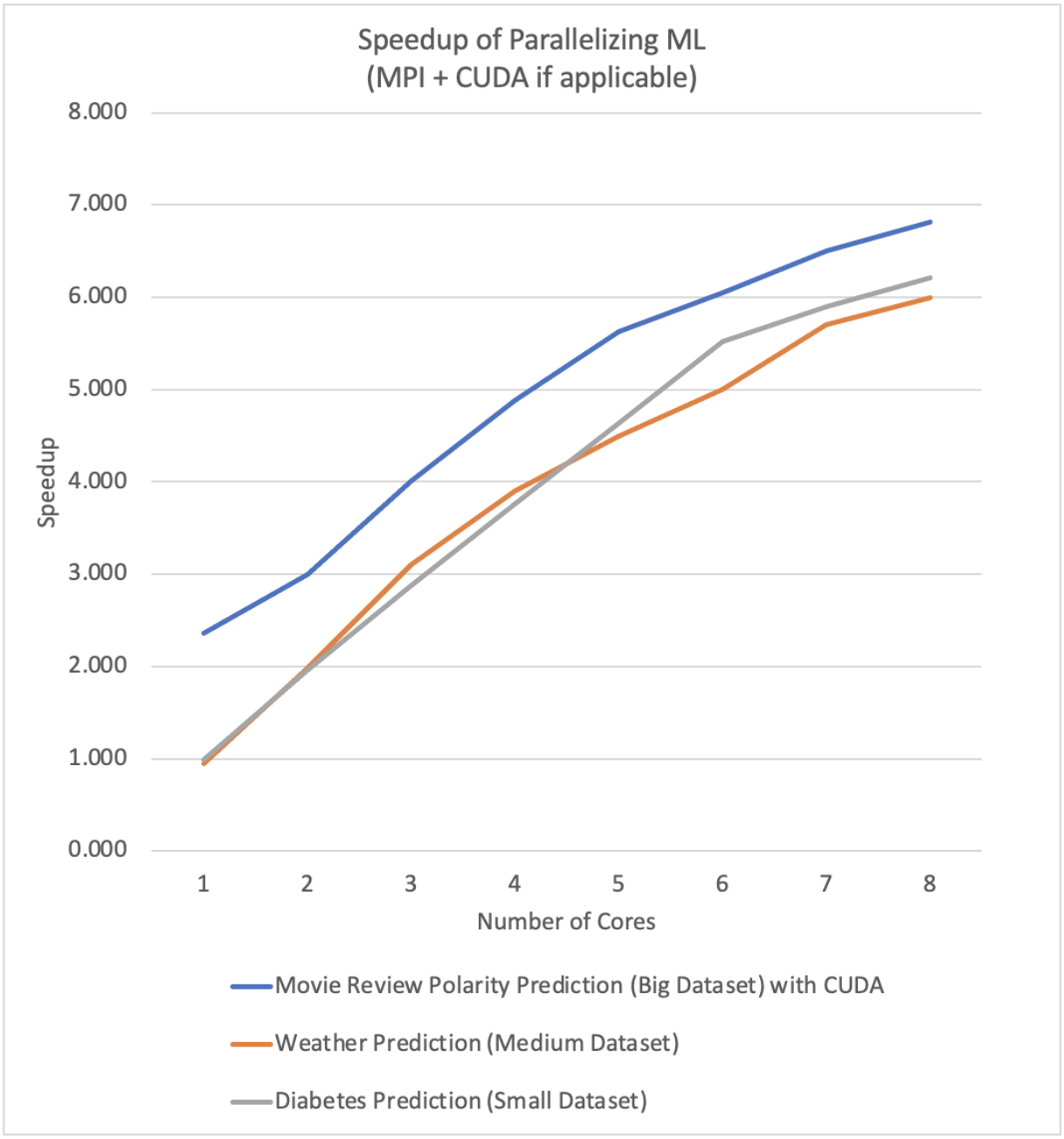
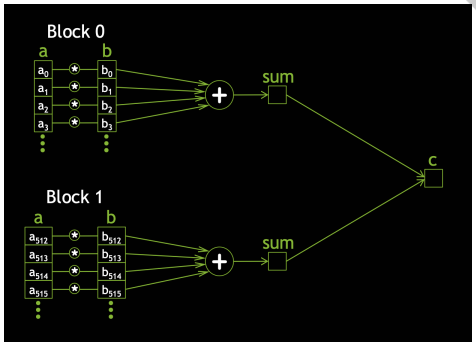


Figure 9: Speedup of Distributed ML on GHC machines with datasets of varying sizes.

Given all the previous analysis, CUDA gives us an extra boost in speedup. This is because the dot product is now happening in parallel, yet it only is applicable if our parameter vector size is large enough. One limiting factor of the dot product is the inherently sequential part of the reduce. Given Amdahl's Law, this will inhibit us from achieving linear speedup on the dot product in isolation, but we can still have speedup related to number of cores when we reduce in chunks (per block), utilizing parallel computation.



Additional Analysis:

Was your choice of machine target sound? (If you chose a GPU, would a CPU have been a better choice? Or vice versa.)

We were able to apply what we learned in class effectively when choosing a target machine. We are sharding the data and distributing to different nodes, and this is an excellent fit for MPI, given that each node does not share an address space. Each node gets its own shard of data to train on, and thus we are able to parallelize this training process at this level.

One layer lower regarding per-data-point computation, however, we are dealing with a basic operation on vectors with thousands of parameters. This, as we learned in class, is a fit for the GPU given its advantage of having much more cores than the CPU, thus we can do this large number of computation in parallel. Because the throughput within our vector computation is increased, our performance increases. If this were done on the CPU, it would not have enough cores relative to the GPU to compute as quickly.

Given our results, we believe that our decision of using a mix of the CPU and GPU was sound.

If we had more time, what would we do?

Our biggest setback during this project was being unable to run MPI jobs on multiple nodes in the GHC cluster. If we were able to do this we could fully take advantage of two layers of parallelization. The first layer would be different MPI nodes being responsible for different shards of the data, and the second layer would be the nvidia GPU's on each node parallelizing

the training of each model. With our current setup we only have one GPU and 8 nodes in our system that share the GPU.

Division of Work

Equal work was performed by both project members. Diego focused mostly on the MPI work with the centralized and decentralized architectures, while Nang focused on implementing the parallelized machine learning models and connecting them to the aforementioned MPI architectures.

References

[1]https://towardsdatascience.com/visual-intuition-on-ring-allreduce-for-distributed-deep-learning-d1f34b4911da

[2] https://www.pdl.cmu.edu/PDL-FTP/associated/hotOS-13-cipar.pdf

[3] https://docs.nvidia.com/cuda/thrust/index.html

[4] https://www.nvidia.com/content/GTC-2010/pdfs/2131_GTC2010.pdf