ASYNCHRONOUS DISTRIBUTED DEEP NEURAL NETWORK TRAINING AT SCALE

FINAL REPORT

Yannick BLOEM
Department of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213
ybloem@andrew.cmu.edu

Yves ZUMBACH
Department of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213
yzumbach@andrew.cmu.edu

May 6, 2019

Keywords Neural Network · Distributed Systems · Stochastic Gradient Descent · Cloud Computing

ABSTRACT

This report documents the efforts conducted in order to implement a cluster that would distribute the training of densely connected neural networks in the cloud using 1-bit quantization compression to alleviate the communication bottleneck.

1 Summary

We implemented an asynchronous distributed deep neural network training cluster which achieved close to linear training speedup in the number of nodes, while maintaining the same error loss.

2 Background

The algorithm described in [8] pursues the goal to enable distributed asynchronous deep neural network training at scale. The previously known ways of distributing DNN training were either to train multiple DNN on a subset of the data and average the final weights, or to use a master-slave architecture where the master distributes the work to the slaves and aggregates intermediate computations. Both these solutions have drawbacks, mainly synchronization requirements and communication bottleneck when transmitting all the NN weights for parameter averaging, and the communication bottleneck for the centralized parameter server in asynchronous SGD. By using an asynchronous peer-to-peer architecture, it is possible to remove the synchronization requirements and the central communication bottleneck.

The paper introduces a couple of crucial ideas that enable efficient peer-to-peer communications by reducing the size and the quantity of messages to send. First, the messages sent only include gradient elements that exceed a certain threshold value: τ, *i.e.* the elements that will have a significant impact on the neural network. This reduces the amount of elements to send by delaying the transmission of smaller gradient elements. Second, the gradient can be compressed using 1-bit quantization, where it represents either $+\tau$ or $-\tau$ (τ is defined as the threshold to cross to trigger the communication). This allows to package the gradient and the index of the gradient, i.e which weight it refers to, into a 32-bit integer: 1 bit for the plus or minus sign and 31 bits for the index, thus allowing to encode $2^{31} - 1 = 4\ 294\ 967\ 295$ different indexes which is more than enough for all modern NN. The scale and training speed in [8] are "the highest ever reported for fully connected DNN training." [8]

2.1 Pseudocode

The pseudo-code given in algorithm 1, found in [8], gives an idea of the code executed to train the NN.

Algorithm 1 Pseudo-code describing how to train the NN in a distributed fashion using 1-bit quantization

function UPDATE
  Receive and uncompress any weight update messages from other compute nodes and apply them to the local replica of the DNN.
  Load feature vectors and supervision targets for a mini-batch.
  Compute a sub-gradient $G(s)$ by BackPropagation.
  Aggregate the sub-gradient in the gradient residual $G(r) = G(r) + G(s)$.
  Reset the message map M.
  For each element $g_i(r)$ of $G(r)$:
  if $g_i(r) > \tau$ **then**
    push the pair $i, +\tau$ to the message M.
    Subtract τ from residual: $g_i(r) = g_i(r) - \tau$.
  else if $g_i(r) < -\tau$ **then**
    push the pair $i, -\tau$ to the message M.
    Add τ to the residual: $g_i(r) = g_i(r) + \tau$.
  Compress M and send to all other compute nodes
  Apply M to the local replica of the DNN.

2.2 Key data structures and operations

In our project, the key data structure is the list of parameters that define our neural network. A neural network is made of multiple layers of neurons. Our project focuses on deeply connected neural network, which means that each layer is generally fully connected to the next layer. If the hidden layers have N neurons, each pair of layers has N^2 connections linking them together. In our case, using an input layer of 13 neurons, 4 hidden layers of 200 neurons each, and an output layer of size 30, we achieve a total amount of trainable parameters of 168'600. In our structure, we have a couple of other important data structures. First of all, we keep a second list of parameters that has the exact same format as the weights defined previously, except that it contains what is called "residuals" in the pseudocode. The residuals are the changes to individual weights that haven't crossed the threshold τ after one batch. Instead of applying them to the neural network, we store them in a separate structure. After every batch, we add or substract the weight changes that would normally be applied to the NN to their corresponding entry in the residual list. If the resulting entry in the residual list is either larger than $+\tau$ or smaller than

$-\tau$, we apply $+\tau$ or $-\tau$ to the neural network, and remove so much from the residual list.

Our last important structure is the messages of deltas that have to be sent to all other nodes in our cluster. As mentioned above, at the end of each batch, after adding or substracting to the residuals list, we only apply the elements that surpass our defined threshold τ. For each element that qualifies, we send the newly applied to change to all the other nodes in the cluster, so they can apply it to their own network. This is required to have a NN that actually converges.

In order to avoid having a communication bottleneck with potentially huge messages sent over the wire, we have to compress our messages as much as possible. To achieve this, we use a technique known as 1-bit quantization, as described in the background section. Each weight to update can be compressed in a single 32-bit integer. At the end of every batch, we thus have a list of integers that get sent to all other nodes, that will in turn decompress it, and apply the threshold τ to the corresponding parameters.

Obviously, the main operations happening on our data structures are machine learning related. Each input into our network gets forwarded to the next layer based on the weights of each connection, and the activation function at each neuron. Once it reaches the output layer, it gets compared to the actual target output, also called "label". Based on the divergence between the output and the label, a loss function is used to compute the loss. Using stochastic gradient descent, the backpropagation algorithm then updates each weight and bias in the neural network to slightly "correct" it (i.e so that the next input will generate a slightly less wrong output).

2.3 Parallelization

In a sequential neural network, the computationally expensive operations are the forward propagation of an input, the loss computation[1], and the final backpropagation. These are extremely parallelizable operations because they are essentially massive matrix operations, which a GPU can compute very efficiently. More importantly, modern neural networks are computed using "batch gradient descent", which means that multiple inputs are batched together, the outputs are computed for each input at the same time, the loss is then computed for all outputs at the same time, and the whole neural network is updated once at the end of the batch. Since there are no dependencies between different pairs of input/output, these operations can be massively parallelized using GPUs. Libraries like `PyTorch` are fully compatible with CUDA and enable users to use GPU computations by turning on a simple boolean parameter.

This initial parallelization makes for huge speedup when training deep neural network, but is still intrinsically limited by the enourmous amount of training data to process[2]. It's not a solution to simply make larger batches because at some point the network will stop converging if we evaluate too many points at once without updating the network's weights with regard to the computed losses. Another idea is to parallelize the network training on multiple nodes, where each node trains on a subset of the input data. That is what we implemented in this project.

Finally, in the specification of our algorithm, a computationally very expensive task is the so-called "compression" of all the parameters in the network into a message to send to all other nodes in the cluster. Indeed, after the training on a batch is completed, we have to loop over every single weight in the network, update the residual list at the corresponding input, and if the sum passes the threshold, we have to compress a delta, and add it to the message to be sent. In a network with potentially millions of weights (which is very common, we kept our network very small for practical reasons), this operation, when executed fully sequentially, is unbearably slow. It's an operation which is entirely memory-bound, as the computation itself is very simple, but there are a lot of memory reads and writes. There are no dependencies between the different values, as the computation of a single compressed message is entirely independent of the other ones. Ultimately, all the messages have to be aggregated in one message, and this reduction operation is more difficult to achieve then simple parallel operations.

3 Approach

3.1 Technologies & architecture

Our technological stack is very diverse. Our project is made of multiple nodes that make the cluster and communicate over HTTP. Each node is comprised of:

- A communication service: The communication service is written in Scala and uses Akka for mes-

[1] To measure the performances of our cluster, we used a speech recognition NN that we implemented. Computing the loss for such NN requires special algorithm and is indeed computationally expensive.

[2] Standard speech recognition NN are trained on tens of gigabytes of data.

sage passing. Akka is an actor-based concurrency framework, built for the JVM, which enables the creation of clusters of actors that can communicate through message passing.

Once the communication service is launched, it will connect to a cluster based on a provided IP entrypoint. It will start interacting with all other nodes in the cluster.

It also sets up an asynchronous bi-directional TCP communication socket (also called channel) backed by a concurrent queue. This concurrent queue is required to buffer the messages received from other nodes in the cluster, before sending them through to the worker service. The communication service also receives messages from the worker service (see below for details about the worker service) like the delta messages to send, runtime information with the amount of time it took to process one epoch of training, how much loss was incurred, and other useful statistics. Receiving messages from the worker service did not need to be bufferized since this is automatically done by the underlying operating system.

The communication service is fully resilient to connection losses with the worker service: it will automatically start looking for a new connection if it was to fail at some point. This is very important because TCP connection can sometime fall but you do not want that it crashes the whole cluster after already hours of working.

- A worker service: The worker service is built in Python, and implements the actual neural network training using the PyTorch framework.

  As soon as it is launched, the worker service sets up a fully asynchronous TCP connection to the communication service using the `asyncio` library. It sets up multiple concurrent queues for both reading and writing messages to this TCP connection.

  After launch, it first expects to receive a message from the communication service informing it of the total number of nodes in the cluster, and which ID this node was assigned. This information enables the worker to compute the correct subset of the total data that has to be used for training. If there are n nodes in the cluster, it will train on $1/n$ of the total data set. Once it received this data, it sets up the neural network, and starts training. In order to achieve convergence, the first epoch is trained on the entire dataset. This is necessary since otherwise there is a large chance that the nodes diverge.

  After the first epoch, the worker only train the NN on the subset of the data it was assigned. After each batch, it computes the list of deltas to be send to the cluster, aggregates it in a single message and sends it to the communication service. Asynchronously, it also receives delta messages from the cluster at all times. At the end of each batch, the worker service processes all pending delta messages received in the concurrent queue, and apply them.

  At the end of each epoch, it creates a message containing the total training time, the time spent on compression and decompression, and the loss incurred during this epoch, and sends this message to the communication service.

The final part of the architecture is the client service. The client service is also built in Scala, and is simply a different type of actor in the Akka cluster. The roles of the client service are the following:

- Coordinating the initial start of the whole cluster by taking a launch argument `n` which represents the total number of nodes expected to work for the cluster. Once started, it will wait until it receives initial messages from `n` different nodes, before sending a start message to each one of them, containing a node index, which the communication service will then send to the worker service.

- Aggregating the runtime information and producing a final report. As explained above, the worker service will send runtime information after each epoch. This information is sent to the communication service, which will forward it to the client service. The client service aggregates this information and prints everything into a `.csv` file which is used to analyze the runtime, which parts of the program were slow (compression, decompression, training), and compare losses.

As for the machines we targeted, our initial plan was to be able to deploy our whole cluster in the cloud. To achieve this, we built a very portable cluster: the whole client and communication services are built into a .jar that can be executed on any machine containing a Java Virtual Runtime Environment (JRE), and the worker service can be executed inside a Python virtual environment with a small number of dependencies that can be automatically installed. This allows for a fast and easy installation on any hardware configuration.

3.2 Neural Network

As part of this project, we had to implement an entire neural network that was capable of doing speech recognition using PyTorch's framework. Our model was the following:

- A recurrent, fully connected recurrent neural network
- The input layer is comprised of 13 input points, each representing one of the extracted features from the audio at a specific dataframe.
- There are 4 fully connected hidden layers of 200 neurons each.
- Each hidden layer is followed by a $\tanh$ (hyperbolic tangent) non-linearity.
- The output layer is comprised of 30 output elements, each representing a specific phonetic class.
- The output layer is followed by a softmax layer which outputs the percentage of chance that a specific phonetic class is represented by the input.

To actually use this neural network, we had to preprocess large amounts of data. We used LibriSpeech' ASR corpus [6] to get a large amount of "clean" audio files, with the corresponding text in `.txt` files. The pre-processing worked as follows: go through each `.txt` file in the folder, get the associated `.flac` file, use a library for MFCC (Mel-frequency spectrum features) extraction, and create files with the input features and the corresponding output layer. Each character in the output text gets encoded into a number (from 1 to 30), since our neural network works with numbers and not characters. We thus also create an encoding/decoding map. We then randomize the whole files, and separate 20% for out test validation.

To compute the loss of an output compared to the expected label, we use the CTCLoss function. CTC stands for Connectionist Temporal Classification. CTCLoss is a famous loss function that is commonly used for speech recognition as well as text (image) recognition. Why do we need a special loss function? Well, let's say the input audio is someone pronouncing the word "Hello" out loud. Once this gets converted into dataframes and goes through the neural network, the output might be of the form "Hheeeellllooooo", since some letters are pronounced for a longer time than others. We have to somehow be able to compare "Hello" with "Hheeeellllooooo" and conclude that it is indeed the right word. That's where CTCLoss comes into play and helps us train our neural network the right way.

We use a decreasing learning rate. The learning rate starts at 0.008, and after every epoch, it gets divided by two. The reason behind this is that we need our neural network to take big steps at the start, in order to quickly improve. But the more epochs we train on, the smaller the learning rate should become, since we want to converge, and we want our training to become more precise, which is exactly achieved with this dynamical learning rate.

3.3 Problem mapping

This sub-question doesn't really apply to our project specifically because we parallelized at a higher level, namely through our multiple nodes on different machines. That being said, we ran into the specific issue of mapping our data structures to machine concepts when we were implementing the compression of messages in parallel. We will explain later in this report why we didn't manage to achieve our plan. The problem we had to solve is how we planned on parallelizing the compression of the hundreds of thousands of different weights. The whole neural network is defined by a list of so-called "tensors", which are the PyTorch objects that represent multi-dimensional matrices of parameters. For our neural network, we had about 20 different tensors. The smallest of them was the very first layer, containing only $13 * 200$ weights, while most of the other ones contained $200 * 200$ weights. Our initial idea was to parallelize this by mapping each tensor onto one specific thread. We implemented this version (the code is not used but is in the worker service, in the `/neural_network/model_trainer.py` file, in the `compress_gradients_parallel` method). We considered other options, namely: mapping a ranges of weights to threads, instead of tensors, in order to get a perfect work balance. This solution had other issues though as it required to correctly compute the tensor index for each given parameter index, which could have led to longer compute time per weight ($O(\log n)$) where n is the number of tensors) or increase memory usage ($O(m)$ where m is the number of weights in the NN). The first issue would have diminished the obtained speedup. The second is an issue because the NN we are talking about are enourmous and often barely fit into the RAM memory of the CPU/GPU that trains them. In fact, it was a major issue with the machine in the GHC and required us to use a smaller NN than the one we implemented initially. The reason why

we did not implement any of the later solutions is described after.

3.4 Iterations

We started this project from a blank sheet. We had to build our three different components, as well as the entire neural network, entirely from scratch, which was a very hard but interesting task.

Our project involved a very large amount of iterations and changes to our plans, due to different factors: time limitations, machine capacity limitations, financial limitations, to name just a few.

The first topic of discussion concerned the training data and how to get the data on each of the worker programs. Our first idea was to implement an entirely distributed system that worked as follows: the client service would be the only one with all the data. The worker programs would keep a queue of data to process, and whenever they would run low (a few elements left), they would send a request to its communication counterpart, who would in turn request new data from the client. The opposite trajectory would then happen to get the data back to the worker. This would lead to an optimal workload balance across workers as runtimes would be equal between all nodes in the cluster. We implemented this method entirely using Akka message passing, but we started running into two main issues that prevented us from pursuing it: when scaling to a larger number of nodes, this method turns the client service into a clear communication bottleneck, which was exactly what we were trying to avoid by using one-bit quantification and delta message compression. A second issue was that in order for the nodes to converge, we had to have the first epoch train on the entire data set, which would render this whole method useless as all data would have to be shared between all nodes anyway. Thus, we had to give up on this idea after a fair bit of implementation was already done.

Our second idea was to use a distributed data storage system like HDFS, but this quickly became impossible since we planned on using the GHC machines to do our testing. The solution we ended up implementing was more straightforward: each worker program contained a copy of the entire dataset. This way, it could train on the full dataset during the first epoch, and then simply take the chunk of data that it got assigned (by the client service), and train for the rest of the epochs on that chunk. This solution had no communication bottleneck, no implementation issues, but required more memory and preparation to work, which ended up being a slight problem later on.

Our second big topic of discussion concerned the communication between the worker program and the communication program. Both programs would be executed on the same machine, thus no network was required to communicate. The complicated part was that both programs were implemented in different languages (Scala vs Python), which made communicating more difficult. After extensive research of the potential solutions out there, our first idea was to use ZeroMQ.

ZeroMQ is a high-performance asynchronous messaging library that doesn't require a dedicated message broker. That last piece of information is important because it separated ZeroMQ from another potential alternative, RabbitMQ, which required a separate message broker. ZeroMQ has libraries for most major languages, and we started implementing a Scala and a Python version. Fast, it turned out to be a clear example of over engineering for our purposes. The documentation of the Scala library was very partial, and more importantly, the examples of asynchronous methods were completely outdated. Being able to asynchronously receive and send message was a central requirement for us, and we didn't manage to make this work with ZeroMQ.

We decided to implement our own asynchronous, TCP based message-passing service for communication between the two parts of our nodes. This ended up being quite a challenge, especially in Python, where asynchronous development is still in early stages and not as straight-forward as it is in Scala. We had to use concurrent, thread-safe queues from the `janus` package, both for sending and receiving messages, and create resilient connections that could restart themselves in case of failure. We also had to go through multiple iterations regarding the specification of the messages to send: first, we wanted to send fix-sized messages, as we had to allocate buffers to receive them, but it turned out that we needed to be able to send messages of varying size (it is not possible to know the size of the delta message in advance). We ended up structuring our messages in the following way: first send a message that contains the total size of the incoming message, and then send the actual message. We also had to adapt to the fact we wanted to send different type of messages: messages containing the deltas to be shared in the clusters, messages containing runtime information to be sent to the client service for data aggregation, and so on.

The other large topic of discussion in this porject was the actual implementation of the neural network and the (de)compression algorithm. Once again, our final implementation ended up being wildly different than our initial idea. Our first idea was to implement the whole neural network in C++. The reflection behind this was that if we used C++, we could take advantage of machine concepts much more easily, and more importantly, take advantage of CUDA to implement other computationally expensive parts of the project. Quickly, we understood that implementing neural networks in anything else than Python would be a challenge. Python is the industry standard in machine learning, and all important frameworks are built in Python. We therefore adopted PyTorch for the neural network part.

At this point, we still wanted to use CUDA for the compression and decompression of delta messages since we knew this would be a computationally expensive operation and found out that we could use PyCUDA to execute CUDA kernels from a Python program. PyCUDA had multiple limitations though: we couldn't install it locally for development since we didn't have CUDA on our machines, which made development hard. Second, once again, the documentation was limited. But the main issue that made us drop this idea was the potential issues we would have run into when trying to use PyCUDA on the GHC cluster where we do not have admin rights.

Nevertheless, we found the PyTorch C++/CUDA extension. PyTorch itself is built in C++, and the Python version is simply a wrapper around the C++ version. However, the C++ version is much more complicated than the Python wrapper and the documentation is way less extensive than the Python one. So it turned out that PyTorch itself makes it possible to write CUDA extensions that can be pre-compiled and then used straight from Python code. We started going down that path and writing the necessary kernels, when we ran into our final, and largest issue: memory.

The GHC clusters impose some very strict memory limitations on individual users. You can only use 2GB of data in your individual session. For us, this turned out to be quite a problem, since the installation of the necessary libraries in Python and all the other dependencies already amounted to close to 1.4GB. PyTorch itself takes up more than 1.2GB. This meant that we only had 600MB of available memory to store training data. But more importantly, the available RAM was also limited. And with the extremely large and deep neural network we had, it was actual impossible to run it: first of all, on the regular CPU, we ran out of RAM. We had to downscale the neural network (5 hidden layers of 2000 neurons became 4 layers of 200 neurons, the largest amount that fit in memory). More importantly, we were incapable of using CUDA with PyTorch for the regular neural network training. Even the downscaled neural network didn't fit in the the device memory. Scaling the NN down further would not have made any sens, as the cluster we were developing specifically targets deep neural networks.

We considered using mutli-threading to implement the compression but Python (CPython at least) is fundamentally single-threaded (the GlobalInterpreterLock or GIL effectively prevent multi-threading). Multi-processing can be done in Python but this was no use to us as we needed shared memory for the parallelized compression algorithm.

Faced with these limitations, we decided to entirely drop parallelizing the compression. This was very disappointing for us, but made sense in the larger picture, because the real parallelization we were working on consisted of the distributed nodes in a the cluster, and the parallel training there. If we used a CPU-only sequential implementation as a baseline, it wouldn't matter in our potential speedup. This ended up being our choice and we moved on from there.

Finally, a last choice concerned the architecture of our cluster: master-slave or purely peer-to-peer. We decided to settle on a middle ground, with a purely peer-to-peer training, but still using the client service as a single entry point to synchronize the beginning of the training, and the distribution of the data. This was the most sensible solution because the client doesn't become an actual central point of failure (once the training has started, it can run even if the client fails), but we still get the ease of being able to start without synchronizing all nodes in a peer-to-peer fashion.

4 Results

4.1 Measuring performance

In the context of a neural network, performances are defined by two main metrics: speed and test loss. To evaluate how well a neural network performs, the go-to method is to train your neural network on a large subset of the data (about 80%) and then evaluate your neural network on the remaining (usually 20%) of the data that the network will have never seen

before. The reason for this is that's it's very easy to overfit data. Overfitting means that the NN starts to over-adapt itself specifically to the dataset. It will then perform extremely well on the original dataset, but poorly on unseen data, i.e on a test set. To avoid this issue, we use the test set as the actual metric for performance.

Obviously, in our specific case, the metric that is most interesting is the training speed. We measure this as the total time it takes for the slowest node to complete the entire 10 epochs of training of its subset of the data, including evaluation on the test set. We compare this to the time it takes the sequential neural network to perform the training of the whole set of data. Obviously, a speedup in training only makes any sense if the test loss is at least as good as before. In the opposite case, our speedup isn't of much worth.

4.2 Experimental setup

The goal of our experimental measurements was to analyze the effect of several different training parameters on the overall speedup, median speedup (defined as the median of the training time of the different nodes), the test loss average over all nodes, the spread of the test loss on the different nodes, and so on.

To achieve this, we first tested our neural network in an entirely sequential setup. All experiments were executed on the GHC clusters, on machines 27 to 36. Our sequential test was simply to run the entire neural network training on one single node, with no compression/decompression, no message passing, and no cluster. The training time found in that experiment was used as the baseline for our future experiments.

We then decided to focus on three main parameters and study their effects. The three parameters were the number of nodes (later "node count"), the threshold value (τ) and the batch size. We fixed three "standard" values, respectively `nodes_count` $= 3$, $\tau = 2.0$ and `batch_size` $= 256$. We then executed the following set of experiments:

- Vary over the number of nodes: 2, 3, 4, 6 (with $\tau = 2.0$ and batch size 256)
- Vary over the value of τ: 0.5, 1.0, 2.0, 4.0, 6.0 (with 3 nodes and batch size 256)
- Vary over the batch size: 128, 256, 350 (with $\tau = 2.0$ and 3 nodes)

To aggregate the data of each experimental run, we fetched the logs printed by the worker service and the data automatically gathered by the cluster. Indeed, at the end of each epoch, each node sends a message to its communication service, with the following information: number of samples processed, total time spent training, total time spent compressing, total time spent decompressing, average loss per batch over the epoch, timestamp. The communication service then forwards this information to the client service, who writes this information to a .csv file. This enabled us to easily extract the data we needed, analyze it and plot relevant graphs.

We chose these three parameters for different reasons:

- The number of nodes is an obvious choice, since we want to be able to know how our system scales with a large number of nodes. Is the speedup linear? Does the communication become too big of an overhead?
- The value of the threshold τ is very interesting because it has potentially a lot of different impacts on the network. τ is the threshold that decides if a specific change in the weight/bias of a neuron has to be applied to the network and sent to the other nodes, or if it has to be stored in a residual list and wait for the next batch. The larger this value, the less changes will actually be applied at each batch on the neural network, and the less messages will be sent over the network. The smaller this value, and the more changes are applied, and the more messages are sent. The questions are: if the value becomes too large, do we still converge between the nodes? Since they will only receive very sporadic messages, they might not converge anymore. On the other hand, with a lower value of the threshold, the cluster will send more messages. Does that become a communication bottleneck?
- Finally, the batch size is interesting in our specific case because the time spent compressing the network is linear in the size of the neural network, independently of the size of the batches, but the training time of a batch is linear with the size of the batch. After each batch, we perform a compression operation, so changing the batch size might make the compression time a limitation to the obtained speedup. It's a good metric to prove how our speedup could be improved with more data, and the ability to train larger batches at a time, which would thus make our compression time overhead negligible.

4.3 Graphs

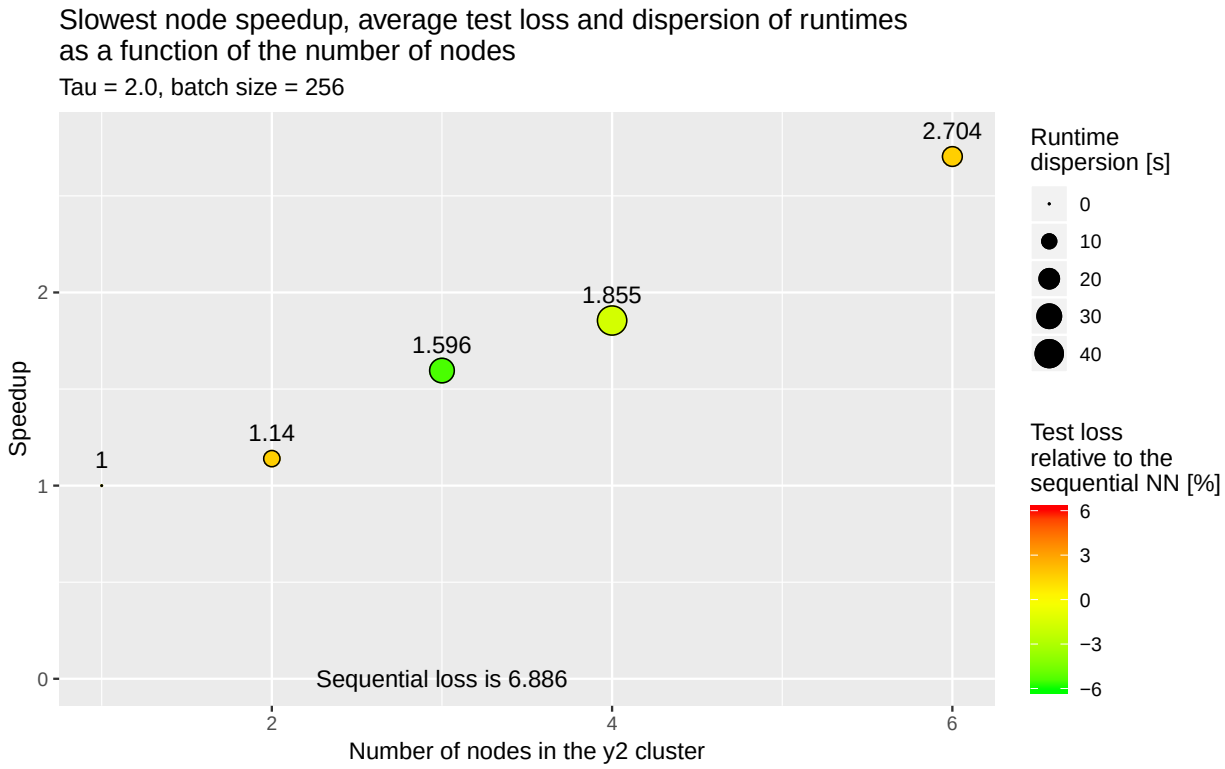

Figure 1: Slowest node speedup as a function of the node count

Figure 1 shows the main results of our project. We compare the runtime of the slowest node in a n-node cluster, the average test loss among all nodes, and the dispersion of the runtimes of all nodes, to our sequential baseline. We decided to use the slowest node since this is a more precise metric than taking the average or the fastest, even though theoretically since we observe a great convergence of all nodes, we could use the fastest node. The size of the point indicates how large the runtime dispersion was. This was computed as the standard deviation of the runtimes of all nodes in the cluster. The color of the point indicates the relative difference of test loss in % to our sequential implementation. A greener color indicates an improvement, red colors indicate a worst test loss.

We can see that as we scale the number of nodes, we obtain a relative speedup in runtime which is slightly sublinear. Though not perfect, we still obtain a very interesting speedup. For a 6-node cluster, we obtain a 2.704x speedup, which means that on a large scale deep neural network training of 100 hours, we would gain nearly 65 hours. We explain in detail in the following sections why we aren't closer to a linear speedup.

More importantly, this graph shows that the test loss is extremely close, and sometimes *even better*, than the test loss of our baseline sequential run. This is a crucial result for us as it shows that we still train our neural network as precisely as in the sequential version, and that we can thus say that we actually achieve a real speedup with no precision loss. It's important to realize that the differences in test loss as seen in the above graph are very small and conclusively show that the nodes converge to the same result as the sequential implementation, which is the most important result.

We can also see that with more nodes, we start obtaining runtimes that variate more amongst the different nodes, which shows that there are potential optimizations to be done in order to have a more homogeneous execution time across nodes which would in turn increase the overall execution speed. This could be achieved by having dynamic load balancing of the data and synchronization after epochs.

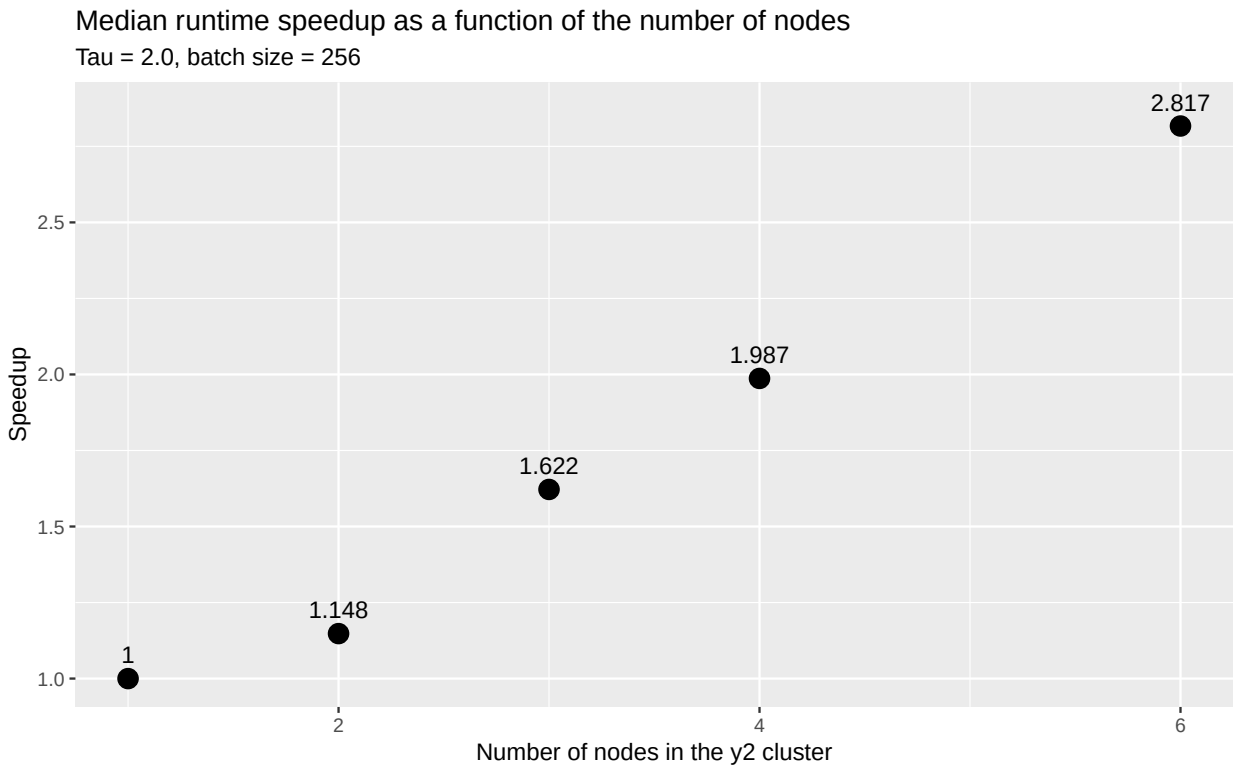

Figure 2: Median runtime speedup as a function of the node count

Figure 2 shows the median speedup of a cluster of n-nodes, ranging from 2 to 6 nodes. Our goal with this graph is to show that by taking the median instead of the slowest node in the cluster, we achieve a slighter better speedup relative to the sequential implementation. Multiple reasons can cause one node to run slower than others. The performance of the machines is the simplest explanation, as no two machines have the exact same performances, especially in a cloud infrastructure where machines might be shared. Another reason can be data imbalance: batches of training data are comprised of the same number of input audio files, but a specific file can be longer than another one. Even though we use randomizing techniques in the batch creation to avoid imbalance issues, one node could still statistically be "unlucky" and end up with heavier training data.

A technique to mitigate this would be to synchronize all nodes at the end of each epoch, and have the client re-balance the data in order to have homogeneous computation times. This is a hypothesis, and it might have other consequences, such as a worse convergence rate. The main idea remains that in a cluster of multiple nodes, we should be aiming for the most evenly distributed runtimes among all nodes, which in turn results in better performance on the test set.

Our hypothesis is that if we can achieve less dispersion in the runtimes, we would end up with a faster and more precise overall system. This also because the learning rate is hlafed after each epoch, thus having nodes work on the same epoch at the same times guarantees that they will "learn" at the same rate and prevent having a node in its last epoch, thus learning very slowly to increase the precision of the learning, be "crushed" by messages containing a lot of changes from other instances still a few epoch behind. We haven't been able to follow through on this hypothesis due to a lack of time.

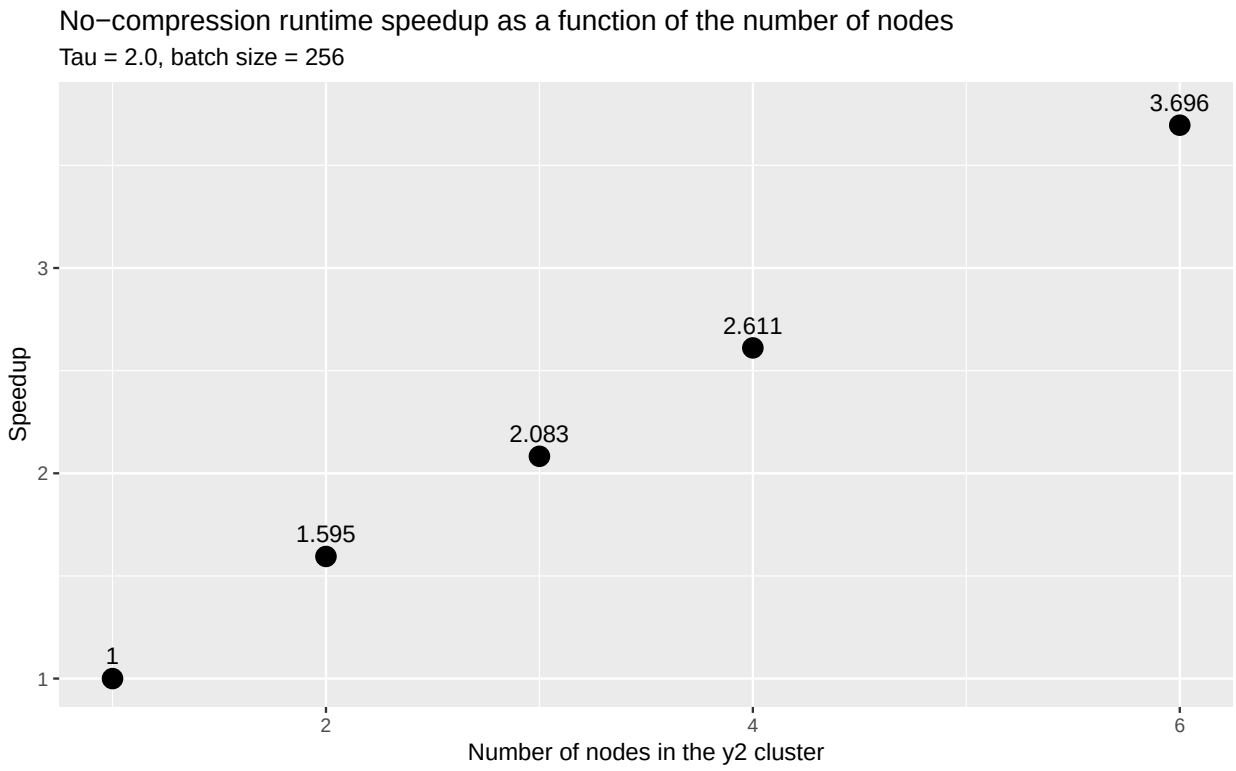

Figure 3: No compression slowest node speedup as a function of the node count

Figure 3 shows an important result for our project: we can achieve a much higher speedup than our current results by optimizing the computation of the delta messages, i.e the compression methods of our neural network. We see that in our 6-node cluster, we would improve from a 2.7x speedup to a 3.696x speedup, a huge jump. This is obviously an idealistic scenario: we will never be able to entirely get rid of the compression time. That being said, the compression time is a product of multiple side-effects which we can't change. First of all, we weren't able to do the compression on a CUDA device, which would've reduced the time by multiple factors. Second, due to memory limitations, we weren't able to train the network using larger batch sizes (minimum 1024, if not 2048). The larger the batch, the smaller the overhead incurred by the compression. We will go into more detail in the following sections about why we ran into these issues. With this extra background information, computing the speedup without the compression time makes more sense, and shows us that we can get close to a linear speedup in the number of nodes, which was exactly the result we were striving towards in this project.

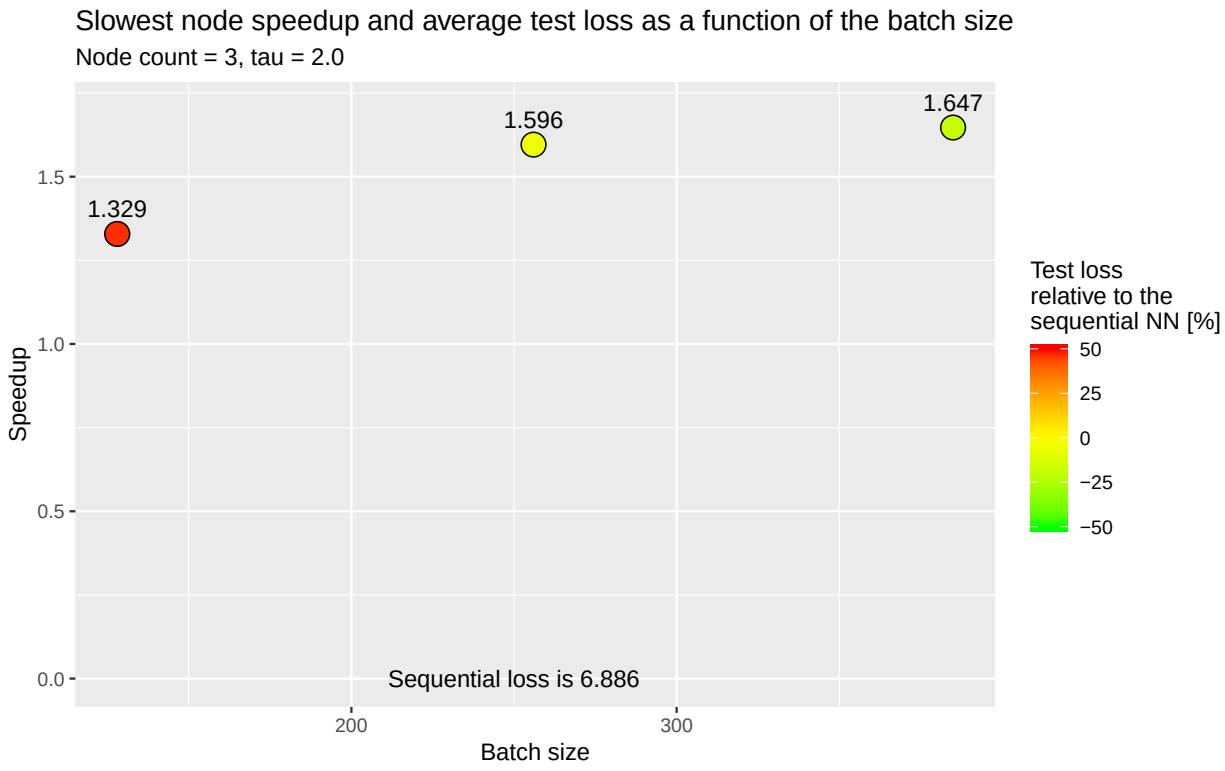

Figure 4: Slowest node speedup as a function of the batch size

Figure 4 shows the speedup of the slowest node in a 3-node cluster as a function of the batch size. We used three different batch sizes: 128, 256 and 385. The reason we had to stop at 385 was that using larger batch sizes resulted in RAM issues on the GHC cluster machines. Sadly, this prevented us from proving a larger point, but the graph still shows us some interesting information.

First of all, we can see that with a larger batch size, we obtain a speedup compared to smaller batch sizes. This makes sense because as explained previously, the compression time is fixed (depending on the neural network architecture), and doesn't vary based on the batch size, whereas the training time does obviously vary based on the batch size. The latter is true because PyTorch is capable of computing batch outputs in a very efficient manner, and computing two batches of size 128 will thus take more time than computing one batch of size 256. There is obviously a limit on the size of the batch that is realistic, but to give an idea, [8] used a batch size of 1024. This would still represent a 4x increase in our standard 256 batch size, thus showing a clear speedup possibility.

The second interesting information that can be seen on this graph is that the test loss improves considerably with larger batch sizes. We aren't entirely sure why this is the case, and can thus only make hypothesis, but one idea is that larger batch sizes prevent a neural network from overfitting, since it won't model itself to fit very specific points, but has to try to fit a larger amount of points, which makes it more generalizable. This is a mere hypothesis and it would be very interesting to analyze the underlying cause of this improvement in test loss with even larger batch size. Is there a batch size above which the convergence starts to sink (messages sent too rarely causing convergence issues for example)?

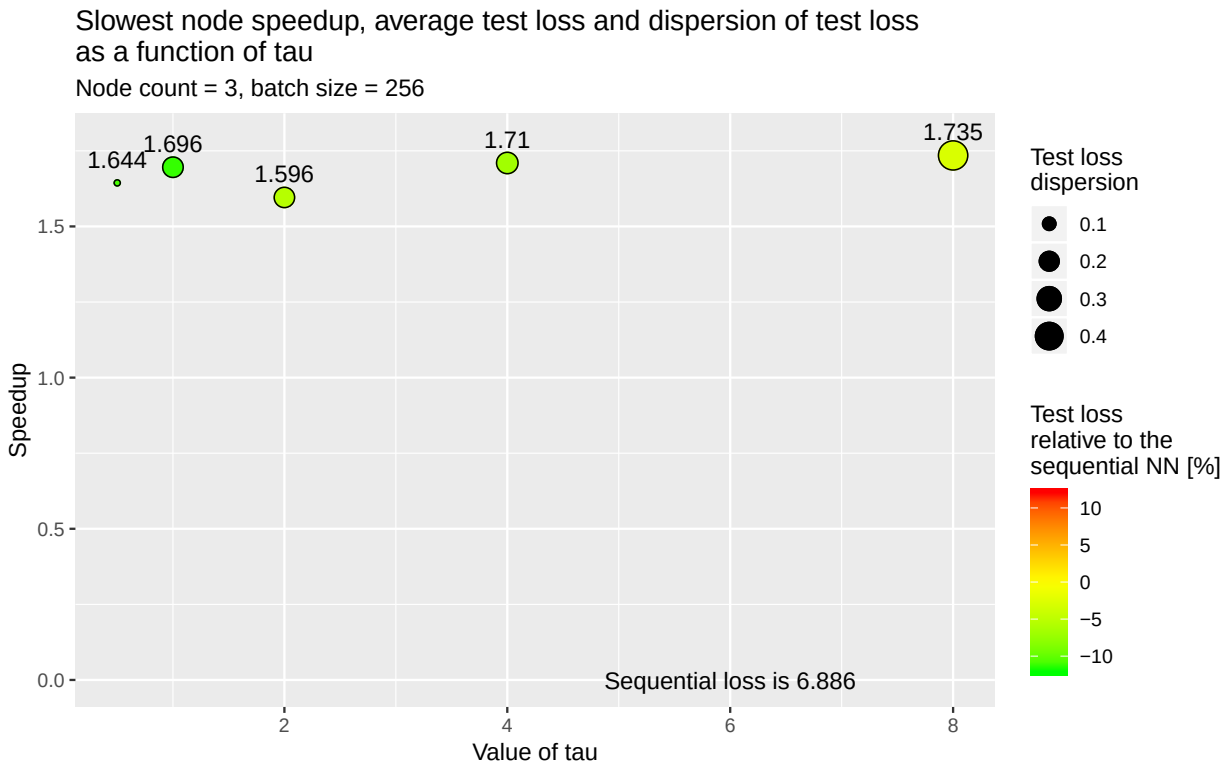

Figure 5: Slowest node speedup as a function of τ

Figure 5 shows the slowest node speedup, the average test loss and the dispersion of test loss in a 3-node cluster, with a fixed batch size of 256, as a function of the threshold value τ (varying: 0.5, 1.0, 2.0, 4.0, 8.0).

For some context, the value of τ is important in our project since it determines the value at which weights/biases of neurons in the neural network will directly be applied to the neural network and sent to the rest of the cluster, or if they will be stored in a residuals list, waiting for the next batch. This has impacts on the size of messages sent in the cluster, and on the speed at which changes are applied to the neural network itself.

Our first observation from this graph is that the speedup remains similar for all values of τ. We can conclude that the size of messages doesn't negatively impact the runtime of our project, at least not at such small scale. This is a great result as it means we are far away from reaching a communication bottleneck in our cluster, which is one of the potential fears with our system. We haven't been able to try out the system in more extreme situations (> 80 nodes, > 50GB of data), but [8] suggests that even at those levels, there still is no communication bottleneck.

The second observation is that for all values of τ, we have a better test loss than our sequential implementation. This is closely linked to our third observation: the larger the value of τ, the larger the test loss dispersion. These two observations give us some very interesting insight. The fact that the average test loss remains better than the sequential test loss shows us that at least one of the nodes achieves very good performance. But at the same time, the dispersion becomes larger, which means that the convergence isn't guaranteed for all nodes anymore. Since the learning rate changes after each epoch, and since less messages are sent, it might mean that the timing of the sent messages (relative to the epoch each node is in) becomes more important. Generally, it seems to indicate that a value of τ between 2.0 and 4.0 seems optimal, as it achieves low dispersion, and high accuracy while keeping the number of sent messages lower than smaller values of τ.

4.4 Problem size

As explained earlier in the report, the problem size played a huge role in our results. As we can see in the graphs, the speedup becomes larger with problem size (amount of training data). This makes sense, because the compression of the neural network into a message to send to the rest of the cluster takes a time linear in the neural network size. This computation is done once every batch. The training time per batch depends on the size of the batch: the larger the batch, the larger the training time. This means that the larger the batch, the smaller the influence of the compression time on the total time per batch, which in turn means that we get a larger speedup. Finally, the more amount of data we manage to get, the larger the batches we can make. Sadly, we ended up being entirely limited by the amount of free memory on the GHC machines, which limited us to 600MB of data. If we had more memory, we could have tried training up to 20GB of data, and we would have had much more interesting speedups.

4.5 Analysis

We can see that we basically achieve a 2.5x speedup on the total training time using 6 nodes compared to our baseline results.

Table 1 is the breakdown of the execution time of the slowest of the 6 nodes in a 6 node execution, with 256 batch size and $\tau = 2.0$. The speedup compared to the baseline version is 2.7x.

Task	**Time [s]**	**Percent [%]**
Ep. 1 Train	133.216	26.000
Ep. 2-10 Train	233.321	45.500
Ep. 2-10 Compress	137.508	26.800
Ep. 2-10 Decompress	0.007	0.001
Remaining time	8.206	1.600
Total time	512.252	100.000

Table 1: Breakdown of the NN training time in a 6 nodes configuration

Table 1 gives us multiple reasons that make for a non-linear speedup in the number of nodes:

- The biggest reason is the compression computation after every single batch. We were unable to parallelize this in Python due to the limitations of multi-threading in Python, and we couldn't use CUDA either. The smaller the batch size, the larger the overhead on compression. To illustrate this, we plotted the graph of the speedup if we could bring down compression to a negligible time, which seems very achievable since the operation is extremely parallelizable. We can see that this operation makes up for 26.8% of the total training time. This could clearly be reduced to less than 5% of the total training time, which would give us more than a much bigger speedup (we could reach speedups of 3.7×, as plotted in figure 3). We actually computed the speedup if we ignored the compression (which isn't entirely realistic, obviously, but still gives us a good approximation), and we found that for this particular instance, we would have had a 3.69x speedup, a clear improvement on our original 2.7x speedup.
- As explained above, Python has an inherent lack of multi-threading due to the GLI (GlobalInterpreterLock). Operations like receiving messages and sending messages during batches thus interrupt the actual neural network computation, albeit for a very small time. We can't compute it exactly, but there is definitely some overhead. This can be classified as communication overhead. In table 1, we see that about 1.6% of the overall runtime is "unaccounted for" in our major component breakdown. This is a good approximation for the communication overhead.
- The first epoch has to be trained on the entire data set on each node. This means that for 10% of the time (1/10th of the epochs), we don't actually parallelize any computation. As we can see in table 1, the first epoch training accounts for 26% of the total time. This is something that is required by the algorithm to ensure convergence, even though we could potentially try to reduce the time (not train a full epoch, but a smaller fraction) and still try to achieve convergence, to reduce this overhead. It still explains why we're not closer to a 6x speedup.

There is no synchronization overhead since we don't synchronize any of the computation on the different nodes – it's entirely asynchronous.

4.6 Choice of machine

Sadly, our initial choice of having part of the work on the CPU (communication) and part of the work on the GPU (worker) wasn't possible, because of the limitations imposed by the GHC clusters. Due to that, we had to switch to a full CPU version. We still managed to demonstrate very good speedup numbers compared to a baseline CPU sequential implementation, which was the main goal of our project.

That being said, using a GPU would still be much more interesting to do all of the computationally heavy work (compressing and neural network training), and would allow us to achieve truly interesting speedup numbers.

5 Future Improvements

If given more time and resources, there are multiple interesting improvements we could work on to achieve an even better speedup and more scalable program.

- The obvious first improvement is to entirely parallelize the compression algorithm using CUDA in order to reduce the massive overhead we incur after every single batch. This would bring us much closer to an actual linear speedup.
- Applying our cluster to much bigger amount of training data and larger batch size would probably increase the training speedup a lot.
- We would obviously want to put the whole neural network computation on CUDA too, in order to be able to compete with state of the art implementations.
- Finally, using more nodes, we would want to test whether the divergence in the total training time per node increases. To avoid having one node finishing much earlier than others, we could dynamically re-balance the data to be computed after an epoch, by moving some of the "heavier" data to a node that finished earlier. This would force us to implement some kind of synchronization in between epochs. This would also ensure that each node receives all of the delta messages from all other nodes before finishing its training (currently, if one node finishes is 10th epoch will another node is still at epoch 6, it will miss 4 epochs of improvement messages to its neural network, thus potentially resulting in a higher test loss).

6 Division of work

Equal work was performed by both project members.

7 Conclusion

In this report, we explained how we implemented an asynchronous distributed deep neural network training framework, using message passing at scale. The speedups demonstrated in this report show great promise for the algorithm described in [8]. We also detailed the limitations we had to face on the GHC clusters, which could be alleviated if we had more available memory to store larger amounts of data, and access to a GPU with more available RAM to store the extremely large amount of neural network parameters. Achieved speedups could be much higher when training *at scale*. Nonetheless, we fulfilled our initial goals of implementing a functional asynchronous distributed deep neural network training framework set out in our project proposal and managed to achieve promising speedups.

References

[1] *Akka Documentation*. URL: https://doc.akka.io/docs/akka/2.5.4/scala/ (visited on 05/07/2019).

[2] *Akka: Build Concurrent, Distributed, and Resilient Message-Driven Applications for Java and Scala | Akka*. URL: https://akka.io/ (visited on 04/02/2019).

[3] *Asyncio — Asynchronous I/O — Python 3.7.3 Documentation*. URL: https://docs.python.org/3/library/asyncio.html (visited on 05/07/2019).

[4] *Deeplearning4j*. URL: https://deeplearning4j.org/ (visited on 04/09/2019).

[5] James Lyons. *This Library Provides Common Speech Features for ASR Including MFCCs and Filterbank Energies.: Jameslyons/Python_speech_features*. May 6, 2019. URL: https://github.com/jameslyons/python_speech_features (visited on 05/07/2019).

[6] *Openslr.Org*. URL: http://www.openslr.org/12/ (visited on 05/06/2019).

[7] *PyTorch Documentation — PyTorch Master Documentation*. URL: https://pytorch.org/docs/stable/index.html (visited on 05/07/2019).

[8] Nikko Strom. "Scalable Distributed DNN Training Using Commodity GPU Cloud Computing". In: (), p. 5. URL: http://www.nikkostrom.com/publications/interspeech2015/strom_interspeech2015.pdf.

[9] *The Scala Programming Language*. URL: https://www.scala-lang.org/ (visited on 04/02/2019).

[10] zenecture. *NeuroFlow*. Mar. 27, 2019. URL: https://github.com/zenecture/neuroflow (visited on 04/09/2019).