Assignment 2
Group 11
CS744-Fall-2020

Names: Shing Ng, Jiun-Ting Chen, Matt Blakely

Introduction:

Our goal is to learn how to use the PyTorch framework, understand the basic mechanisms needed to train a neural network, and common techniques for distributing model training. We will Investigate the claims of the PyTorch and Pipedream papers that data parallelism incurs a high communication cost and if distributing the training of a model speeds up the training of the model. The code structure section describes the general model training methodology which is similar in the different tasks. The task 1 section investigates a single node instance of model training. The task 2 and 3 section investigates the differences between the different distributed communication techniques and compares the system utilization of all three tasks. We conclude with a brief section describing our main takeaways from this assignment.

Setup:

One physical machine on cloudlab split into 4 VMs. Each VM has 5 Cores (1 thread per core) running at 2.2 Ghz and 15GB of RAM. We tested with PyTorch version 1.4.

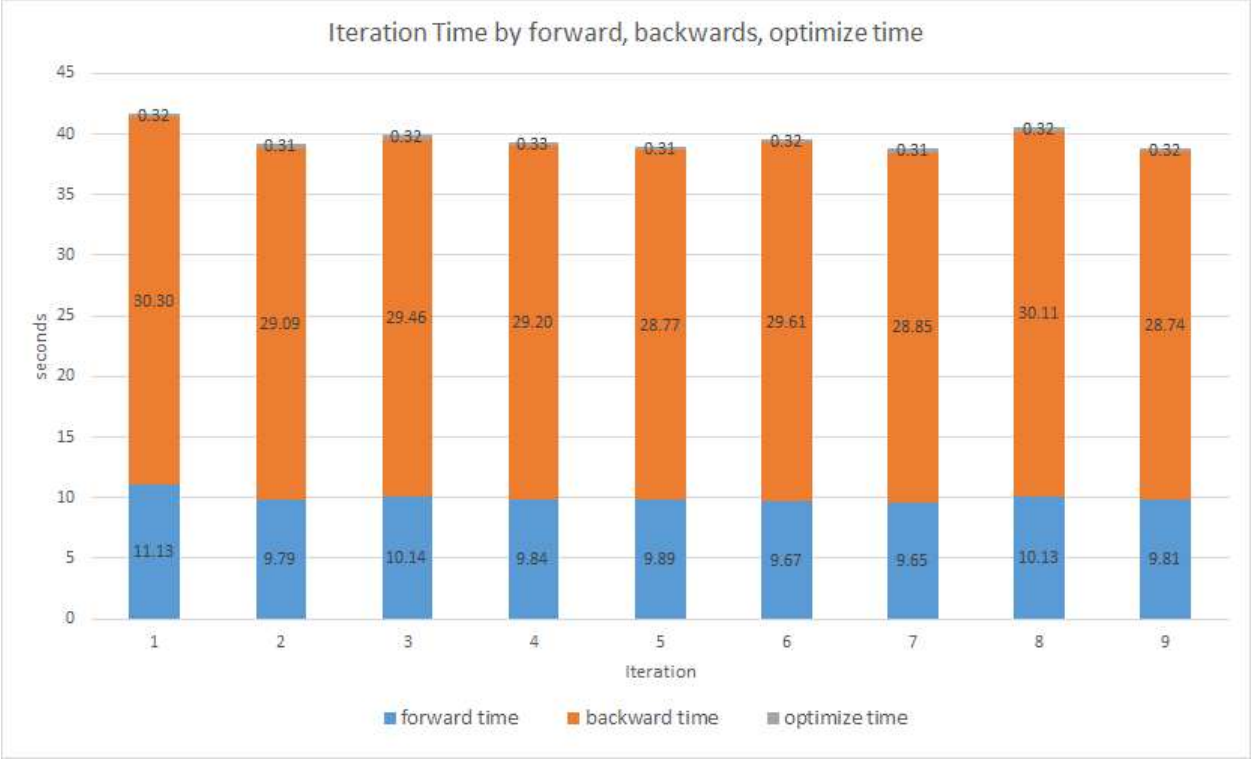
Code structure:

The code for task 1 functions as the baseline for the subsequent tasks. We were given a largely complete implementation of a VCG-11 network to which we added the training steps of forward pass, backward pass and optimize. The forward pass (for a VCG-11 network) accepts a batch of labeled images and determines a classification for that image based on the current model parameters. We then use a cross-entropy loss function to determine how incorrect the classification is, generally referred to as the loss value. The backward pass uses stochastic gradient descent to determine how each parameter affected the classification. This step is referred to as the “backwards step” because using the derivative chain rule, it calculates the gradient for each parameter starting with the last parameter in the neural network. This step is the most computation intensive step and is where we will focus the discussion for the rest of this report. Finally the optimize step uses the derivative to update the parameters to minimize the loss they contributed to the final classification. For all of the tasks, we define an iteration to be 20 forward-backward-optimize passes, this is roughly ~10% of the total data (there are 50,000 images, each iteration is 20*256 images). In the data parallelized runs each node operates on the batch size divided by the world size number of images per batch. After each training pass,

the parameters are synced so even though each node operates on only 1/world_size of the batch, after the syncing each node incorporates the updates from the entire group.

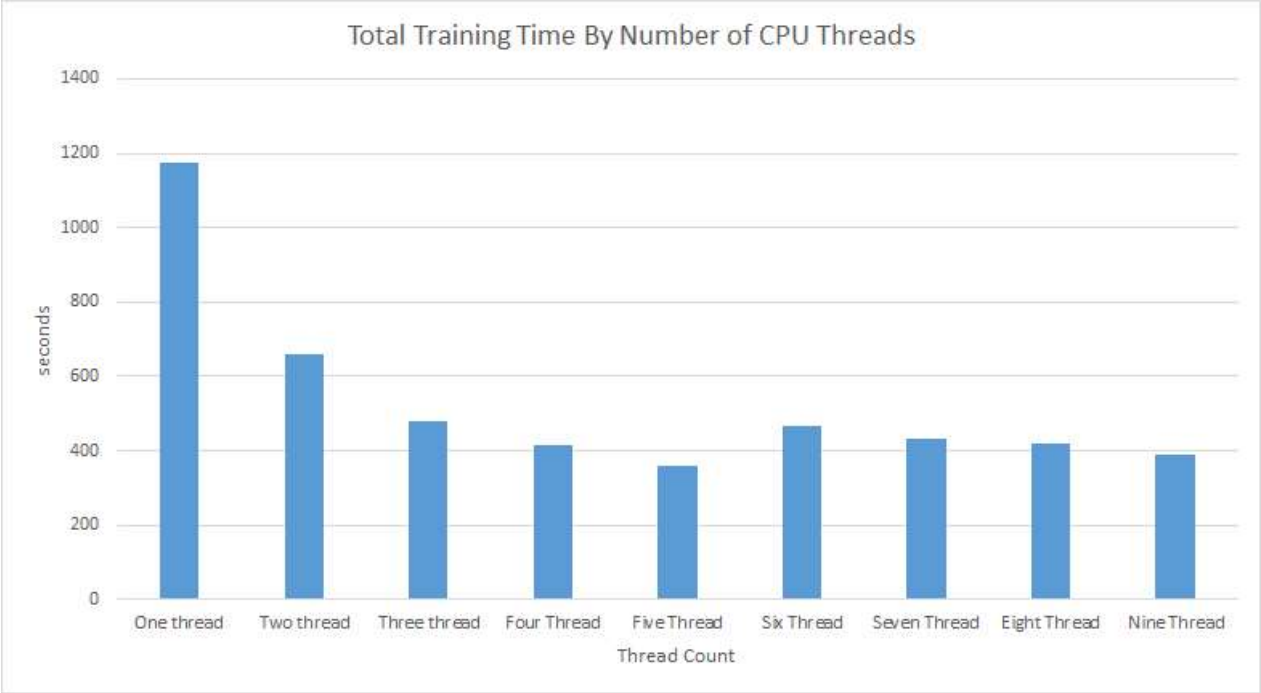
Task 1 - single node:

For task 1 we investigate whether the Pytorch and Pipedream paper's claim that the backward pass is the most resource intensive stage of training, and if the workload is appropriate to parallelize. Figure 1 shows the training pass per iteration of one epoch broken out by the training stages. We see that indeed, the backward pass consumes the majority of resources during the training pass. There is also little length variance across iterations indicating the work is well distributed and likely suitable for parallelization. The training required only 8 GB of memory, about 50% of the available memory, indicating that the CPU is the limiting resource during training.



[Figure 1 - bar graph of three runs average of task 1 broken out by iteration]

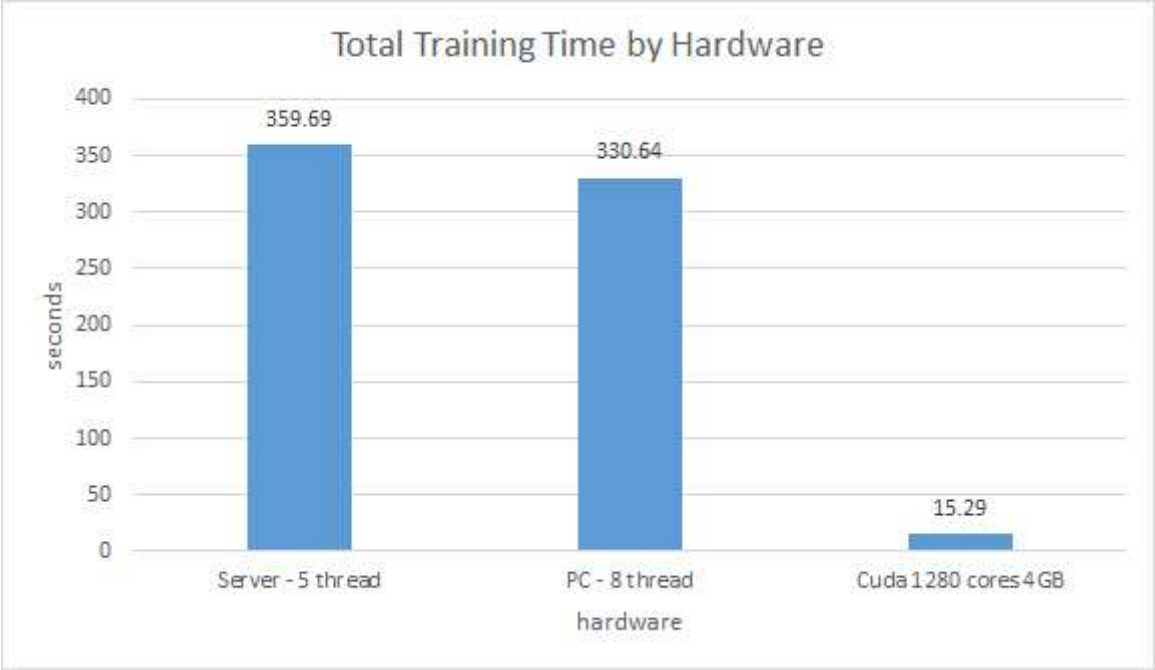
Figure 2 shows the total training time ranging from 1 thread to 9 threads. The server has 5 cores and doesn't seem to support hyperthreading so the performance improvements of additional threads tapers off at 5 threads due to the processors hitting 100% CPU utilization. Since we were able to hit 100% utilization for the CPU, likely this workload could benefit from additional parallelism which a distributed environment could provide.



[Figure 2 - total training time for a single node using different numbers of threads]

We ran task 1 on a Windows 10 machine with 4 Core (2 threads per core) 3.6 Ghz, 16 GB RAM and a 4GB Memory, 2Ghz Nvidia GTX 1650 Super on the CPU and on the GPU. While running on the GPU it utilized 100% of the cores and ~60% of the available memory. While running on the CPU it utilized 60% of the CPU and 70% of the RAM. Figure 3 shows a comparison of the total run times of the cloudlab server (5 threads), pc (8 threads), and GPU (1280 cores). The GPU finishes 20 times faster than the CPU training runs with a run time of 15 seconds and with expected accuracy for one epoch. We will not use the GPU training time in further comparisons but it is an interesting reference point for further parallelism.

From this task we conclude that the both the PyTorch and Pipedream papers are correct that the backwards pass requires the most resources and deserves the most focus when trying to improve run times. Additionally, the PyTorch framework does an excellent job of utilizing native parallelism on a single node with the scaling of threads and cores on a GPU. Since it is able to scale well on a single node, we would expect to see improvements in a distributed environment as well.



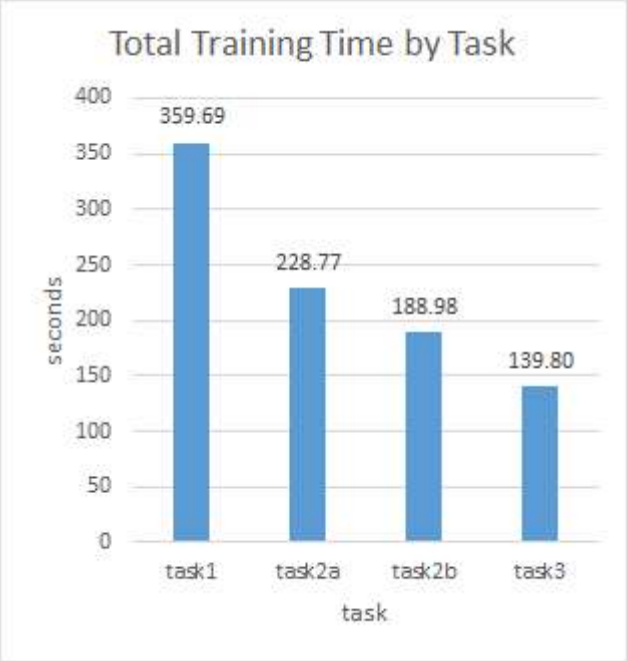
[Figure 3 - bar graph of CPU vs GPU total training time]

Task 2 and 3 - Distributed:

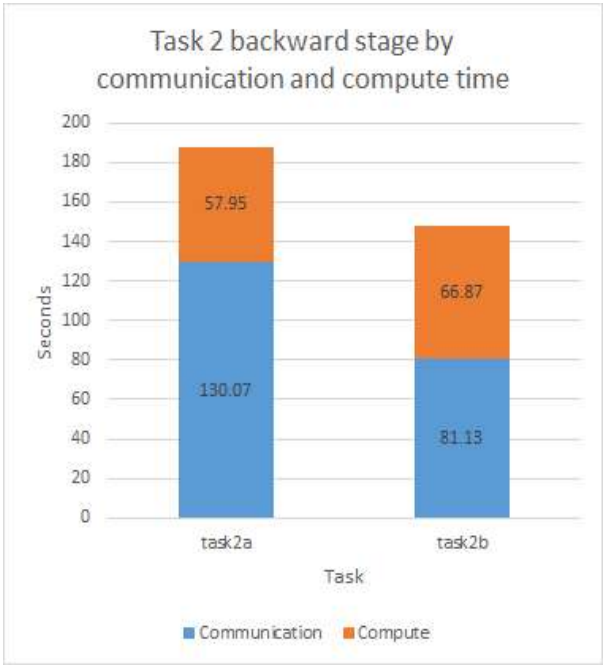
In task two and three we incorporate distributed aspects into the model training. All three tasks introduce a new step to distribute the data between the different nodes. For this implementation, each node receives the full dataset, but we use a sampler function to partition the dataset into one partition per node. Each model then calculates the gradient for its batch and syncs its gradient with the whole system. The tasks in this section take different approaches to the gradient synchronization: scatter and gather, allreduce, and an asynchronous communication version of allreduce. The main reason we expect to see a performance increase is that each node is able to process a fraction of the total data (1/world_size per batch) but due to the gradient synchronization each node is able to incorporate the entire batch world of training. The gradient synchronization is critical for ensuring the accuracy of the model but the communication inherently slows down the performance. We will investigate how each communication strategy affects the total run time length, amount of time spent communicating, and overlap between computation and communication.

Training Time: Figure 4 shows the total run time between all four tasks. Ignoring the GPU run, all three distributed tasks run more quickly than the single node task. In task 2, the gradient computation and communication happen in serialized steps so we are able to calculate exactly how much time each task spends on communication. Task 3 uses asynchronous communication to combine the gradient information so we cannot use the same timing methods. Figure 5 breaks down the backward pass of task 2a and task 2b by computation time and communication time. Task 2a uses a naive scatter and gather approach: each node sends the gradient tensor data to the rank 0 node, then the rank 0 node sends the reduced data back to each node.

Task2b uses the PyTorch built in AllReduce communication which implements a ring communication algorithm reducing the amount of information sent over the network. The ring approach in task 2b minimizes the communication and thus increases performance.

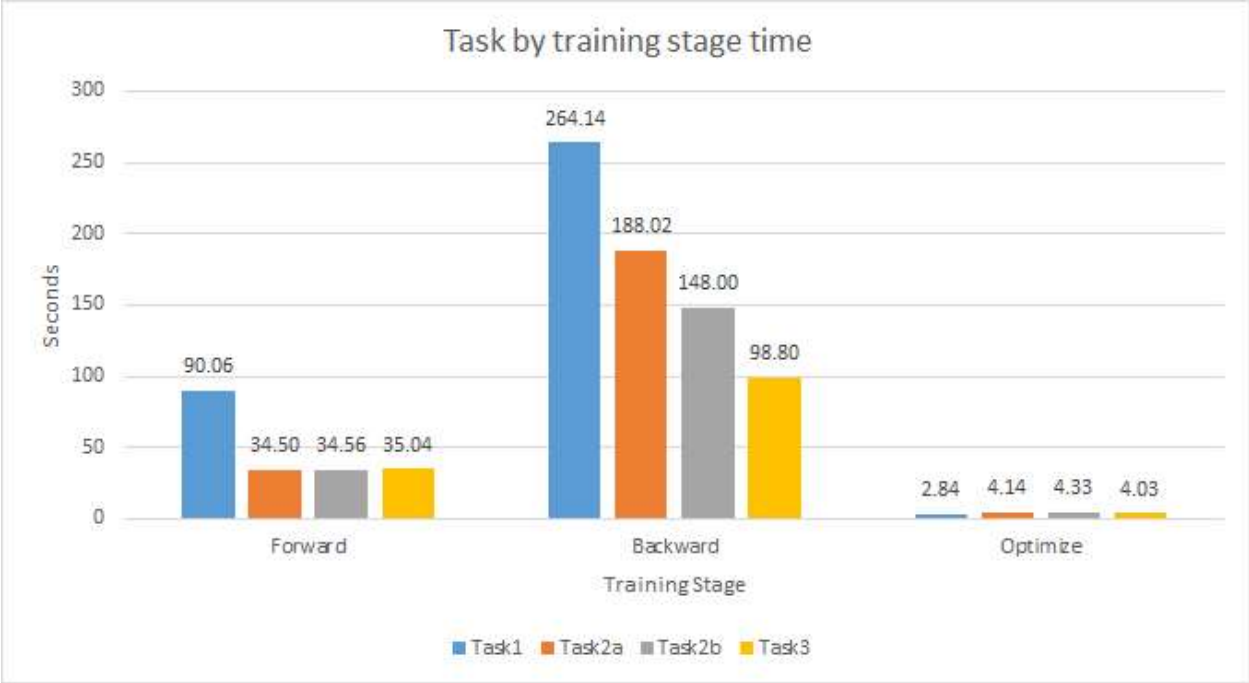


[Figure 4 - total training time by task]



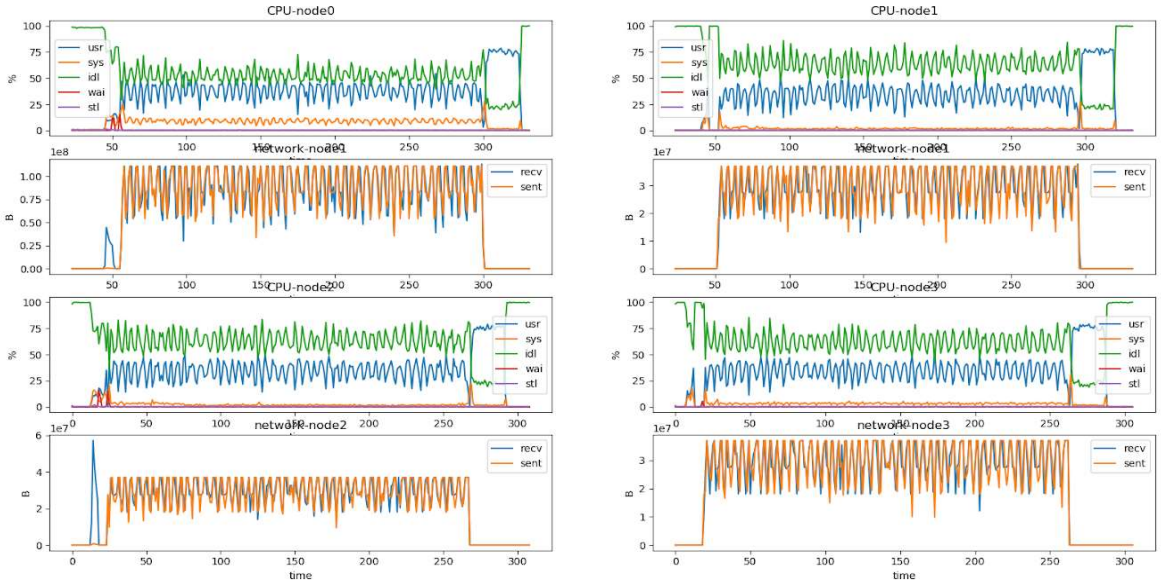
[Figure 5 - backward stage for task 2a and 2b broken out by communication and compute times]

Figure 6 summarizes the main differences between the different training strategies. The forward step is 3 times in the distributed environment because each node processes 1/world_size portion of the dataset without need to synchronize. While one might expect to see similar gains in the backward step, we now incur a communication cost to synchronize the gradients which based on the communication strategy decreases the gains we see from chunking the dataset. The higher task numbers utilize more complicated algorithms (ring and then asynchronization via buckets) to minimize the communication time during the gradient synchronization resulting in faster total run times.

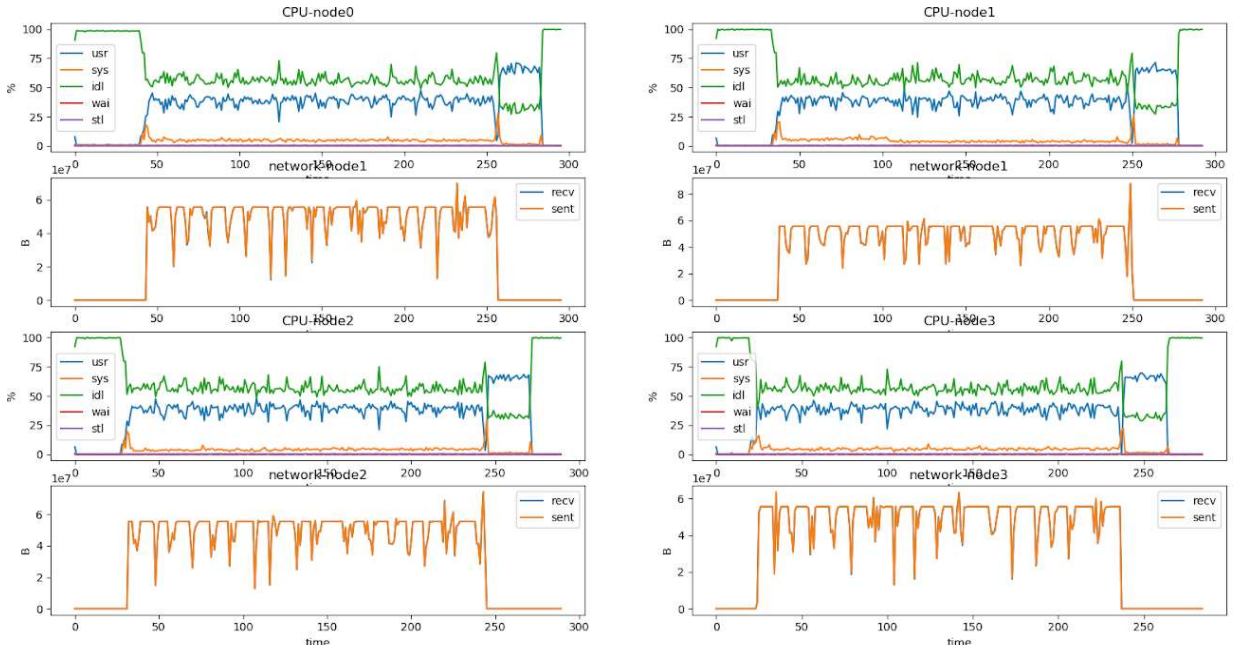


[Figure 6 - The tasks split out by their training stages]

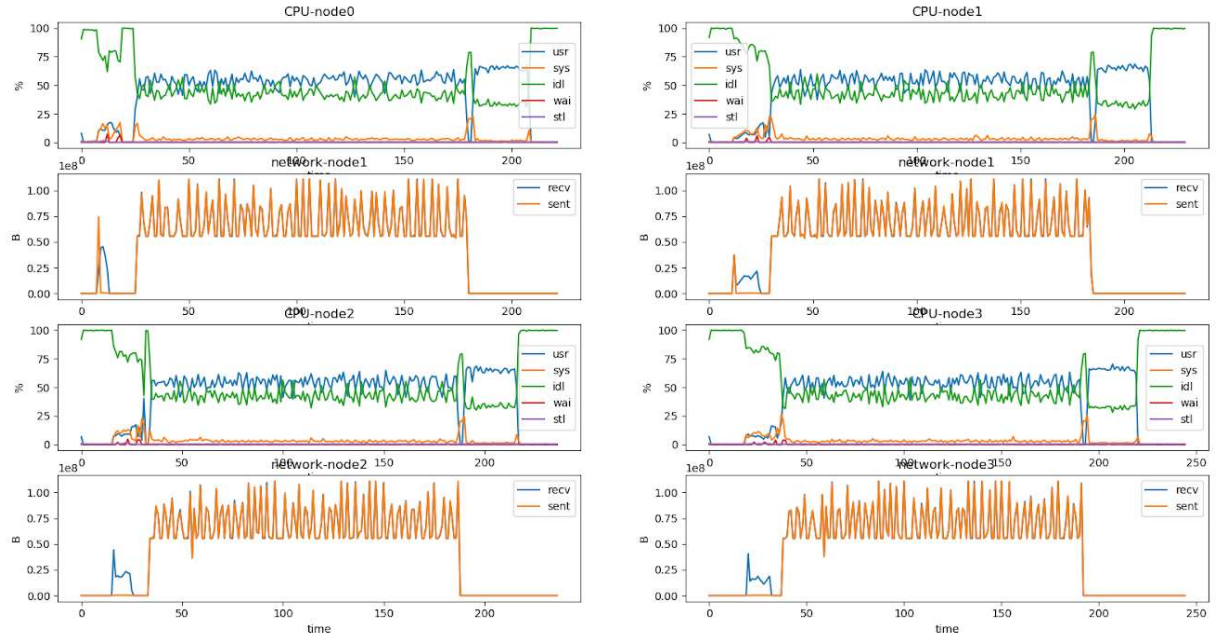
Workloads Distribution: Figure 7, 8, 9 shows CPU usage and network transmission information of 4 nodes during training, which was collected by dstat**.** The usage of resources are quite evenly in all cases**.**



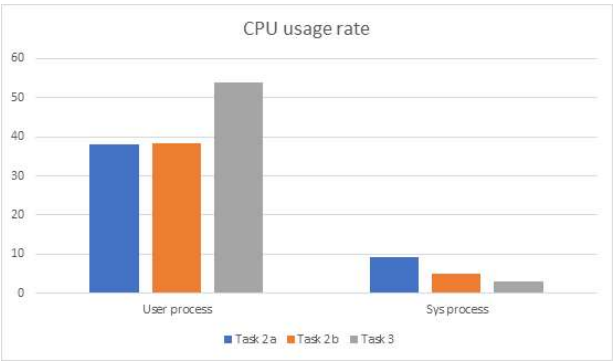
[Figure 7 - CPU usage and network transmission: Task 2a]



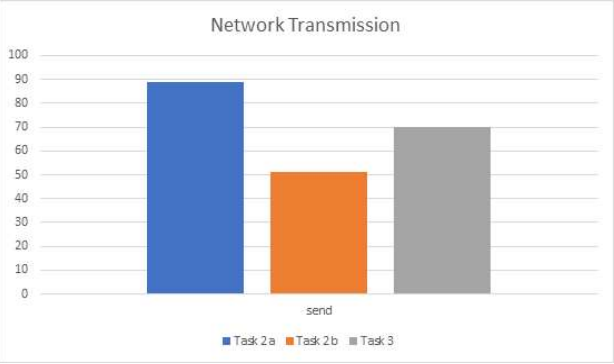
[Figure 8 - CPU usage and network transmission: Task 2b]



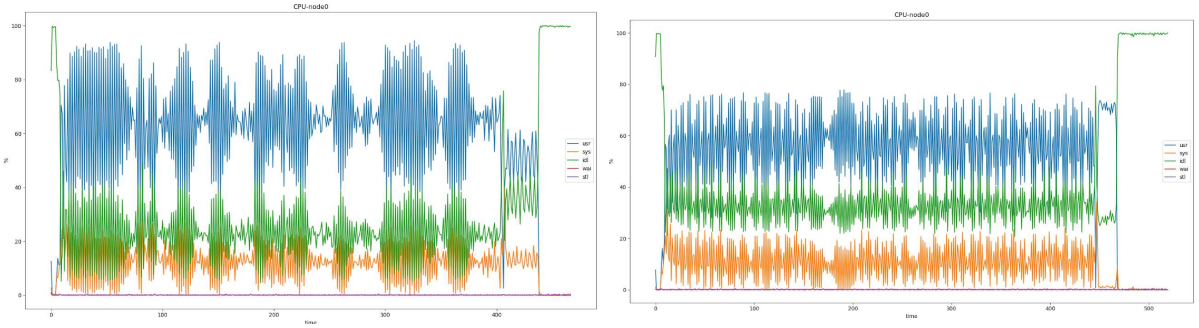
[Figure 9 -CPU usage and network transmission: Task 3]



[Figure 10 - Mean of CPU usage]



[Figure 11 - Mean of send per second]



[Figure 12 - CPU usage of training with single node -Task 2b vs Task3]

Communication: Our experiments support the efficiency of ring allreduce. As Figure 7-9 shows, the amount of data transmission is more stable in Task 2b. In addition, Figure 11 shows that the data transmission in Task 2b is also much less than other tasks, which is less than 60% of Task 2a and 80% of Task 3.

CPU Usage: The CPU usage of Task 3 is higher than Task 2 and the task time of task3 is shorter, which indicates that the default communication method of Pytorch has a more efficient resource utilization. Moreover, Table 2 shows that the usage of the user process does not increase significantly, which makes sense due to some limitations on each thread. On the other hand, we found the usage of the system process became higher as the number of nodes decreased. Figure 12 also shows that it kept around 10% during the whole process, which is different from the 4 nodes cases.

Scalability: Table1, 2, 3 compares the performance of task 3 with different numbers of nodes. Based on the training time and the network transmission, it seems that the gain from 3 to 4

nodes is not significant. This result might be due to the fact that VGG is not a large model, which indicated that it might not be a good choice for adding workers with additional communication cost when the total workload is not heavy.

	node0	node1	node2	node3
1 workers	437	-	-	-
2 workers	228	228	-	-
3 workers	169	169	170	-
4 workers	153	153	153	153

[Table 1 - Training time with different number of nodes (s/epoch) - Task 3]

	node0	node1	node2	node3
1 workers	56.3/10.8	-	-	-
2 workers	57.6/9.0	56.7/5.3	-	-
3 workers	52.7/3.2	51.3/3.2	50.0/3.2	-
4 workers	53.8/2.9	53.8/3.2	53.6/3.0	53.5/3.1

[Table 2 - Average of CPU usage (user /system) -Task 3]

	node0	node1	node2	node3
2 workers	31.8	31.8	-	-
3 workers	64.7	64.4	64.8	-
4 workers	70.1	70.6	70.6	70.6

[Table 3 -Average of Network Transmission (MB/s) - Task 3]

Conclusion:

From the PyTorch paper, one of the main goals for PyTorch is that the distributed API is non-intrusive. PyTorch successfully implements a seamless distributed integration; we were able to use the built in API with just a few additional lines of code. Beyond that, using a GPU was even simpler, we needed to change one parameter to use a GPU. PyTorch does a fantastic job of abstracting the model design from the underlying hardware which processes the model training.

Due to PyTorch's model and tensor abstractions, the different setups for task 1, 2, 3 had surprisingly similar source code that belies the underlying complexity. The backward pass greatly benefits from a distributed setup, however the distributed setup also incurs communication overhead. The communication overhead can be minimized by using ring communication patterns and using asynchronous communication via buckets to maximize the computation and network communication overlap. However, from task 3 we see that there is little improvement by increasing the world from 3 to 4 nodes because the backward pass is dominated by the communication overhead. VCG-11 is a fairly small model, and the dataset is only a few hundred megabytes. If the model or dataset were larger this would increase the computation time on each node and reduce the percentage of time spent on communication.

Contributions:

Shing Ng - Task 2a code, editing the report

Jiun-Ting Chen - Task 3 code and writeup, conducting experiment on Task 2, Task3, dstat graphs and tables for all tasks, analysis of task 2 and task 3, editing the report

Matt Blakely - scripts to initialize servers, workflow to leverage github to keep servers in sync, task 1, 2b, and time tracking code. Intro, code structure, all task training time analysis, conclusion sections of the report, editing the report.