Implementing Job Scheduling for Distributed Training

A Technical Report Submitted to the

Faculty of the School of Engineering and Applied Science
University of Virginia • Charlottesville, Virginia

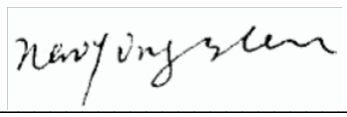
In Partial Fulfillment of the Requirements of the Degree
Bachelor of Science, School of Engineering

Brian Yu
Spring 2021

On my honor as a University Student, I have neither given nor received unauthorized aid on this assignment as defined by the Honor Guidelines for Thesis-Related Assignments

Signature  Date 5/4/21

Brian Yu

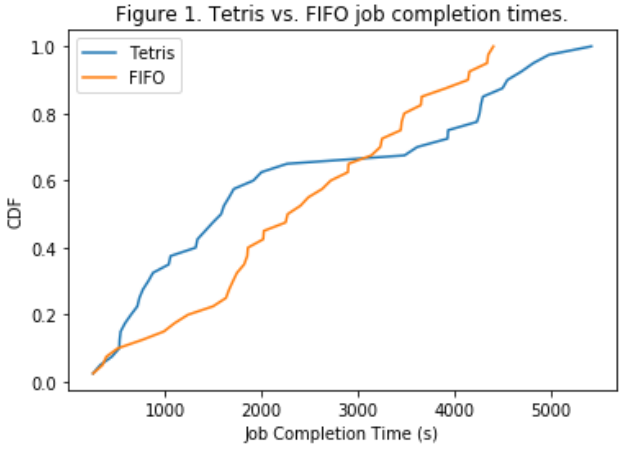
Approved  Date 5/4/21

Professor Haiying Shen, Department of Computer Science

Implementing Tetris in Simulation

In order to learn more about scheduling algorithms and distributed training of deep learning training jobs, I first studied prior art, reading the Tetris, Kairos, and Gandiva papers [1, 2, 3]. I then implemented a simulation of the Tetris scheduler. Tetris places tasks on the node with resource availabilities that best match the tasks' resource requirements. Tetris considers all axes of resource usage (CPU, memory, disk, and network) as a vector and assigns tasks to the node whose resource availability vector has the highest cosine similarity to the task resource requirements vector [1]. Thus, more tasks can be packed on nodes and resource fragmentation is reduced. The Tetris scheduler was compared to a FIFO scheduler that simply placed tasks on the first node that could accommodate each task. A workload consisting of 40 jobs, each with 30 tasks and heterogeneous resource requirements, was run on a cluster of 20 simulated nodes. Jobs arrived at uniformly random times during the first 3000 seconds of the simulation. The Tetris scheduler had a makespan of 5519 seconds while the FIFO scheduler had a makespan of 7083 seconds, a 22.1% improvement. Average job completion times were 2230 seconds for Tetris and 2415 seconds for FIFO. The CDFs of job completion times for both schedulers are shown in Figure 1.

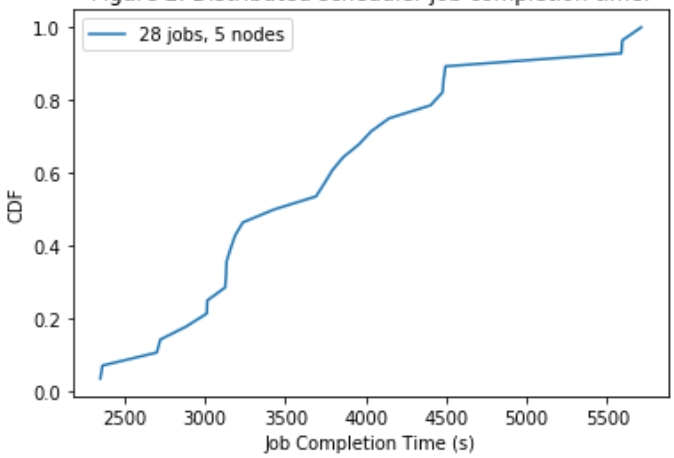
Figure 1. Tetris vs. FIFO job completion times.



Implementing Gandiva

Next, we implemented a simplified version of the Gandiva scheduler. Gandiva relies on efficient suspend and resume mechanisms that allow deep learning training (DLT) jobs to be migrated efficiently between different nodes [3]. This scheduler consists of two main components: a node daemon responsible for managing nodes and a master scheduler responsible for scheduling jobs on different nodes. Each DLT job is split into scheduling intervals consisting of 1 epoch. When the scheduler assigns a job, it sends training requests over a socket to the node daemons on both a worker node and a parameter server node. The daemon on the worker node then fetches the model's weights from an FTP server running on the worker node that the model was most recently trained on. The worker and parameter server then train the model for an epoch and report back to the master scheduler. The daemon is also responsible for removing stale model weight checkpoints and logging various statistics for each job, such as accuracy, loss, and epoch training times. These statistics are then aggregated and analyzed by the master scheduler after training is complete. The CDFs of job completion times for this scheduler is shown in Figure 2.

Figure 2. Distributed scheduler job completion time.



Results

By executing the DLT jobs on a distributed cluster, we hoped to beat non-distributed training performance 100% of the time. However, Figure 3 shows that this was not the case. As the number of jobs in the workload increased, the percentage of jobs that were completed before the deadline decreased. To investigate why performance was poorer than expected, we compared the distributed AlexNet model to its non-distributed equivalent. We noticed that training the distributed model on 1 parameter server and 1 worker was many times slower than training the non-distributed model. After experimenting with different hyperparameter changes, we concluded that the mini-batch iteration size was to blame. As shown in Figure 4, training an epoch of AlexNet took 330 seconds using a batch size of 16. Increasing the batch size to 128 decreased the time to 49 seconds. Meanwhile, the training time for non-distributed AlexNet was stable at 5-8 seconds per epoch, regardless of batch size. After exploring the Tensorflow 1.15 source and reading previous studies, we discovered that in native distributed TensorFlow, workers send gradient updates to parameter servers after each mini-batch iteration [4]. Thus, increasing the batch size decreases the number of times that a worker has to communicate with the parameter servers over the network. Indeed, Malik et al. [5] showed that larger batch sizes result in significantly increased distributed training performance.

Figure 3. Job deadline performance on 2 nodes.

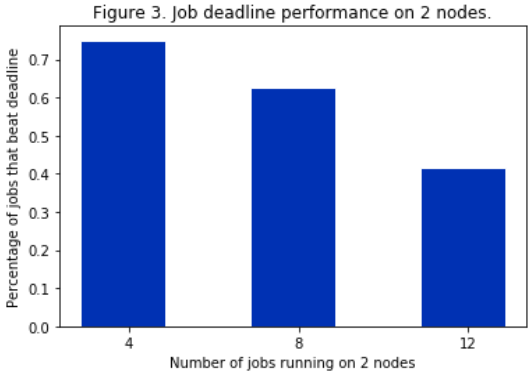
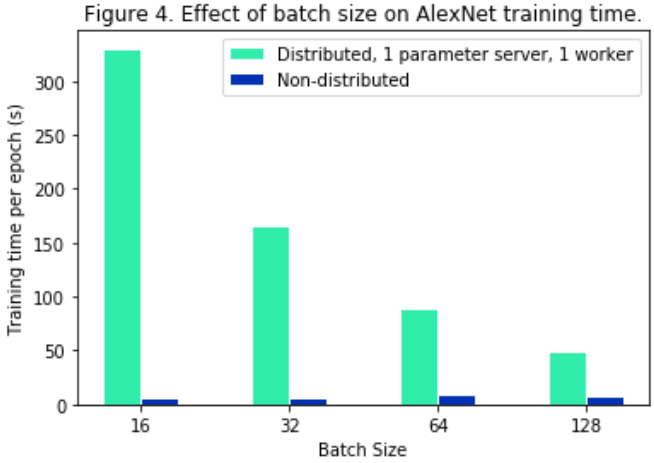


Figure 4. Effect of batch size on AlexNet training time.



Future Work

There are many areas to explore in future work. To avoid the performance bottleneck caused by TensorFlow's native parameter servers, a more efficient inter-GPU communication method for parameter averaging could be employed, such as the ring all-reduce algorithm used in the Horovod distributed training framework [6]. More deep learning models and algorithms could be adapted for distributed training in order to test the scheduler on a more heterogeneous workload. Further research could be conducted into what hyperparameters and factors besides batch size most affect distributed training, and how different distributed training strategies (e.g. synchronous/asynchronous training and in-graph/between-graph replication) affect training performance. Furthermore, upgrading to TensorFlow 2.0 would aid in the implementation of distributed deep learning models. Tensorflow 2.0 provides support for more varied and easier to implement distributed training strategies and allows both Keras and TensorFlow models to be easily trained in a distributed manner.

References

[1] Robert Grandl, Ganesh Ananthanarayanan, Srikanth Kandula, Sriram Rao, and Aditya Akella. 2014. Multi-resource packing for cluster schedulers. SIGCOMM Comput. Commun. Rev. 44, 4 (August 2014), 455-466. DOI: https://doi.org/10.1145/2740070.2626334

[2] Pamela Delgado, Diego Didona, Florin Dinu, and Willy Zwaenepoel. 2018. Kairos: Preemptive Data Center Scheduling Without Runtime Estimates. In Proceedings of the ACM Symposium on Cloud Computing (SoCC '18). ACM, New York, NY, USA, 135-148. DOI: https://doi.org/10.1145/3267809.3267838

[3] Wencong Xiao, Romil Bhardwaj, Ramachandran Ramjee, Muthian Sivathanu, Nipun Kwatra, Zhenhua Han, Pratyush Patel, Xuan Peng, Hanyu Zhao, Quanlu Zhang, Fan Yang, and Lidong Zhou. 2018. Gandiva: introspective cluster scheduling for deep learning. In Proceedings of the 12th USENIX conference on Operating Systems Design and Implementation (OSDI'18). USENIX Association, Berkeley, CA, USA, 595-610.

[4] Jim Dowling. 2017. Distributed Tensorflow. (December 2017). Retrieved December 1, 2019 from https://www.oreilly.com/ideas/distributed-tensorflow

[5] Abid Malik, Yuewei Lin, Shinjae Yoon, Nathaniel Wang, and Michael Lu. 2018. Detailed Performance Analysis of Distributed Tensorflow on a GPU Cluster using Deep Learning Algorithms. New York Scientific Data Summit 2018 (NYSDS'18). Retrieved from https://www.bnl.gov/nysds18/files/talks/session3/malik-nysds18.pdf

[6] Alexander Sergeev and Mike Del Balso. 2018. Horovod: fast and easy distributed deep learning in TensorFlow. arXiv:1802.05799. Retrieved from https://arxiv.org/abs/1802.05799