Doing More by Doing Less: How Structured Partial Backpropagation Improves Deep Learning Clusters

Adarsh Kumar
adrshkm@amazon.com
Amazon Alexa AI

Kausik Subramanian
University of Wisconsin, Madison

Shivaram Venkataraman
University of Wisconsin, Madison

Aditya Akella
University of Texas at Austin

ABSTRACT

Many organizations employ compute clusters equipped with accelerators such as GPUs and TPUs for training deep learning models in a distributed fashion. Training is resource-intensive, consuming significant compute, memory, and network resources. Many prior works explore how to reduce training resource footprint without impacting quality, but their focus on a subset of the bottlenecks (typically only the network) limits their ability to improve overall cluster utilization. In this work, we exploit the unique characteristics of deep learning workloads to propose Structured Partial Backpropagation(SPB), a technique that systematically controls the amount of backpropagation at individual workers in distributed training. This simultaneously reduces network bandwidth, compute utilization, and memory footprint while preserving model quality. To efficiently leverage the benefits of SPB at cluster level, we introduce Jigsaw, a SPB aware scheduler, which does scheduling at the iteration level for Deep Learning Training(DLT) jobs. We find that Jigsaw can improve large scale cluster efficiency by as high as 28%.

ACM Reference Format:
Adarsh Kumar, Kausik Subramanian, Shivaram Venkataraman, and Aditya Akella. 2021. Doing More by Doing Less: How Structured Partial Backpropagation Improves Deep Learning Clusters. In *2nd International Workshop on Distributed Machine Learning DistributedML 2021 (DistributedML'21), December 7, 2021, Virtual Event, Germany*. ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3488659.3493778

1 INTRODUCTION

With the widespread use of deep learning models in a variety of applications, many organizations are employing large clusters of machines equipped with hardware accelerators for model training. These clusters are expensive to build and are power hungry. They are also oversubscribed with Deep Learning Training(DLT) jobs often having to wait in queues for hours to days before getting scheduled [10, 11]. Thus, understanding how to improve the utilization of such clusters is an important goal. In this paper, we develop a novel technique for improving deep learning cluster efficiency that is centered around the idea of having **DLT jobs do *less*** work which can be systematically leveraged to **run *more* training jobs in a highly efficient manner**.

Today, training of DNNs is typically done using algorithms like stochastic gradient descent (SGD), an iterative algorithm where every iteration computes gradients by running a forward pass followed by a backward pass. To scale SGD across machines the most commonly used method is data parallel distributed training, where gradients computed at each worker are aggregated after every iteration of training. SGD can take thousands of iterations to converge and thus training a deep learning model can take days or weeks [11].

Many recent works [2, 14, 22] have considered how to reduce the resources required by a given job with the primary focus being algorithmic techniques that can approximate gradients exchanged between workers during distributed training without affecting accuracy. Most of these techniques aim to **solely** minimize network communication and examples include quantizing gradients [2], dropping parts of the gradient [14] or using low-rank updates [22, 23].

Unfortunately, these techniques provide limited improvements for deep learning clusters equipped with modern hardware that are used to train multiple models simultaneously [10, 11, 25]. This is primarily due to following reasons: First, while existing approaches optimize communication, they fail to reduce the total amount of compute resources or memory required during training. For example, if a gradient dropping scheme only preserves the top-20% of the gradient values for communication, we still need to compute the gradients for all the parameters during the backward phase, ending up using the **same** compute. Thus, prior approaches simply shift bottlenecks from the network to compute in large multi-tenant clusters. Further, while running in a multi-tenant cluster, the goals for an organization are typically to improve overall cluster-wide utilization [25] (e.g., reduce makespan given a set of jobs) and only reducing the network resources required provides limited cluster-wide benefits.

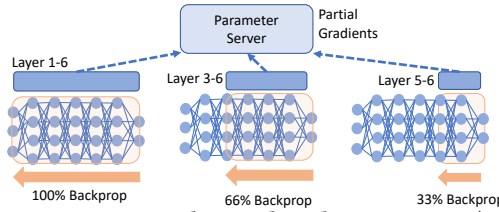


Figure 1: Structured Partial Backpropagation (SPB)

This prompts us to ask the following question:

DistributedML'21, December 7, 2021, Virtual Event, Germany

ACM ISBN 978-1-4503-9134-4/21/12…$15.00
https://doi.org/10.1145/3488659.3493778

% Back-prop	Forward Pass (ms)	Backward Pass(ms)	Peak Mem (GB)
100	34.79	79.89	6.5
90	34.85	70.24	5.7
80	34.71	58.86	5.1
70	34.61	49.92	4.6
60	34.72	43.74	4.3
50	34.70	36.58	4.1
40	34.61	29.93	3.8
30	34.67	23.02	3.5
20	34.84	16.34	3.3
10	34.86	8.35	3.1
5	34.64	3.56	2.9

Table 1: Effect of % partial Backprop on Resource Utilization of ResNet101 model. BatchSize:64, Tesla V100

"How can we design a distributed training scheme that can provide cluster-wide benefits in a multi-tenant setting?"

To improve the overall cluster efficiency we need to improve the resource utilization of jobs across both compute as well as communication dimension. Thus, techniques that avoid computation of gradients which do not need to be communicated can lead to savings across both resources. Based on this insight, we design Structured Partial Backpropagation (SPB), a new gradient compression scheme that can provide cluster-wide benefits.

Our first observation in designing SPB is that similar to existing schemes like top-K gradient dropping, we can approximate the gradient by updating different parts of the overall gradient from different workers. For example, consider a job which is spread across three workers as shown in Figure 1. In this case, two of the three workers compute partial gradients (33% and 66%) which are aggregated to derive an overall approximate gradient. Assuming synchronous SGD, the above described scheme does not improve the time taken by a single iteration as there is one worker which needs to compute the entire gradient. However, this scheme does yield cluster-wide benefits as it leads to an aggregate reduction in **network** utilization (similar to the top-50% scheme), while also saving **compute** resources along with **memory** as we describe next.

The compute and memory savings in SPB arise from the way gradients are computed for deep learning jobs. To compute a gradient, first a forward pass is performed that computes the loss value for the data examples with respect to the current model. Following this, a backward pass is performed to compute the final gradient values. When using SPB, we conduct the backward pass for a fraction of the layers from the end[1] of the DNN network, and this leads to reduced computation as shown in Figure 1. The fraction of layers are carefully chosen to be the suffix layers as computing gradients for layer k requires gradients from all layers l, where $l \geq k$.

SPB can also save memory, as we can avoid storing intermediate outputs for layers that are not going to participate in the backward pass. This opens the door for space multiplexing in hardware accelerators.

[1]By "end" we refer to the final output layer of the DNN

We theoretically prove that SPB converges in the same order as SGD and has bounded impact on model quality in a limited setting and empirically show minimal impact on model quality across 10 models ranging from computer vision models (e.g., ResNet-50, VGG-19 etc.) to NLP models (CNN-Text, LSTM etc.).

While SPB can help reduce the resource requirement for a single job, there is limited impact on cluster-wide resource utilization without corresponding support from the job schedulers. Today, schedulers treat DLT jobs as fixed-work units and perform gang scheduling such that every worker receives the same amount of compute, memory and network resources. Thus, even if SPB were used, existing schedulers would not be able to capitalize on resources that become available (e.g., $2^{nd}/3^{rd}$ worker in Fig 1).

To this end, we design a new cluster scheduler, Jigsaw, that can account for the variable amounts of computation that are created as a result of SPB and pack computations from different workloads at fine time-scales to improve overall cluster utilization. Performing such packing requires reasoning about when and how much resources are going to be available in the future. We perform fine-grained iteration-level scheduling and exploit the repetitive nature of DL workloads to derive a centralized scheduling policy that intelligently places tasks to maximize resource utilization.

Our experiments on a 45-GPU cluster show that Jigsaw, in conjunction with SPB, can improve the makespan and cluster utilization by 28% while running a publicly available trace of DL jobs [11]. We also show that Jigsaw is more tolerant to resource contention in comparison to other state of the art DLT schedulers.

2 STRUCTURED PARTIAL BACKPROP

We now describe Structured Partial Backpropagation, a distributed training scheme that can provide benefits across multiple resource dimensions. Consider a distributed training job running with k workers, training a L layer DNN. We assume a parameter server (PS) setup [12] for data parallel training where each worker computes the gradient for all L layers and the gradient values are aggregated at the PS. The PS maintains a global copy of the model, which is updated after every iteration.

We first describe the changes in the role of each worker with SPB. At the start of an iteration i, as in existing frameworks, each worker fetches the latest model version from the parameter server (PS). To make sure all training data points contribute equally, each worker samples its batch of data from the entire dataset and performs a forward pass to obtain the loss with respect to the model. The main changes occur during the backward pass: instead of performing the entire backward pass i.e. computing the gradient for all the layers, each worker performs backpropagation on a fixed suffix of the network. As shown in Figure 1, the j^{th} worker does backward propagation for $\frac{jL}{k}$ of the layers starting from the output layer of the model. Finally, each worker sends the gradients computed to the PS.

Similar to existing setups, the PS in SPB sends the latest version of the model to all workers at the start of an iteration. The main difference occurs in the aggregation phase: the PS now receives partial gradient from each worker and needs to aggregate them to update the model. The PS computes a weighted average, i.e., a layer's gradient update is averaged by the effective number of

Table 2: Median time and memory usage during forward and backward pass, and gradient size of different DNN architectures on Tesla V100 (batch size of 128).

Model	Forward		Backward		
	Time (ms)	Mem (GB)	Time (ms)	Mem (GB)	Grad (MB)
ResNet18	9.19	0.05	21.49	2.46	44
ResNet34	16.11	0.08	36.69	3.08	85
ResNet50	36.32	0.09	78.9	7.33	94
ResNet101	60.51	0.17	135.14	9.79	170
ResNet152	86.9	0.23	197.05	12.81	232
VGG19	6.82	0.08	16.31	2.02	80
VGG16	5.68	0.06	13.96	1.97	59
VGG11	3.34	0.04	7.8	1.83	36
GoogleNet	41.33	0.05	99.17	5.96	24

workers that contributed to it. Accordingly, the learning rate is also scaled with respect to the number of workers contributing towards the gradient.

The above *structured* approach to backpropagation has following notable aspects:

Resource Savings: As shown in Table 2, the backward pass uses more memory and takes longer to complete for DNNs. By doing backward pass for fraction of layers, SPB can save both compute and memory, thus enabling better time multiplexing of a GPU across jobs. Table: 1 shows the savings when partial backpropagation is done for ResNet 101 model.

Gradient Dependency: Compute can only be saved with our proposed SPB approach. Avoiding backpropagation in Random-k or Top-k dropping is not possible because of the dependency of the gradients of early layers on later ones.

Per-Iteration Time: On average, SPB reduces workers' running time. However, the per-iteration time remains the same as in synchronous SGD training as there's one worker that does complete backward propagation of all L layers. Also, unlike previous gradient compression techniques like PowerSGD [22] or Atomo [23], we do not add any additional compute overhead to perform gradient compression.

2.1 Theoretical Analysis

We now present a worst-case theoretical analysis that proves that SPB converges to an equivalent solution as traditional distributed SGD. We approach this problem by first presenting the convergence rate for traditional distributed SGD and then compare this to the convergence rate when using SPB.

Assumptions and Notation. We denote X as the trainable parameters of the model and f as the loss function. To simplify our theoretical analysis we assume that $f : X \rightarrow \mathbb{R}$ is a convex function with β-smoothness[2].

Given these assumptions the convergence rate for SGD is known from [6] to be:

Theorem 2.1. *Let X be convex compact set and $R^2 = sup_{x,x_1 \in X} ||x - x_1||^2$. If the approximate gradient $\tilde{g}(x)$ is such that $E||\nabla f(x) - \tilde{g}(x)||^2 \leq V^2$, after t iterations, SGD with step size $\frac{1}{\beta + \frac{1}{\eta(t)}}$, where $\eta(t) = \frac{R}{\sigma}\sqrt{\frac{2}{t}}$, achieves*

$$E(f(\frac{1}{t}\sum_{s=1}^{t} x_{s+1}) - f(x^*)) \leq R\sqrt{\frac{2V^2}{t}} + \frac{\beta R^2}{t} \quad (1)$$

The above result bounds the difference between $f(\hat{x})$ after t iterations and the optimal value $f(x^*)$ in terms of V^2, which is a bound on how far our estimated gradient is from the true gradient.

Distributed Mini-batch SGD. Now, we consider the scenario where we have k workers each processing a batch $\frac{B}{k}$ at every iteration. We first consider the scenario where every worker computes the entire gradient and attempt to bound the value of $E||\nabla f(x) - \tilde{g}(x)||^2$.

Lemma 2.2. *With k workers, an overall batch size of B and assuming $||g(x_i)|| < P \; \forall i$, for mini-batch SGD, we have $E\|\nabla f(X) - \tilde{g}(x)\|^2 \leq 2P^2\frac{k}{B}$*

SPB Convergence Rate We next consider the case when SPB is used to prune gradients. The main difference here is that i^{th} block of the gradient vector will be updated by i workers.

Theorem 2.3. *When using* SPB *with k workers and overall batch size of B and assuming $||g(x_i)|| < P \; \forall i$, we have $E\|\nabla f(X) - \tilde{g}(x)\|^2 \leq 2P^2\frac{k}{B}log(k)$*

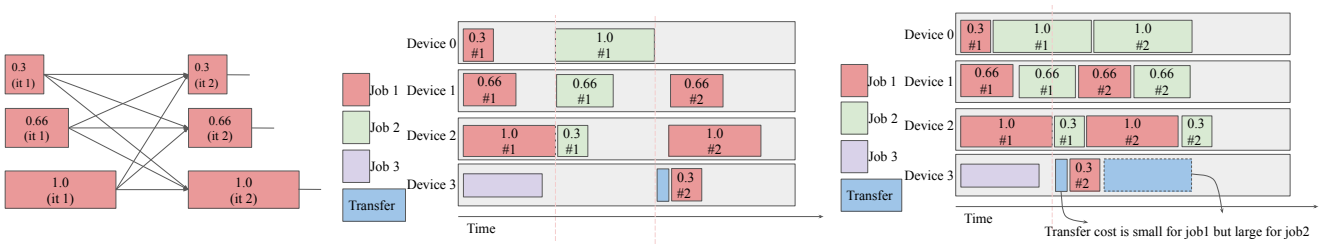
The above result shows that when using SPB we can similarly bound the value of $E||\nabla f(x) - \tilde{g}(x)||^2$, but the bound here has an additional factor of $log(k)$ where k is the number of workers. Applying this result to Theorem 7.1, we see that the convergence rate when using SPB is, in the worst case, $log(k)$ slower, i.e., when using SPB we might require a factor of $log(k)$ more iterations to converge to a similar quality result. However, this is a worst-case bound for convex functions. We empirically show how SPB performs in practice for real-world deep learning models next.

3 JIGSAW: A SPB AWARE SCHEDULER

In this section, we describe the Jigsaw scheduler and how it is designed to leverage the gains obtained from our SPB to maximize cluster efficiency. We first describe some unique properties of SPB jobs that our scheduler needs to consider. Following that we present our scheduling algorithm that aims to improve cluster utilization and makespan.

Figure 2(a) shows the task DAG of a single deep learning training job trained with SPB with three workers; for each iteration, we have multiple tasks running on parallel workers. Once the workers have computed the gradients for iteration i and updated the parameter server, we can schedule tasks corresponding to iteration $i + 1$. Each task has two resources in its specification: peak GPU memory required, and running time. We note that these resource specifications can be obtained by profiling a job for a few iterations [25]. Different tasks in the same iteration can have different resource characteristics due to SPB. Given a set of DLT task DAGs, the Jigsaw scheduler computes *when* a task should run, and on *which* machine.

[2] A continuously differentiable function f is β-smooth if the gradient ∇f is β Lipschitz continuous.



(a) Job DAG Structure **(b) Gang Scheduling with SPB** **(c) Iteration-level scheduling with SPB**

Figure 2: (a) DAG of a DNN training job trained with SPB with arrows indicating dependencies. Tasks of iteration 2 can run only after all tasks of iteration 1 have completed. Worker sizes indicate varying work distribution. Example of (b) gang scheduling versus (c) iteration-level scheduling. Gang scheduling, which operates at a job-level, cannot utilize the resource savings generated by SPB.

3.1 Iteration-Level Scheduling

Existing state of the art DLT schedulers like Gandiva [25] and Tiresias [10] perform scheduling at the granularity of jobs, and rely on the paradigm of *gang scheduling*, i.e., for each iteration, all parallel workers begin and end at the same time. However performing scheduling only when all workers have finished one (or more) iterations is insufficient to make use of resources saved by SPB.

Consider the example shown in Figure 2(b). In this case we have two parallel DLT jobs (Job1, Job2) each of which has workers using SPB. Thus one of the workers finishes in 0.3s while the other two take 0.6s and 1s respectively. With gang scheduling we can see that the scheduler is only invoked once all tasks of the iteration complete. Thus even though Device0 has finished worker0 of Job1 at 0.3s, it remains idle for 0.7s before being assigned a new task. By scheduling at the *granularity of iterations*, we can utilize the gains from SPB and utilize the cluster in a more effective way. In Figure 2(c), we can see that each worker in a job can start at a staggered time (e.g. for Job2) but they still follow the dependencies where one iteration is completed before the next one starts. In this case as soon as Device0 is done with worker0 of Job1 at 0.3s, it can be assigned to start Job2 thus improving resource utilization and makespan for the cluster.

Scheduling overheads. However performing iteration-level scheduling can be challenging as there is additional overhead for a centralized cluster scheduler to come up with iteration-level schedules as iterations can be very short (100ms). To overcome the challenge of generating schedules efficiently, in Jigsaw we utilize the fact that ML-training jobs are iterative and thus the scheduler has visibility into what tasks may arrive in the future from a given job. Thus we perform coarse-grained scheduling (on the granularity of minutes) of fine-grained iterations [21].

Task switching overheads. Switching between tasks at a fine granularity on a device can be expensive, especially when workers are moved to a new device. This arises from the fact that moving a worker to a new device entails fetching the model's computation graph and its latest parameters to the GPU before training can resume. We find that the model parameters and computation graph are typically much smaller [26] compared to the intermediate data generated. Thus for workers that are time-sharing in the same device (e.g. Device1 in Figure 2(b) we retain the model parameters in the global memory of the GPU and this helps reduce switching overheads. For moving models across machines, as shown (e.g. Device3 in Figure 2(b)), we account for the overhead of data transfer as a part of scheduling.

3.2 Scheduling Algorithm

We next describe our scheduling algorithm. The scheduler computes the task placement for a scheduling interval $\mathcal{I}$. Our heuristic places schedulable tasks (taking into account iteration dependencies) in a 2-dimensional space of resources (compute/device) and time (RT-Space) and tries to minimize makespan.

Task Prioritization. All tasks that are ready to run are placed in priority queue. Similar to existing big-data schedulers that aim to improve packing [8, 9], we prioritize tasks that require greater amount of resources. We calculate this priority as a normalized product of the task's GPU resource and running time. Given the dependency structure across iterations, placing the task with the highest resource requirement first also ensures that overall job progress does not get delayed.

Task Placement. Given a task and a partially occupied RT-Space, our scheduler next decides when a task must be run and on which machine. There are multiple considerations which decide placement of a task - first and foremost, we must wait for tasks corresponding to the previous iteration to finish before we can schedule the current task. Furthermore, the machine must have resources available for the entire duration of the task.

The scheduler first finds the earliest time the task can be scheduled to run based on the end times of the tasks of the previous iterations T_{i-1}. However, not all machines can start the task immediately after T_{i-1}. If a machine ran the task corresponding to the same worker and the $i-1$ iteration, the task is immediately schedulable as the machine can reuse the model computation graph from previous iteration. For the other machines, we add a startup time which is dependent on the size of the model ($\gamma \times model\_size$) to capture the delay in transferring the model to a new machine. To reduce makespan, we greedily schedule the task on a machine where the task can start the **earliest** of all machines. Since the scheduler chooses a machine where we can start the task the earliest, it will naturally prefer machines which ran the previous iteration tasks and avoid unnecessary task movement across iterations.

4 EVALUATION

4.1 SPB effect on Model Quality

We first present empirical results on the convergence of popular deep learning models when using SPB while training. We consider models from a number of domains including computer vision, NLP etc. and also consider models of different complexity (i.e., depth or number of layers).

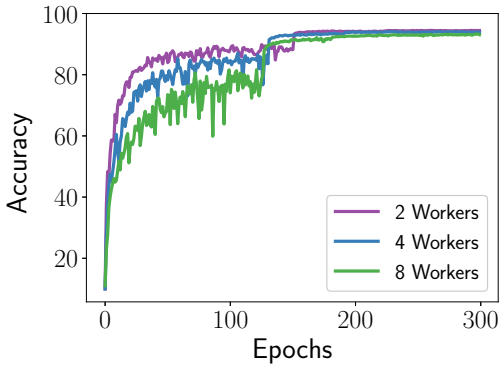


Figure 3: Convergence of ResNet18 on CIFAR10 as we vary the number of workers. (SPB**)**

For CV, we considered image classification with CIFAR10 and CIFAR100 datasets and a number of DNN families (ResNet, VGG, GoogLeNet, and DenseNet). We used SGD with momentum and 10^{-4} weight decay as our optimizer configuration. We tried 3 learning rates for each model and picked the one attaining best accuracy. For NLP, we selected models for classification on the MR dataset [16] and language modeling on WikiText [15]. We use 4 workers, each running on an NVIDIA V100 GPU, for all the experiments.

We present a comparison of the accuracy achieved when using SPB vs. a baseline where models are trained using standard distributed SGD optimization in Table 3.

Our results from Table 3 shows that there is a minimal effect on the accuracy achieved when models are trained with SPB. Even though the DNN early layers are getting updated by fewer workers, model convergence is not significantly impacted. We attribute this behavior of SPB to the fact that early layers converge faster than later layers, and thus require fewer gradient descent steps. This has also been observed by previous work [5, 18].

Further, we note that using SPB for a more complex task could lead to a bigger difference in accuracy (though still under 2%). For example, with ResNet18, the accuracy difference due to SPB increases from 0.06% to 1.52% as we go from CIFAR10 to CIFAR100. This we believe can be overcome by warm up epochs, where for initial N epochs we train the model using the full gradient from all the workers.

Figure 3 shows the convergence of ResNet18 model when varying the number of workers on the CIFAR10 dataset while using SPB. We maintain a batch size of 128 per worker. From the figure we see that as we increase the number of workers it takes more iterations to converge to the best accuracy. Our theory also suggests this in terms of the $log(k)$ factor in the convergence rate for k workers, albeit for convex functions. Also, we find that the maximum accuracy

Dataset	Model	Accuracy SPB	Accuracy SGD
CIFAR10	ResNet18	93.96	94.02
	ResNet34	93.81	94.96
	DenseNet121	94.18	94.35
	GoogleNet	94.07	93.83
	VGG16	91.93	92.35
CIFAR100	ResNet18	71.31	72.83
	ResNet34	69.94	71.2
MR	CNNText	77.12	77.35
	LSTMText	78.02	80.23
WikiText 2	LSTM	105*	103*

Table 3: Comparison of maximum accuracy of various models w/o SPB. *Perplexity is reported for language model on WikiText.

achieved using 8 workers is slightly lesser than that achieved with 4 workers. We believe that this could be related to using a larger batch size in this setting. We discuss ways in discussion section to close this gap in accuracy.

4.2 JIGSAW effect on Cluster Efficiency

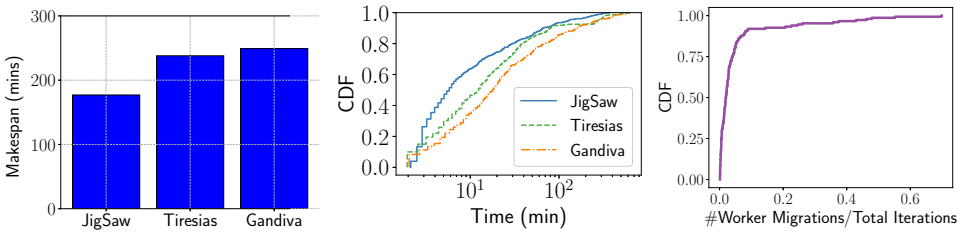
In this section, we use a mix of simulations and cluster experiments to present the performance of our system JIGSAW and compare with other state of the art DLT schedulers.

Setup: For our simulations, we created a discrete event based simulator which we run using a large scale DLT job trace from Microsoft [10, 11]. We enriched the trace with additional information, specifically peak memory usage, execution time, gradient size and model size etc. by profiling models used in Gu et al. [10] trace. To enable SPB, we also profiled execution time of DLT workers when varying the proportion of layers that perform backpropagation. All our profiling was done in isolation on Tesla V100 GPU. To account for migration overheads, we measure the transfer cost of moving models across different machines and the network overhead in PS-worker communication.

As described in Section 3, JIGSAW creates a schedule for all the active DLT jobs for next T minutes considering the complete cluster and communicates this schedule to the individual machines. In our experiments we have kept T, the scheduling interval to be 1 min. While creating the scheduling plan for next scheduling interval, JIGSAW considers the affinity of tasks towards the machine they were executed previously to avoid overheads from migration.

Workload: We have evaluated JIGSAW on publicly available workload of DNN training from Microsoft [10] which was scaled down for practical purposes. This workload contains around 500 DLT Training Job, with each job arriving at an average interval of 30 seconds. Around 50% of the DLT jobs had 1 worker and the rest are multi-worker DLT jobs, with 10% having 2, 20% having 4, 15% having 8 and 5% having 16 workers. A DLT job is considered finished once a pre-specified number of iterations has been executed.

Metrics: To quantify and compare the performance of JIGSAW, we use two metrics: **Job Completion Time(JCT):** JCT is defined as the time taken to finish one job from the time it was submitted. This also includes any waiting time which the job had to incur. **Makespan:** Makespan is the end time of the last job to finish in



(a) Makespan comparison on 45 GPU Cluster **(b) JCT comparison 45 GPU Cluster** **(c) Worker Migrations**

Figure 4: [Simulated]: (a) Makespan comparison on 45 Tesla V100 cluster. (b) CDF of JCT for different schedulers (c) CDF for fraction of iterations in which DLT jobs were migrated to another machine

a trace. Improving makespan, correlates with improvement in resource utilization.

Baselines: We evaluate our system Jigsaw against two state of the art DLT schedulers: Gandiva [25], which improves cluster efficiency by packing jobs on as few machines as possible and Tiresias [10], which schedules job according to Least Attained Service(LAS) policy so that all jobs obtain equal service over time. Both baselines are run standard DLT jobs, without SPB, as their APIs do not allow workers of same DLT job to have different resource requirements.

Results: Figure 4 we compare makespan and JCT for Tiresias and Gandiva with Jigsaw. As can be seen, on 45 GPU Cluster, Jigsaw improves makespan by around 22% over Tiresias and 28% over Gandiva.

Similarly, as can be seen from the CDF plot of JCT, our system Jigsaw improves the JCT of DLT jobs against Gandiva and Tiresias on a 45 GPU Cluster.

4.3 Testbed Overheads

We discuss the systems overheads observed in our testbed.

Model Movements: Migrating DLT job's worker across different machines is costly [25]. Since Jigsaw does fine grained scheduling at the level of iteration, migrating the workers of DLT jobs at every iteration would result in significant overheads. Jigsaw mitigates this by considering the migration overhead while deciding the placement of a task. In Figure 4(c) we plot the CDF of fraction of iterations for which a worker had to be migrated to different machine. Fraction of iteration is calculated as total migrations divided by total iterations (lower the better). As can be seen from the plot, even after scheduling at iteration level Jigsaw has low migration overhead.

Time Multiplexing Overhead: To improve the cluster utilization, Jigsaw also does time multiplexing of DLT tasks. SPB creates opportunity for time multiplexing by finishing tasks earlier by doing partial backprop. To study the overhead of time multiplexing of DLT tasks, we ran 500 iterations of 5 real DNN models from Computer Vision and NLP family in sequence and in a round robin manner. In sequence, the models are trained for 500 iterations one after the another, while in round-robin we train a model for an iteration and then we switch to another model and proceed in a round-robin manner. We found the slowdown to be less than 1% during round-robin training, thus, the overhead of time multiplexing **within a GPU** is minimal.

5 RELATED WORK

DL training schedulers: There have been several recent works on improving cluster level scheduling of DL training jobs, with each having a different objective function [10, 17, 25, 27]. For example, Gandiva [25] focuses on improving overall cluster efficiency, while Tiresias [10] optimizes on job completion time and fairness. Similar to Tiresias, Optimus [17] also focuses on average job completion time, while SLAQ [27] optimizes on model quality. All existing DL training schedulers gang-schedule and assume equal resource requirement for all the workers and hence cannot capitalize on resource savings from SPB.

Gradient Pruning: Ali et al [1] introduced the idea of gradient dropping. Strom et al [13, 20] introduced dropping gradients larger than a threshold and Adacomp [7] proposed automatically tuning it. To further improve the convergence of gradient dropping schemes, Chen et al [3] introduced layer normalization. Other works in this domain involve gradient quantization [4, 19, 24], using low rank estimation of gradients'[22]. All the above works focus on a single job setting where compute is not a contended resource, and thus focus solely on improving communication. We introduce a new structured pruning approach that can save compute, memory, and network resources.

6 CONCLUSION

We show how using a structured approach to partial backpropagation can improve resource utilization of DL training jobs in shared clusters. We introduced a novel gradient pruning scheme called Structured Backprop Pruning, which saves compute, memory and network resources, while preserving model quality. To leverage the gains from SPB, we also introduced a new SPB-aware cluster scheduler Jigsaw that can improve utilization by performing better time and space multiplexing. We show that our approach of making the cluster scheduler aware of the ML workload properties can lead to significant benefits.

REFERENCES

[1] Alham Fikri Aji and Kenneth Heafield. 2017. Sparse Communication for Distributed Gradient Descent. *ArXiv* abs/1704.05021 (2017).

[2] Dan Alistarh, Demjan Grubic, Jerry Li, Ryota Tomioka, and Milan Vojnovic. 2017. QSGD: Communication-Efficient SGD via Gradient Quantization and Encoding. arXiv:1610.02132 [cs.LG]

[3] Jimmy Lei Ba, Jamie Ryan Kiros, and Geoffrey E. Hinton. 2016. Layer Normalization. *CoRR* abs/1607.06450 (2016).

[4] Jeremy Bernstein, Yu-Xiang Wang, Kamyar Azizzadenesheli, and Anima Anandkumar. 2018. signSGD: compressed optimisation for non-convex problems. In *ICML*.

[5] Andrew Brock, Theodore Lim, James M. Ritchie, and Nick Weston. 2017. FreezeOut: Accelerate Training by Progressively Freezing Layers. *ArXiv* abs/1706.04983 (2017).

[6] Sébastien Bubeck. 2014. Theory of Convex Optimization for Machine Learning. *ArXiv* abs/1405.4980 (2014).

[7] Chia-Yu Chen, Jungwook Choi, Daniel Brand, Ankur Agrawal, Wei Zhang, and Kailash Gopalakrishnan. 2017. AdaComp : Adaptive Residual Gradient Compression for Data-Parallel Distributed Training. *CoRR* abs/1712.02679 (2017).

[8] Robert Grandl, Ganesh Ananthanarayanan, Srikanth Kandula, Sriram Rao, and Aditya Akella. 2014. Multi-Resource Packing for Cluster Schedulers. *SIGCOMM Comput. Commun. Rev.* 44, 4 (Aug. 2014), 455–466. https://doi.org/10.1145/2740070.2626334

[9] Robert Grandl, Srikanth Kandula, Sriram Rao, Aditya Akella, and Janardhan Kulkarni. 2016. GRAPHENE: Packing and Dependency-Aware Scheduling for Data-Parallel Clusters. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. Savannah, GA, 81–97.

[10] Juncheng Gu, Mosharaf Chowdhury, Kang G. Shin, Yibo Zhu, Myeongjae Jeon, Junjie Qian, Hongqiang Harry Liu, and Chuanxiong Guo. 2019. Tiresias: A GPU Cluster Manager for Distributed Deep Learning. In *NSDI*.

[11] Myeongjae Jeon, Shivaram Venkataraman, Amar Phanishayee, Junjie Qian, Wencong Xiao, and Fan Yang. 2019. Analysis of Large-Scale Multi-Tenant GPU Clusters for DNN Training Workloads. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. USENIX Association, Renton, WA, 947–960.

[12] Mu Li, David G. Andersen, Jun Woo Park, Alexander J. Smola, Amr Ahmed, Vanja Josifovski, James Long, Eugene J. Shekita, and Bor-Yiing Su. 2014. Scaling Distributed Machine Learning with the Parameter Server. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation* (Broomfield, CO) *(OSDI'14)*. USENIX Association, USA, 583–598.

[13] Mu Li, David G. Andersen, Alexander Smola, and Kai Yu. 2014. Communication Efficient Distributed Machine Learning with the Parameter Server. In *Proceedings of the 27th International Conference on Neural Information Processing Systems - Volume 1* (Montreal, Canada) *(NIPS'14)*. MIT Press, Cambridge, MA, USA, 19–27.

[14] Yujun Lin, Song Han, Huizi Mao, Yu Wang, and William J. Dally. 2020. Deep Gradient Compression: Reducing the Communication Bandwidth for Distributed Training. arXiv:1712.01887 [cs.CV]

[15] Stephen Merity, Caiming Xiong, James Bradbury, and Richard Socher. 2016. Pointer Sentinel Mixture Models. *CoRR* abs/1609.07843 (2016).

[16] Bo Pang and Lillian Lee. 2005. Seeing Stars: Exploiting Class Relationships For Sentiment Categorization With Respect To Rating Scales. In *Proceedings of ACL*. 115–124.

[17] Yanghua Peng, Yixin Bao, Yangrui Chen, Chuan Wu, and Chuanxiong Guo. 2018. Optimus: an efficient dynamic resource scheduler for deep learning clusters. In *Proceedings of the Thirteenth EuroSys Conference*. 1–14.

[18] Maithra Raghu, Justin Gilmer, Jason Yosinski, and Jascha Sohl-Dickstein. 2017. SVCCA: Singular Vector Canonical Correlation Analysis for Deep Understanding and Improvement. *ArXiv* abs/1706.05806 (2017).

[19] Frank Seide, Hao Fu, Jasha Droppo, Gang Li, and Dong Yu. 2014. 1-bit stochastic gradient descent and its application to data-parallel distributed training of speech DNNs. In *INTERSPEECH*.

[20] Nikko Strom. 2015. Scalable distributed DNN training using commodity GPU cloud computing. In *INTERSPEECH*.

[21] Shivaram Venkataraman, Aurojit Panda, Kay Ousterhout, Michael Armbrust, Ali Ghodsi, Michael J Franklin, Benjamin Recht, and Ion Stoica. 2017. Drizzle: Fast and adaptable stream processing at scale. In *Proceedings of the 26th Symposium on Operating Systems Principles*. 374–389.

[22] Thijs Vogels, Sai Praneeth Karimireddy, and Martin Jaggi. 2019. PowerSGD: Practical Low-Rank Gradient Compression for Distributed Optimization. In *Advances in Neural Information Processing Systems 32*. 14259–14268.

[23] Hongyi Wang, Scott Sievert, Zachary Charles, Shengchao Liu, Stephen Wright, and Dimitris Papailiopoulos. 2018. ATOMO: Communication-efficient Learning via Atomic Sparsification. arXiv:1806.04090 [stat.ML]

[24] Wei Wen, Cong Xu, Feng Yan, Chunpeng Wu, Yandan Wang, Yiran Chen, and Hai Li. 2017. TernGrad: Ternary Gradients to Reduce Communication in Distributed Deep Learning. In *Advances in Neural Information Processing Systems 30*, I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett (Eds.). Curran Associates, Inc., 1509–1519.

[25] 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 *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. USENIX Association, Carlsbad, CA, 595–610.

[26] Peifeng Yu and Mosharaf Chowdhury. 2019. Salus: Fine-Grained GPU Sharing Primitives for Deep Learning Applications. arXiv:1902.04610 [cs.DC]

[27] Haoyu Zhang, Logan Stafman, Andrew Or, and Michael J. Freedman. 2017. SLAQ: Quality-Driven Scheduling for Distributed Machine Learning. In *Proceedings of the 2017 Symposium on Cloud Computing* (Santa Clara, California) *(SoCC '17)*. Association for Computing Machinery, New York, NY, USA, 390–404. https://doi.org/10.1145/3127479.3127490

7 APPENDIX

7.1 Convergence Analysis

We start by presenting the convergence theorem about SGD in Theorem 7.1 from [6]

Theorem 7.1. *Let X be convex compact set and $R^2 = sup_{x,x_1 \in X} ||x - x_1||^2$. If the approximate gradient $\tilde{g}(x)$ is such that $E||\nabla f(x) - \tilde{g}(x)||^2 \leq V^2$, after t iterations, SGD with step size $\frac{1}{\beta + \frac{1}{\eta(t)}}$, where $\eta(t) = \frac{R}{\sigma}\sqrt{\frac{2}{t}}$, achieves*

$$E(f(\frac{1}{t}\sum_{s=1}^{t} x_{s+1}) - f(x^*)) \leq R\sqrt{\frac{2V^2}{t}} + \frac{\beta R^2}{t} \tag{2}$$

The above result bounds the difference between $f(\hat{x})$ after t iterations and the optimal value $f(x^*)$ in terms of V^2, which is a bound on how far our estimated gradient is from the true gradient.

Distributed Mini-batch SGD. Now, we consider the scenario where we have k workers each processing a batch $\frac{B}{k}$ at every iteration where B is cumulative mini batch size. We first consider the scenario where every worker computes the entire gradient and attempt to bound the value of $E||\nabla f(x) - \tilde{g}(x)||^2$.

Lemma 7.2. *With k workers, an overall batch size of B and assuming $||g(x_i)|| < P \ \forall i$, for mini-batch SGD, we have $E\|\nabla f(X) - \tilde{g}(x)\|^2 \leq 2P^2\frac{k}{B}$*

Proof: We partition the parameter space X into subparts, x_i

$$\tilde{g}(x, \epsilon) = [...., \frac{1}{B}\sum_{j=1}^{B} g_i(x_i, \epsilon_j), ...] \tag{3}$$

For i^th block of parameters,

$$\begin{aligned} E\|\nabla f(x_i) - \tilde{g}(x_i, \epsilon)\|^2 &= E\|\nabla f(x_i) - \frac{1}{B}\sum_{j=1}^{B} g_j(x_i)\|^2 \\ &= \frac{1}{B}E\|\nabla f(x_i) - g_1(x_i)\|^2 \\ &= \frac{1}{B}p_i \end{aligned} \tag{4}$$

where we denote p_i as $E\|\nabla f(x_i) - g_1(x_i)\|^2$

For the overall parameter space this reduces to:

$$E\|\nabla f(x) - \tilde{g}(x, \epsilon)\|^2 = \sum_{i=1}^{k} \frac{1}{B}p_i \tag{5}$$

Under the assumption that $||g(x_i)|| < P \ \forall i$, we get

$$E\|\nabla f(x) - \tilde{g}(x, \epsilon)\|^2 \leq 2P^2\frac{k}{B} \tag{6}$$

We now obtain the same bound for SPB case.

SPB Convergence Rate We next consider the case when SPB is used to prune gradients. The main difference here is that i^{th} block of the gradient vector will be updated by i workers.

Lemma 7.3. *When using SPB with k workers and overall batch size of B and assuming $||g(x_i)|| < P \ \forall i$, we have $E\|\nabla f(X) - \tilde{g}(x)\|^2 \leq 2P^2\frac{k}{B}log(k)$*

In SPB setting, i^{th} block will be updated by i workers, leading to $\frac{iB}{k}$ mini batch, as each worker is processing $\frac{B}{k}$ mini batch.

$$\tilde{g_{SPB}}(x, \epsilon)) = [...., \frac{1}{\frac{iB}{k}}\sum_{j=1}^{\frac{iB}{k}} g_i(x_i, \epsilon_j), ...] \tag{7}$$

For the i^{th} block of parameters,

$$E\|\nabla f(x_i) - \tilde{g}(x_i, \epsilon)\|^2 = \frac{k}{iB}p_i \tag{8}$$

Similar to previous baseline case, for the overall parameter space we get,

$$\begin{aligned} E\|\nabla f(x) - \tilde{g}(x, \epsilon)\|^2 &= \sum_{i=1}^{k} \frac{k}{iB}p_i \\ &\leq P^2\frac{k}{B}\sum_{i=1}^{k}\frac{1}{i} \\ &\leq 2P^2\frac{k}{B}log(k) \end{aligned} \tag{9}$$

Substituting Equation (9) and Equation (8) in Theorem 7.1, we compare the convergence rate, which shows that SPB needs $log(k)$ times more iterations to achieve similar performance as the baseline setting where all workers are sending complete gradients. This bound suggests that on increasing the number of workers, logarithmically more iteration are going to be needed, which seems intuitive as the more the number of workers, the more fraction of data points that are not getting utilized in updating the parameters in comparison to baseline.