A model-based approach to streamlining distributed training for asynchronous SGD

Sung-Han Lin*‡, Marco Paolieri*, Cheng-Fu Chou†, Leana Golubchik*
*Department of Computer Science, University of Southern California – {*sunghan, paolieri, leana*}@usc.edu
†Department of Computer Science, National Taiwan University – ccf@csie.ntu.edu.tw
‡NetApp, Inc. – sunghan.lin@netapp.com

***Abstract*—The success of Deep Neural Networks (DNNs) has created significant interest in the development of software tools, hardware architectures, and cloud systems to meet the huge computational demand of their training jobs. A common approach to speeding up an individual job is to distribute training data and computation among multiple nodes, periodically exchanging intermediate results. In this paper, we address two important problems for the application of this strategy to large-scale clusters and multiple, heterogeneous jobs. First, we propose and validate a queueing model to estimate the throughput of a training job as a function of the number of nodes assigned to the job; this model targets asynchronous Stochastic Gradient Descent (SGD), a popular strategy for distributed training, and requires only data from quick, two-node profiling in addition to job characteristics (number of requested training epochs, mini-batch size, size of DNN parameters, assigned bandwidth). Throughput estimations are then used to explore several classes of scheduling heuristics to reduce response time in a scenario where heterogeneous jobs are continuously submitted to a large-scale cluster. These scheduling algorithms dynamically select which jobs to run and how many nodes to assign to each job, based on different trade-offs between service time reduction and efficiency (e.g., speedup per additional node). Heuristics are evaluated through extensive simulations of realistic DNN workloads, also investigating the effects of early termination, a common scenario for DNN training jobs.**

***Index Terms*—Distributed Machine Learning, Parallel Scheduling, Queueing Networks, TensorFlow.**

I. Introduction and Related Work

Machine learning is one of today's most rapidly growing fields in computer science. Its combination of artificial intelligence and data science has led to the development of practical technologies currently in use for many applications. Deep learning [1] is a novel area of machine learning that recently achieved breakthrough results in several domains, including computer vision, speech recognition, natural language processing, and robot control. Its distinctive trait is the use of Deep Neural Networks (DNNs) to discover, directly from input data, internal representations suitable for classification tasks, without the need for manual feature engineering.

To be effective, this approach requires very large amounts of data and computation. For example, DNNs for image classification include tens of layers and millions of *weights* (parameters that combine the outputs of one layer to produce inputs of the next one) that are tuned using datasets of millions of images [2]. Training can take weeks and must often be repeated for multiple values of hyperparameters of the training algorithm, such as the *learning rate* or *momentum* parameters of Stochastic Gradient Descent (SGD).

To speed up training and provide quick turnaround to users submitting DNN training jobs, it is important to leverage hardware acceleration (e.g., by using GPUs that implement DNN primitives) and *distributed training*, which uses multiple machines in parallel. Machine learning frameworks such as TensorFlow [3], Caffe2 [4], MXNet [5] provide high-level abstractions that allow the user to easily run training algorithms on GPUs with hardware acceleration and in parallel on multiple machines. Using a *parameter server architecture* [6], the dataset is split among many *worker nodes* that perform training in parallel, sending parameter updates to a parameter server and receiving the most recent version of parameters (which include updates from other workers).

Scheduling. As shown by experimental measurements [7], when more worker or server nodes are assigned to a job, its throughput (number of training examples processed per second) increases only *sublinearly*; in some cases, when a shared resource (e.g., the network) is congested, adding more nodes can *reduce* the throughput, thus increasing the overall job service time. In practice, users need to benchmark each training job with an increasing number of worker nodes to evaluate the corresponding throughput. Even when throughput is known for each number of assigned nodes, dynamic scheduling of heterogeneous jobs is an open research problem.

Several efforts based on linear programming or approximation algorithms tackled *offline formulations* with the goal of *minimizing the makespan* of a set of tasks while satisfying deadline requirements and precedence relations [8], [9], [10], [11]. High-Performance Computing (HPC) scheduling considers *rigid jobs* where the number of required nodes is fixed and specified by the user [12], [13]; this formulation reduces to 2D bin-packing (an NP-hard problem), for which HPC heuristics sort jobs according to many factors (age, size, priority) and schedule them in order, with backfilling of unused timeslots.

When jobs with generally-distributed size are *moldable* (i.e., the scheduler can select an arbitrary but fixed level of parallelism) or *malleable* (i.e., the scheduler can adapt the level of parallelism over time) and arrivals of new jobs are described by a Poisson process, optimal algorithms are known only for the case of *linear speedups* [14]; in particular, mean response time is minimized by assigning all nodes to the job that has the least expected work on a single node. This policy, which extends the approach of Shortest-Remaining-Processing-Time (SRPT) to parallelizable jobs, can be quite

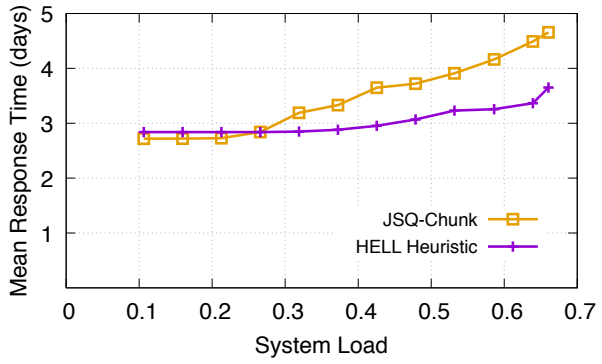


Fig. 1: Mean response time of AlexNet [17] training jobs on a cluster of 100 nodes using JSQ-Chunk scheduling [15] (with best chunk size) or HELL heuristic (Section IV)

inefficient when speedups are only sublinear, as is the case for DNN training jobs. A recent independent effort [15] considers the case of parallel scheduling of jobs with non-decreasing, concave speedups and *exponential job size distributions*: the proposed near-optimal policy, JSQ-Chunk, minimizes mean response time of *homogeneous* jobs (i.e., with the same mean service time and speedup function).

Fig. 1 (generated by our experiments detailed in Section V) illustrates that, when job sizes are *not exponentially* distributed (as in the case of DNN training jobs), better heuristics can be constructed, similarly to [16], by analyzing both job speedup properties and remaining amount of work (see details in Section IV-A). In this figure, the mean response time (including waiting and service time of jobs) is obtained from a simulation of AlexNet [17] training jobs, where the mean service time on a single node is $E[S(1)] = 12$ days, and the speedup from additional workers is almost linear for 2-4 workers, saturates gradually for 5-9 workers, and then converges to 4.8 due to network bottlenecks. The cluster includes $n = 100$ nodes and arrivals are generated as a Poisson process with varying rate λ to produce system loads $\lambda E[S(1)]/n$ from 0.1 to 0.66. This experiment illustrates that, in the case of DNN training jobs, the exponential assumption is problematic; this makes intuitive sense, as remaining service times with an exponential distribution (assumed in JSQ-Chunk) have the same distribution as the entire service duration, which is not the case for DNN training jobs. In this work, we consider a scenario where each job requires training of a DNN from an initial value of weights and for a given number of *epochs* (i.e., passes through the entire dataset of training examples): the user inspects loss and accuracy curves at the end of a job and decides whether to resubmit the model to train for an additional number of epochs. This interactive scenario is typical during the development and tuning of DNN models, where many hyperparameter values or model variants can quickly lead to unsatisfactory results.

Throughput Prediction. Resource management becomes even more difficult when training jobs are heterogeneous and the speedup from using multiple nodes is not known in advance. Existing approaches to throughput prediction of distributed machine learning jobs adopt either black-box models or very simple analytical ones.

A black-box model is used in [18] to fit linear parameters α_i of a service-time function of the form $S(n) = \alpha_0 + \alpha_1/n + \alpha_2 \log(n) + \alpha_3 n$, where n is the number of nodes and $S(n)$ the corresponding service time. The motivation for this approach is that, after splitting the work among n nodes, communication patterns usually introduce linear or logarithmic overhead. But each DNN training job is characterized by the different amount of data exchanged with the parameter server and by the computation times at worker and server nodes. A black-box approach would require profiling each job using multiple choices for the number of workers n to fit the parameters α_i. Instead, our goal is to profile each job using 1 worker node and to leverage a model of distributed training to estimate $S(n)$.

A simple analytical model for service-time speedup of *synchronous* SGD is presented in [19], where the computation load is split equally among workers and communication overhead is proportional to the logarithm of the number of nodes. In contrast, our experimental evaluation highlights that, in *asynchronous* SGD, communication overhead depends on the specific pattern of network access by the nodes: as illustrated by our measurements in Fig. 7 (details in Sect. III), the overhead of additional workers is sometimes negligible at low network utilization, as different workers exchange small amounts of data without transmission overlaps, using the entire available bandwidth (Fig. 8). In addition, while synchronous SGD repeats its communication pattern after each step, asynchronous SGD requires a dynamic model of system resources and of their use by nodes assigned to a training job. Analytical models for general machine learning jobs are proposed also in [20], but communication overhead is assumed to be negligible, which is not the case for real-world DNNs.

Contributions. This work provides two main contributions.

1) *Performance Model of Asynchronous SGD.* We tackle the problem of *throughput prediction* of a given DNN training job as a function of the number of asynchronous SGD workers. In Sect. III, we develop a queueing network model [21] where different stations model worker nodes, the parameter server, and its incoming and outgoing network links. For a given DNN training job, we use this model to estimate the number of examples processed per second with any number of workers; our model requires only *quick profiling with a single worker* to evaluate computation times and data exchanged with the parameter server. To account for the effects of TCP congestion control, we switch, depending on network load, between two models of communication. We validate this model in Sect. V-A using (i) synthetic DNNs on a cluster of CPU-only nodes and (ii) popular DNN models in a public cloud environment with GPU instances; results indicate that our model achieves mean error within 10% of experimental TensorFlow measurements.

2) *Parallel Job Scheduling.* We leverage our performance model to address the problem of scheduling heterogeneous DNN training jobs on a computing cluster. Each job requires the training of a DNN from an initial value of its weights

and for a given number of epochs. This is a common scenario during the development of DNN models, when many model variants and hyperparameter values are evaluated to find a combination giving the best accuracy; users start many jobs for a limited number of epochs, check on them, and decide whether to cancel them or to request additional training. The size of each job has a general distribution, which is determined by the number of training examples to process, the amount of computation required to process each example, and the size of the model (transmitted over the network between the server and the workers). The level of parallelism of a job is either selected when service begins (moldable case) or adapted over time (malleable case). Our proposed heuristics achieve different tradeoffs between system efficiency and speedup of job response time (i.e., time spent in the system by a job, waiting or in service). We also explore mechanisms to reduce the time for completing intermediate results (e.g., 50% of the job size). Speeding up completion of intermediate results is desirable because intermediate results can be used to cancel jobs that are not promising (e.g., training jobs with inappropriate values of hyperparameters). Several classes of heuristics are evaluated in Section V-B through extensive simulations. The results indicate that our heuristics for malleable jobs are stable at 70% load and achieve lower mean response times than simple SRPT-type approaches, which become unstable at much lower loads (10%-30%); for moldable jobs, heuristics are stable at 60% load and provide lower mean response time than JSQChunk.

II. Background on DNN Training

In this section, we give a brief overview of background on DNN training needed in the remainder of the paper.

A. Distributed Stochastic Gradient Descent

Stochastic Gradient Descent (SGD) [22] is the most widely used algorithm for DNN training. For a given set of layers, connections and activation functions, a DNN is a parametric function f computing $y = f(x;\theta)$ (e.g., an image classification) from the inputs x (e.g., the pixel values of each RGB channel), where the parameters $\theta = (\theta_1, \ldots, \theta_n)$ are real-valued *weights* connecting neurons of different layers. From a random initialization, θ is iteratively improved to minimize the error on a training set of labeled examples $\mathcal{D} = \{(x_i, y_i)\}$ as measured by a loss function L, i.e., to solve the problem $\min_\theta J(\theta)$ where $J(\theta) = \frac{1}{|\mathcal{D}|}\sum_{(x,y)\in\mathcal{D}} L(f(x;\theta), y)$. At each iteration t, the algorithm updates the weight vector with the rule $\theta^{(t+1)} = \theta^{(t)} - \eta g^{(t)}$, where $g^{(t)}$ is the gradient of the training error J on a *mini-batch* of examples $\mathcal{B} \subseteq \mathcal{D}$ and η is the learning rate. This step is repeated for several *epochs* (full iterations over $\mathcal{D}$), which can include millions of examples. To provide faster feedback and improve DNN models, clusters of distributed nodes are required. The *parameter server* [6], [23], [24] is a popular architecture to distribute the computation of SGD over multiple nodes. As depicted in Fig. 2, the training dataset is partitioned among multiple *worker nodes* that compute gradients in parallel,

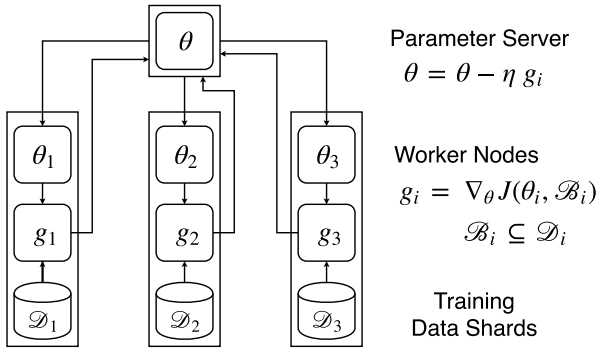


Fig. 2: Parameter Server Architecture

on separate mini-batches of examples (*data parallelism*). To synchronize their execution, worker nodes send gradients $g^{(t)}$ to a parameter server that holds the most up-to-date version of the weights θ. The parameter server applies the gradients and sends back the weights θ to the workers. In *asynchronous* SGD, weights are sent back to a worker immediately after applying its gradient; in *synchronous* SGD, the parameter server sends the weights only after receiving and applying gradients from all of the workers. In this paper, we focus on asynchronous SGD with many worker nodes and a single parameter server. Asynchronous SGD achieves better throughput than synchronous SGD, as it removes synchronization overhead. While synchronous SGD training on K workers with mini-batch size B is equivalent to single-node training with mini-batch size $K \times B$, asynchronous SGD is equivalent to single-node training steps with the same batch size but possibly stale model parameters. Recent results [25] highlight that, in contrast to previous evaluations [26], asynchrony and parameter staleness do not impair accuracy when the *momentum parameter* (used to compute an exponential moving average of mini-batch gradients) is appropriately tuned. Thus, we use the same mini-batch size (specified by the user) for any assignment of workers, and assume that momentum is adjusted appropriately for the configuration.

B. Machine Learning Frameworks and TensorFlow

Machine learning frameworks, including TensorFlow [3] and MXNet [5], allow users to define a DNN model and train it in parallel using asynchronous SGD on multiple machines. TensorFlow represents computations as a *dataflow graph* where nodes encode operations of a DNN, while intermediate results flow along edges as *tensors* (multidimensional arrays). The dataflow graph makes communication between subcomputations explicit and allows the framework to execute independent computations in parallel across multiple CPUs and GPUs, possibly on different nodes of a cluster. Specialized implementations of abstract operations allow the use of hardware acceleration, and the framework transparently handles transmission of data among devices, e.g., among nodes in a network or among GPUs using a parameter server architecture.

In addition, TensorFlow supports fault tolerance through user-level *checkpointing*: the chief node of the cluster peri-

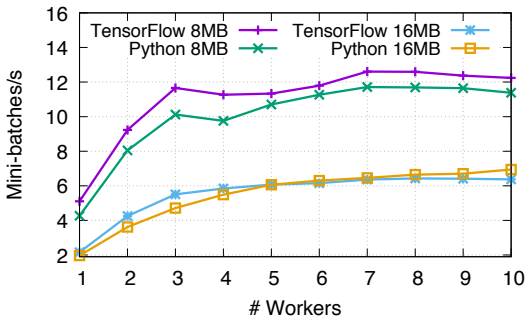


Fig. 3: Measured training throughput of TensorFlow and Python implementations (mini-batch size is 50 examples)

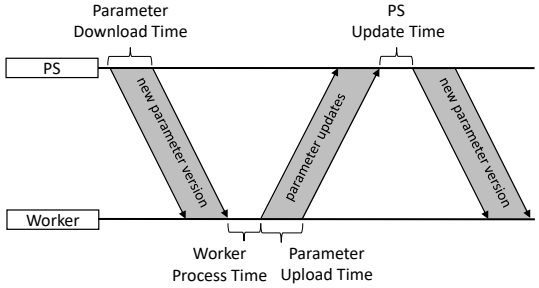


Fig. 4: Phases of asynchronous SGD with parameter server

odically saves the current version of θ to disk; when a client restarts, it automatically attempts to restore θ from the last checkpoint. Through checkpoints, it is possible for workers or parameter servers to recover from faults, or to suspend and resume training, effectively enabling *preemptive scheduling* with limited loss of completed work and minimal overhead.

In this paper, we validate predictions from our throughput model with measurements obtained by running TensorFlow in a distributed cluster. First, we build synthetic DNN models of varying sizes by changing the number of neurons in each layer. Given that we are not evaluating classification accuracy of DNNs, but only throughput, during training we sample random examples to match the input layer of these synthetic models. Then, we validate our model on real-world DNNs and datasets: Google's Inception [27], ResNet-152 [29], and VGG16 [28]. Note that asynchronous SGD can achieve learning accuracy comparable to synchronous training when the momentum parameter is adjusted [25]: this allows us to concentrate on the performance aspect of distributed training, assuming only minor effects on the final accuracy of trained DNNs. While our experimental evaluation uses TensorFlow, we highlight the general applicability of our throughput model in Section III-A.

III. Throughput Estimation

In this section, we propose a performance model of DNN training jobs to estimate, from the communication and computation times measured in a 1-server 1-worker cluster, the training throughput (examples processed per second) as a function of the number of assigned worker nodes.

A. Distributed SGD Measurements

To measure the throughput and understand the behavior of distributed TensorFlow, we built a testbed cluster including 11 servers, each equipped with two AMD Opteron 2376 CPUs (quad-core, 2.30 GHz) and 16 GB of RAM, connected by a 1 Gbps switch in full-duplex mode. Each server runs TensorFlow 1.0.1 on Debian 8 using Python 3. Although limited in size, this cluster is sufficient to hit networking bottlenecks during training and allows us to validate our model. (Later, in Section V-A, we validate our model on Google Compute Engine, using GPU nodes and popular DNNs.)

Fig. 3 reports measurements of training throughput in our cluster with up to 10 workers (as mini-batches per second, for mini-batches of 50 training examples), for DNN models with exchanged parameters size of 8 MB and 16 MB (respectively, a fully-connected $360\times1000\times1000\times1000\times200$ model and a convolutional model with 784 inputs, 3 convolutional layers of 32, 32, and 64 feature maps 3×3, a fully-connected layer of 256 units, and 10 outputs). Interestingly, training throughput highlights different trends: it saturates smoothly for the 16 MB model, but presents a non-monotonic trend for the 8 MB model. To confirm that our measurements are not due to specific implementation details of TensorFlow or to DNN parameters, we developed a client-server Python program mimicking the operations of TensorFlow. Instead of processing real data, our program makes `sleep` system calls for amounts of time equal to the processing times measured in TensorFlow; exchanged data has the same size. Measurements obtained with this client-server program, also reported in Fig. 3, are similar to TensorFlow measurements; this suggests that a performance model should account for the different processing times at each node, and for the interaction between network access and TCP connections. To better account for the interference among transmission sessions, we propose the use of a queueing model for throughput estimation.

The input to the queueing model (presented in Section III-B) includes 4 parameters evaluated through profiling with 1 server and 1 worker: downlink time S^D (weights transmission from server to worker), worker time S^W (gradient computation), uplink time S^U (gradient transmission from worker to server), server time S^S (gradient application). These parameters define the interaction pattern of a single worker, illustrated in Fig. 4; our model accounts for the interaction of *multiple* workers accessing shared resources (parameter server and network).

To measure these parameters in a 1-worker 1-server cluster running TensorFlow, we use `tcpdump` to collect TCP packets during profiling of the training procedure. This is the most reliable method, since TensorFlow uses the binary `protobuf` format to serialize data before transmission with the gRPC protocol over TCP; analysis of TCP packets allows us to avoid framework-dependent instrumentation, enabling the application of our approach to other machine learning frameworks.

For example, Fig. 5 shows the uplink and downlink bandwidth usage measured from packet transmissions collected by `tcpdump` while training the VGG16 DNN model on

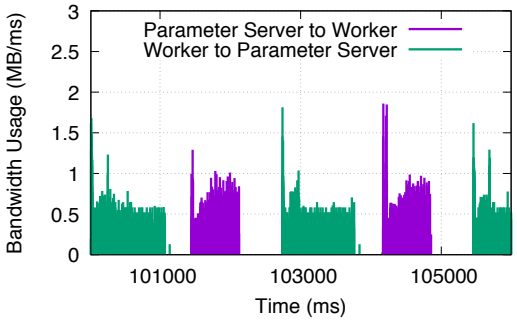


Fig. 5: `tcpdump` measurements for one parameter server and one worker training VGG16 in TensorFlow

Google Compute Engine using one parameter server and a single worker node equipped with GPU. We can see that communication between the parameter server and the worker follows the repetitive pattern abstracted in Fig. 4. We measure transmission times S^D and S^U directly, and estimate the processing times S^W and S^S from the time elapsed between the end of a transmission and the beginning of the next one. We average these measurements over up to 200 server-worker interactions (which require only a few minutes).

B. Queueing Model

We model asynchronous SGD training as the closed queueing system illustrated in Fig. 6. There are exactly K tasks, one for each worker of distributed SGD; a task models the processing of a mini-batch of examples at a worker, the transmission of a gradient to the parameter server, its application, and the transmission of up-to-date parameters back to the worker. Each task $k = 1, \ldots, K$ belongs to a different class (or *chain*) which determines its routing among the queues of the network: task k visits the kth worker node (an infinite server, IS, station), the uplink station (a processor sharing, PS, station), the parameter server (a first-come first-served, FCFS, station), and the downlink station (another PS station), before returning to the kth worker node.

Although the connection of each worker with the parameter server can cross many network switches, the model abstracts the network fabric as one non-blocking switch [30], and only focuses on its ingress and egress ports (e.g., NICs), while the PS policy approximates the behavior of packet-switching networks. The FCFS policy of the parameter server models the sequential application of gradients from different workers; after a task leaves this station, it goes to the downlink station, modeling weights waiting to be sent back to the worker.

Our goal is to estimate the throughput at the parameter server, i.e., the number of mini-batches processed per time unit in a training job, using all workers. To compute the throughput of our model, we use Mean Value Analysis (MVA), a recursive algorithm calculating equilibrium queue sizes, waiting times, and throughputs in product-form queueing networks [31]. The requirements for product-form queues impose exponential distributions for the service time at FCFS stations (in our case, the parameter server—unless a PS discipline is assumed), while other stations can have general service times.

TABLE I: Summary of notation

$1, \ldots, K$	Worker nodes / task classes in the system
$L(k) := \{k, U, S, D\}$	Stations visited by class k: kth worker node, uplink (U), parameter server (S), downlink (D)
S_k^l	Service time of class k at station l
$\vec{n} := (n_1, \ldots, n_K)$	Number of tasks in the system, for each class
$N_k^l(\vec{n})$	Mean number of tasks of class k at station l
$X_k^l(\vec{n})$	Mean throughput of class k at station l
$T_k^l(\vec{n})$	Mean response time of class k at station l
$\rho_k^l(\vec{n})$	Class k utilization at station l

Table I summarizes our notation. Let $\vec{n}^* = (n_1^*, \ldots, n_K^*)$ represent the populations of task classes $\{1, \ldots, K\}$: in our model, $n_k^* = 1$ for all k as each worker has exactly one task. MVA computes mean response times $T_k^l(\vec{n}^*)$ for all classes k and stations l incrementally, for each $\vec{n} = (n_1, \ldots, n_k)$ such that $n_1 = 0, \ldots, n_1^*$, $n_2 = 0, \ldots, n_2^*$, and so on, starting from $T_k^l(\vec{0}) = 0$ and using the identity (following from the Arrival Theorem [32]):

$$T_k^l(\vec{n}) = \begin{cases} S_k^l \left[1 + \sum_{j=1}^K N_j^l(\vec{n} - \vec{e}_k)\right] & \text{if } l \text{ is PS/FCFS} \\ S_k^l & \text{if } l \text{ is IS} \end{cases} \quad (1)$$

where e_k is a vector with k-th component equal to 1 and others equal to 0, and $\vec{n} - \vec{e}_k$ is the population vector $\vec{n}$ decreased by 1 in class k. In Eq. (1), $N_j^l(\vec{n} - \vec{e}_k)$ can be recursively computed from previous values $T_j^l(\vec{n} - \vec{e}_k)$ using Little's law $N_k^l(\vec{n}) = X_k^l(\vec{n})\, T_k^l(\vec{n})$ where $X_k^l(\vec{n}) = n_k / \sum_{l \in L(k)} T_k^l(\vec{n})$.

Then, the mean system throughput (as mini-batches/second) with K workers is given by $X(K) := \sum_{k=1}^K X_k^S(\vec{n}^*)$, i.e., the throughput of all classes at the parameter server S. Note that, while the rest of this paper considers the case where worker nodes are homogeneous (i.e., with the same bandwidth and computation times), this model can be applied to heterogeneous workers. Additional performance metrics can also be derived, such as mean class utilization $\rho_k^l(\vec{n}^*) = X_k^l(\vec{n}^*) S_k^l$ for all k, l.

Results for $X(K)$ are compared in Fig. 7 with throughput measurements obtained by running TensorFlow with a parameter server and $K = 1, \ldots, 10$ workers, for the DNN model of size 8 MB described in Section III-A. As parameters of our model, we use the results of the 1-worker profiling using `tcpdump`: $S^W = 29$ ms (processing at the worker), $S^S = 18$ ms (processing at the parameter server), $S^D \approx S^U = 72$ ms (downlink and uplink). We use these mean service times for all the classes of the queueing model, i.e., $S_k^W = S^W$, $S_k^S = S^S$, $S_k^D = S^D$, $S_k^U = S^U$ for all $k = 1, \ldots, K$.

As shown in Fig. 7, the real throughput of TensorFlow is higher than predicted by our model with PS uplink/downlink (yellow line) when there are 2 to 3 workers in the system (throughput scales almost linearly for 2 workers), while predictions are accurate with more than 3 workers. We address this phenomenon in the next section.

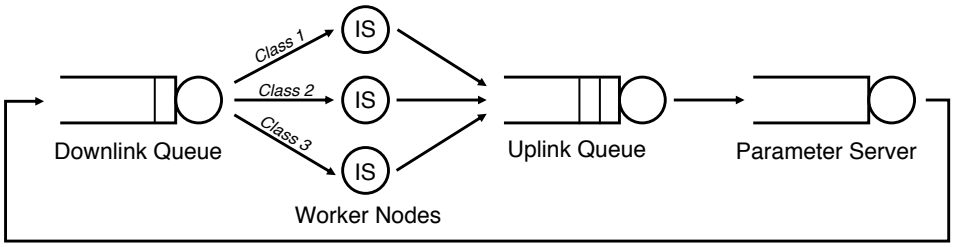


Fig. 6: Queueing model of a distributed machine learning application with a parameter server architecture

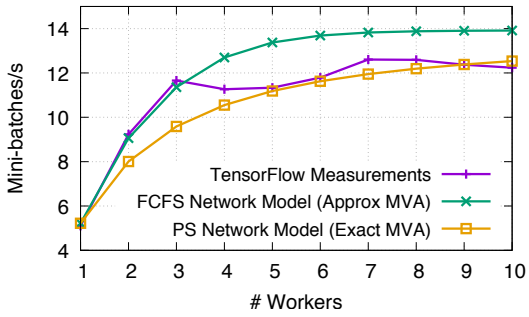


Fig. 7: Throughput with PS and FCFS uplink/downlink

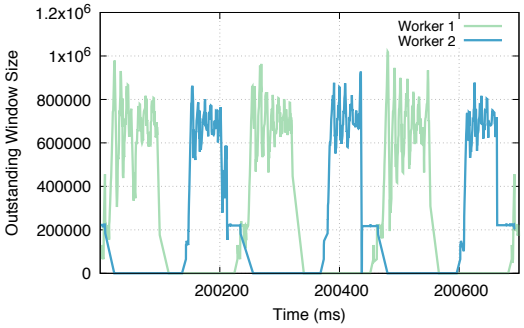


Fig. 8: Outstanding window size for two async workers

C. On the Effects of Short TCP Transmissions

To investigate the almost linear speedup of TensorFlow with 2 workers, we analyzed TCP packets exchanged while running TensorFlow. Fig. 8 illustrates the TCP outstanding window size (data sent but not yet acknowledged), which shows that the workers send data during different time slots, without competing for network bandwidth. This phenomenon is due to the fact that data sent from the workers to the parameter server, and vice versa, has similar size (proportional to the number of weights). If one worker can finish its uplink (or downlink) transmission before the other node, it has a higher chance of starting the next transmission first (after processing at the parameter server or worker); in case of transmission overlaps, TCP congestion control will favor the node that is already transmitting (transmission times are comparable to those required to adapt to bandwidth sharing), thus creating a self-reinforcing mechanism where asynchronous workers transmit at different times.

This phenomenon is also possible for more than two workers. As shown in Fig. 9, if worker computation times are long enough, workers can send updates to the parameter server without competing for network bandwidth. For instance, the distributed training job illustrated in Fig. 9 allows up to 5 workers to transmit their updates without overlaps.

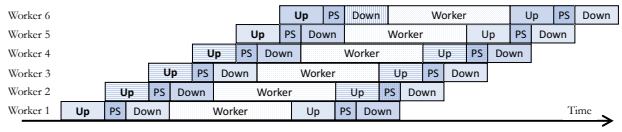


Fig. 9: Illustration of linear speedup in asynchronous SGD

When an ongoing TCP flow is strongly advantaged with respect to new flows, uplink and downlink stations can be modeled as FCFS queues with deterministic service times. Since exact MVA requires exponential service times in FCFS stations, we adopt the *approximate MVA* solution of [31, Eq. (3.29)], which gives downlink/uplink response times

$$T_k^l(\vec{n}) = S_k^l + \sum_{j=1}^{K} S_j^l \left[N_j^l(\vec{n} - \vec{e}_k) - \frac{1}{2}\rho_j^l(\vec{n} - \vec{e}_k) \right] \qquad (2)$$

for $l = D, U$, where $\rho_k^l(\vec{n}) = X_k^l(\vec{n}) S_k^l$ is the utilization of station l by class k. With this FCFS network model, the predicted throughput $X(K)$, illustrated in Fig. 7 (green line), shows good accuracy for less than 4 workers, but overestimates the measured throughput when $K \geq 4$.

In fact, as the number of workers increases, network bandwidth is shared among workers. To model increasing transmission overlaps, we combine Eqs. (1) and (2) to obtain

$$T_k^l(\vec{n}) = \gamma^l \, T_k^{l,\mathrm{PS}}(\vec{n}) + (1 - \gamma^l) \, T_k^{l,\mathrm{FCFS}}(\vec{n}) \qquad (3)$$

where $\gamma^l := g(\rho_k^l(\vec{n}))$ is a function of the utilization of station $l \in \{D, U\}$, $T_k^{l,\mathrm{PS}}(\vec{n})$ is defined as in Eq. (1), $T_k^{l,\mathrm{FCFS}}(\vec{n})$ is defined as in Eq. (2). In this model, which we call *Hybrid MVA*, γ^l should be close to 1 under heavy load (network shared equally) and to 0 under light load (exclusive network access). For predictions in our CPU-only cluster (Fig. 10), we use:

$$g(\rho) = \begin{cases} \frac{\rho - 0.8}{0.2} & \text{if } \rho \geq 0.8, \\ 0 & \text{otherwise.} \end{cases} \qquad (4)$$

Therefore, our throughput estimation considers overlapping transmissions only when the link utilization is higher than 0.8. This model is specific to our environment; for other environments (e.g., large-scale data centers), an appropriate form of g must be determined through preliminary profiling.

IV. Scheduling Mechanisms

The performance model presented in Section III allows us to estimate the throughput of a training job when a different number of nodes is assigned by a scheduler. But, even with exact knowledge of the throughput function, no optimal algorithm is known for response time minimization of a Poisson stream of jobs with general size distribution [14].

In this section, we describe heuristics for this problem when preemption is allowed and jobs are moldable or malleable. In Section VI, we consider extensions that allow users to check intermediate results and decide whether to terminate a job early. This can be of particular benefit for DNN training jobs, where users submit jobs with different combinations of hyperparameters (e.g., learning rate, number of layers, neurons in each layer), monitor their training progress, and terminate jobs corresponding to non-promising models.

Problem Definition. Jobs arrive to the system as a Poisson stream with fixed rate λ; for each job, the scheduler knows the job size (total number of training examples to process, i.e., number of epochs specified by the user, multiplied by the number of training examples per epoch) and the *throughput function* $X_i(w)$, which gives the throughput (training examples processed per second) of job i using w nodes ($w-1$ workers when $w > 1$, as one node is used for the parameter server). The cluster has W homogeneous nodes and each node can be assigned to at most one job at a time. If M jobs with remaining sizes $J_1^t, ..., J_M^t$ are present at time t, the remaining service time of job i with w_i^t nodes is $S_i^t(w_i^t) = J_i^t / X_i(w_i^t)$. Our goal is to determine the proper node allocation $w_i^t \geq 0$ for all i, t so as to minimize the mean response time, i.e., the mean time spent in the system by a job (waiting or in service).

A. Motivation for Parallel Scheduling Heuristics

It is natural to consider extensions of the Shortest-Remaining-Processing-Time (SRPT) policy (which is optimal in a single-node setting), e.g., allocating all nodes to the job that can achieve the shortest service time; this policy minimizes the time to the next job departure, and it was shown to minimize mean response time when job throughput increases *linearly* with the number of workers [14]. However, an SRPT-type policy is not optimal when speedups from additional workers are *sublinear* or *non-monotonic*, such as those presented in Section III for DNN training jobs. In this case, assigning all nodes to a single job can result in an *inefficient use of resources*, since (beyond a certain point) additional workers produce only minor service time reduction.

To gain insight into our scheduling heuristics (described below), consider the following. Our goal is to minimize mean response time by assigning an appropriate number of nodes to each job. A job's response time consists of its service time (with the allocated resources) and its waiting time (i.e., time waiting for resource availability). Intuitively, as the number of nodes allocated to each job increases, the service times of individual jobs (likely) decrease, but the waiting times of individual jobs (likely) increase (i.e., with larger allocation of nodes per job, it is more likely that an arriving job will find all nodes busy upon arrival, as the resources are more likely to be used in a less efficient manner).

Consider the two extremes of (1) allocating all nodes to one job (for instance, the job with the shortest remaining service time) and (2) allocating one node per job. Option (1) decreases the service time but potentially increases the waiting time, while option (2) increases the service time but potentially decreases the waiting time. How significant the effect is on the waiting time (of either option) largely depends on the system utilization: under light loads, the effect is not as significant, while under heavy loads it is. Thus, the heuristics we develop next are motivated by (a) efficient use of resources (so as to decrease the service time but with meaningful returns on additional resources allocated) while (b) incorporating the tradeoff (with corresponding effect on waiting time) through the explicit consideration of system utilization.

KNEE mechanism. As already noted, assigning all nodes to one job can waste resources while achieving minor throughput improvements. In the *KNEE mechanism*, we stop assigning nodes to job i when the next relative speedup

$$\delta_i(w) := \frac{S_i(w) - S_i(w-1)}{S_i(w-1)}$$

becomes lower than a fixed threshold α, i.e.,

$$w_i^{KNEE} = \min\{w \leq W : \delta_i(w) \geq \alpha \text{ and } \delta_i(w+1) < \alpha\}$$

where W is the budget of available nodes.

High Efficiency, Low Latency (HELL) mechanism. Our second heuristic jointly considers service time and efficiency of each assignment of nodes to a job. The policy adopts the "speedup per assigned node" [16] as an efficiency metric:

$$\textit{Speedup } s_i(w) := \frac{S_i(1)}{S_i(w)} \text{ and } \textit{Efficiency } E_i(w) := \frac{s_i(w)}{w}.$$

For each job i, we select the number of nodes that minimizes the ratio between service time and efficiency:

$$w_i^{HELL} = \underset{w \leq W}{\operatorname{argmin}} \left\{ \frac{S_i(w)}{E_i(w)} \right\}.$$

Since this heuristic suggests an allocation that tries to achieve short response time (favoring jobs with short service time) with high efficiency, we refer to it as *High Efficiency, Low Latency (HELL) mechanism*. With respect to the KNEE heuristic, HELL always considers efficiency; in contrast, for small α, KNEE tends to allocate additional nodes even when throughput improvements are limited. Later, we provide an approach for KNEE to adapt α to system load.

In the following sections, we evaluate the impact of KNEE and HELL on the scheduling of malleable and moldable jobs.

B. Malleable Job Scheduling

In the malleable job scheduling case, the scheduler can dynamically update the number of nodes allocated to a job during execution time. This functionality is supported by TensorFlow through checkpointing: the parameter server periodically saves

ALGORITHM 1: Malleable KNEE Job Scheduling

Input: Number of nodes W and jobs M, remaining service times $S_1^t(w), ..., S_M^t(w)$ for $w \leq W$, knee threshold $\alpha < 0$.
Output: Node allocations $w_1^t, ..., w_M^t$.

$\mathcal{A} \leftarrow \{1, \ldots, M\}$ *// jobs to schedule*
$w_i^t \leftarrow 0$ for all $i = 1, \ldots, M$ *// default node assignment*
while $W > 0$ **and** $|\mathcal{A}| > 0$ **do**
  for all $i \in \mathcal{A}$ **do**
    $w_i^{KNEE} \leftarrow \min \{w \leq W : \delta_i(w) \geq \alpha \wedge \delta_i(w+1) < \alpha\}$
      where $\delta_i(w) := [S_i^t(w) - S_i^t(w-1)]/S_i^t(w-1)$
  $j \leftarrow \text{argmin}_{i \in \mathcal{A}} S_i^t(w_i^{KNEE})$ *// shortest knee service*
  $w_j^t \leftarrow w_j^{KNEE}$
  $W \leftarrow W - w_j^{KNEE}$
  $\mathcal{A} \leftarrow \mathcal{A} \setminus \{j\}$
end

ALGORITHM 2: Malleable HELL Job Scheduling

Input: Number of nodes W and jobs M, remaining service times $S_1^t(w), ..., S_M^t(w)$ for $w \leq W$.
Output: Node allocations $w_1^t, ..., w_M^t$.

$\mathcal{A} \leftarrow \{1, \ldots, M\}$ *// jobs to schedule*
$w_i^t \leftarrow 0$ for all $i = 1, \ldots, M$ *// default node assignment*
while $W > 0$ **and** $|\mathcal{A}| > 0$ **do**
  for all $i \in \mathcal{A}$ **do**
    $w_i^{HELL} \leftarrow \text{argmin}_{w \leq W} \{S_i^t(w)/E_i^t(w)\}$
      where $E_i^t(w) := S_i^t(1)/[w\, S_i^t(w)]$ *// efficiency*
  $j \leftarrow \text{argmin}_{i \in \mathcal{A}} S_i^t(w_i^{HELL})/E_i^t(w_i^{HELL})$
  $w_j^t \leftarrow w_j^{HELL}$
  $W = W - w_j^{HELL}$
  $\mathcal{A} = \mathcal{A} \setminus \{j\}$
end

the weights to disk and restores them when model training is restarted. Thus, we can reassign resources at arrivals or departures, when the number of jobs in the system changes.

Since reassignment of nodes is allowed, greedy allocations can be advantageous for scheduling mechanisms. This inspires our malleable KNEE and HELL job scheduling heuristics.

Malleable KNEE Job Scheduling. In this heuristic, detailed in Algorithm 1, the scheduler repeatedly selects the job j that has the shortest remaining time $S_j^t(w_j^{KNEE})$, i.e., using node assignments before the knee threshold α or before exhausting the available workers W. The same step is repeated until all remaining nodes are assigned or all jobs are scheduled for execution. Note that this algorithm may not assign all available nodes, especially if the knee point is reached with few nodes.

Malleable HELL Job Scheduling. In this heuristic, detailed in Algorithm 2, the scheduler selects the job j that has the minimum ratio $S_j(w)/E_j(w)$ for a number of nodes w_j^{HELL} less than or equal to the available ones W. The same step is repeated until all remaining nodes are assigned or all jobs are scheduled for execution. Note that this algorithm can also terminate before assigning all available nodes.

Since these heuristics can leave nodes idle (especially when the number of jobs is much lower than the number of nodes), we adopt a *filling mechanism* to assign the remaining nodes. In this mechanism, the minimum number of idle nodes is assigned to the job that can achieve the shortest remaining time with the use of such additional nodes; formally, if $\overline{W}$ nodes are still available at the end of Algorithm 1 or 2, we assign

$$\overline{w} = \min_{1 \leq i \leq M} \{w \leq \overline{W} \mid S_i^t(w_i^t + w) < S_k^t(w_k^t) \text{ for all } k\} \quad (5)$$

additional nodes to the job i achieving the minimum in Eq. (5). This procedure is repeated until all nodes are assigned or no job can achieve minimum service time with additional nodes.

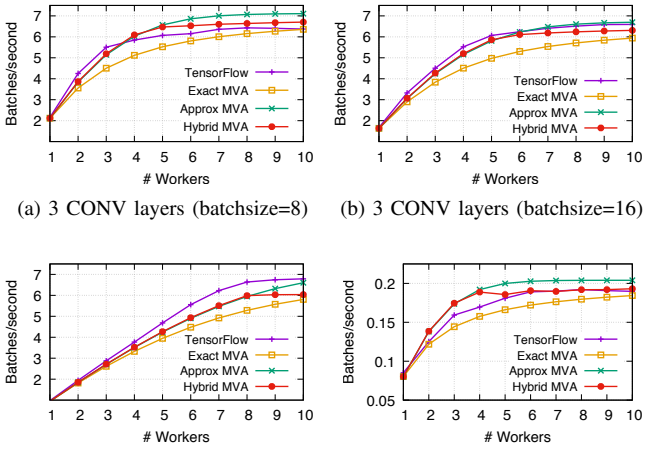
C. Moldable Job Scheduling

Malleable job scheduling (presented in Section IV-B) is more flexible in utilizing system resources, since their allocation can be modified after job arrivals or departures. However, if the number of workers allocated to a DNN training job changes, the complexity of repartitioning and redistributing the job may be significant. In this section, we consider the case of moldable job scheduling, where the level of parallelism (number of workers) of a job does not change after the beginning of its execution (but jobs can still be preempted).

In this case, poor initial allocation choices can have significant consequences, because the job has to continue using the same number of workers even when the system has idle nodes. One strategy is to let a job wait for an ideal number of nodes instead of starting earlier with fewer nodes. However, if the ideal number is quite large, the job may have to wait for a long time before starting service, or block other jobs from executing once it begins, even when the reduction in service time is marginal. On the other hand, if the ideal number of nodes is small, the job can be executed immediately, but it will likely run for a long time, due to reduced parallelism. Thus, we evaluate two heuristics derived from the malleable HELL and KNEE mechanisms.

Moldable HELL Job Scheduling. Similarly to Algorithm 2, this mechanism evaluates node allocations based on the metric $S_i^t(w)/E_i^t(w)$; however, the metric is now evaluated for all $w \leq W$, where W is the *total* number of nodes in the cluster, to find the optimal node assignment w_i^{HELL} for job i. If less than w_i^{HELL} nodes are available (after assigning nodes to jobs with higher metric), the moldable HELL mechanism waits before scheduling job i, since the number of allocated nodes cannot be modified later. In contrast with our malleable heuristics, moldable HELL scheduling is not *work-conserving*, i.e., it can leave computing power unused.

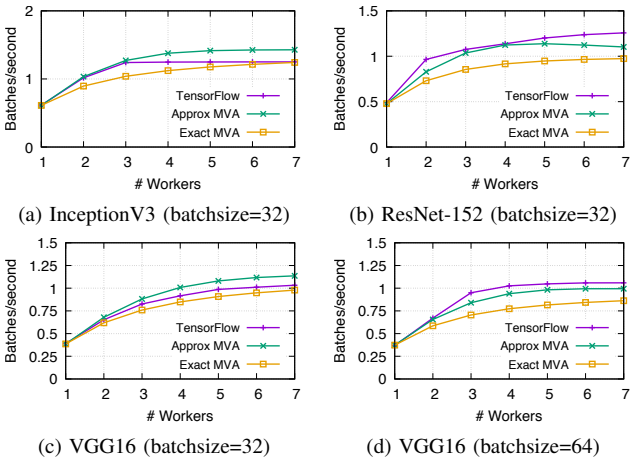
Moldable KNEE Job Scheduling. As described earlier, the number of nodes allocated by the KNEE mechanism depends on the value of α: with smaller α, the job will have larger w_i^{KNEE}. Since the level of parallelism of the job cannot be changed, the scheduler should use a smaller value of α when the system load is low (in order to utilize more resources) and a larger value of α when the system load is high (in order to be more efficient). Thus, we use the system load $\lambda E[S(1)]/n$ as a knob to control how many nodes should be allocated to



(a) 3 CONV layers (batchsize=8) (b) 3 CONV layers (batchsize=16)

(c) 3 CONV layers (batchsize=32) (d) InceptionV3 (batchsize=32)

Fig. 10: Throughput estimation in CPU cluster



(a) InceptionV3 (batchsize=32) (b) ResNet-152 (batchsize=32)

(c) VGG16 (batchsize=32) (d) VGG16 (batchsize=64)

Fig. 11: Throughput estimation in the cloud GPU cluster

TABLE II: Summary of throughput estimation error

DNN	MVA	Error		
		Max	Mean	Min
3 CONV layers (batchsize=8)	Exact	18%	8%	0.24%
	Hybrid	9%	5%	2%
3 CONV layers (batchsize=16)	Exact	18%	13%	3%
	Hybrid	8%	4%	2%
3 CONV layers (batchsize=32)	Exact	21%	14%	3%
	Hybrid	12%	8%	3%
InceptionV3	Exact	9%	6%	3%
	Hybrid	11%	4%	0.05%

a job. For an acceptable range $[\alpha_L, \alpha_H]$, the value of α used for scheduling at load x is $\alpha(x) = \alpha_L + x(\alpha_H - \alpha_L)$.

As in Algorithm 1, this heuristic allocates nodes based on the value of α. However, unlike Algorithm 1, where job i can receive less than w_i^{KNEE} nodes, our moldable KNEE mechanism starts job i only when w_i^{KNEE} nodes become available. This heuristic will not allocate nodes to job i if there are not enough nodes to meet its ideal assignment w_i^{KNEE}; thus, it is not work-conserving.

V. Evaluation And Validation

In this section, we first validate the accuracy of our throughput prediction model on different workloads by comparing predictions with measurements collected in a TensorFlow cluster. We then investigate the performance of the proposed scheduling heuristics based on throughput predictions. All simulations in this section were run until the 95% confidence intervals were smaller than 5% of the estimated values.

A. Throughput Estimation Validation

We evaluate the accuracy of our throughput estimation model under different settings: first, using synthetic DNN models and InceptionV3 on our own cluster of CPU-only nodes; then, using popular DNN models for image recognition and GPU instances in a public cloud environment.

CPU Cluster. First, we validate our asynchronous SGD model on a synthetic 16 MB DNN model with three convolutional layers and two fully connected layers, using our own cluster (described in Section III-A.) For this DNN model, we use batch sizes of 8, 16, and 32 examples/update, and illustrate the results in Figs. 10a to 10c, respectively. From these figures, we see that our Hybrid MVA model can accurately estimate throughput and speedups, which allow the HELL heuristic to identify worker assignments with high efficiency. Similarly, although the model underestimates throughput for batch size of 32, the knee of throughput curves can be selected accurately, allowing the KNEE heuristic to make node assignments close to those using exact throughput values.

Next, we validate our throughput estimation models by running Google's InceptionV3 DNN [27] in TensorFlow using CPU-only nodes. Results, reported in Fig. 10d, show that our Hybrid MVA model slightly overestimates the throughput with 2 to 4 workers; with more than 4 workers, there is almost no difference between our estimate and the measurements. This inaccuracy is likely due to the fact that the Inception model has more dynamic update patterns, while our synthetic model results in repetitive update patterns. However, throughput predictions are sufficiently accurate to allow the scheduler to select efficient resource allocations also in this case.

Moreover, as illustrated in Table II, our throughput estimation based on hybrid MVA outperforms the estimation based on exact MVA. This is largely due to the fact that its throughput model accounts for the synchronization between traffic generated by the workers, particularly when the time spent processing at a worker is greater than the time spent on the network due to traffic (i.e., in cases (a), (b), and (c)). When transmission times are substantially higher than the processing time at a worker (i.e., in case (d)), the synchronization effects are quite small, resulting in higher error rates. However, this should not be the typical case in DNN training applications, as users will likely increase batch sizes in such situations,

to increase the worker processing time (thus reducing the overhead of communication).

Cloud GPU Cluster. To validate our throughput estimation model on more complex DNNs and running environments, we train popular DNN models for image recognition using GPU-enabled instances on Google Compute Engine. In particular, we consider InceptionV3 [27], ResNet-152 [29], and VGG16 [28], with batch size of 32 examples (and also 64 for VGG16). Model sizes are, approximatively, 100 MB for InceptionV3, 232 MB for Resnet-152, and 528 MB for VGG16. We measure throughput of each model by running the TensorFlow benchmarks [33] with the ImageNet data set [34]. Each experiment is repeated by allocating 1 to 8 instances of Google Compute Engine, each with 4 vCPUs, 15 GB of RAM, and one K80 GPU. Each instance runs TensorFlow 1.6.0 on Ubuntu 16.04 LTS with CUDA 9.0 and cuDNN 7.0: out of the allocated instances, one acts as parameter server, while the others are used as worker nodes. Each worker node has exactly 1 GPU and communicates through TCP (over Ethernet) with the parameter server. The available bandwidth of Google Compute Engine (8 Gbps) is limited using Linux `tc` to 1 Gbps for InceptionV3, 3 Gbps for ResNet-152, 5 Gbps for VGG16.

Results, reported in Fig. 11, illustrate a close match between the throughput predicted by our models and that measured in TensorFlow. For InceptionV3 (Fig. 11a), a PS networking model (Exact MVA) provides better predictions at high loads, while FCFS networking (Approximate MVA) is more accurate at low loads. FCFS networking is also more accurate for ResNet-152 and for VGG16 (Figs. 11b to 11d). The non-monotonic trend of Approximate MVA in Fig. 11b is due to the error associated with this solution method: in fact, simulation of the queueing model indicates a monotonic increase.

B. Scheduling Evaluation

Next, we perform experiments to investigate the performance, and specifically, mean response time, of our scheduling heuristics. In contrast with the validation of throughput estimation, which uses our experimental testbed, we use a simulator to evaluate the benefits of scheduling mechanisms in a larger cluster. We simulate a cluster with 100 physical nodes, each with 1 GPU and a 1 Gbps Ethernet interface. (In this environment, we assume that physical nodes are interconnected by a high-speed top-of-rack switching architecture, so that the transmission bottleneck of each node is the bandwidth and buffer size of its interface.) To demonstrate that our mechanism is suitable for general DNN training jobs, we consider the workloads of four widely-used DNN models, listed in Table III. To estimate the throughput of each DNN model, we use the workload characteristics (model size, TFLOPS per batch, dataset size, training epochs) reported in the existing literature [35]. We use the performance of a specific GPU as a reference to calculate the processing time at each worker as well as at the parameter server. We assume that each node is equipped with one NVIDIA Grid K520 GPU, which provides 1229 GFLOPS [36]. To generate our synthetic workload, job inter-arrival times are sampled from an exponential distribution with rate λ and a DNN architecture is drawn uniformly at random from the four in Table III (NiN, GoogLeNet, AlexNet, VGG19) for the new job. Then, the model size of the DNN is used to estimate uplink and downlink transmission times on a 1 Gbps network; worker processing time is estimated for mini-batches of 1024 examples using the corresponding TFLOPS/batch; and the total job size (number of SGD steps) is selected by sampling the number of training epochs from a Gaussian distribution with mean reported in Table III and variance equal to 2.

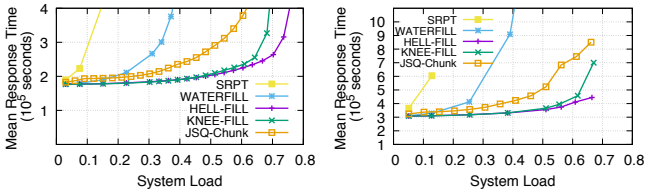
Performance Baselines. We use the two basic mechanisms described in Section IV-A as our baseline heuristics, one focusing on reducing the processing time (i.e., allocating all nodes to one job) and the other focusing on reducing the waiting time (i.e., executing as many jobs simultaneously as possible). For the strategy of allocating all nodes to a job, we use an SRPT-type job scheduling mechanism as a representative policy. For the strategy of executing as many jobs as possible, we adopt a modified version of the iterative mechanism proposed in [39], which assigns a minimum number of nodes to each job and then uses a "filling mechanism" to improve resource utilization (*Waterfill mechanism*). We also compare with JSQ-Chunk [15], where nodes are split into chunks of equal size and incoming jobs are assigned to the chunk with shortest queue (we repeat simulations to select the optimal chunk size, ranging from 7 to 40 in our experiments).

Performance Metrics. The primary performance metric in our evaluation is the mean response time of jobs, and we compare mean response time of each algorithm under the same system load. To compare different algorithms fairly, we define *system load* as $\lambda E[S(1)]/W$ where λ is the arrival rate, $E[S(1)]$ is the expected service time using only one node, and W is the number of nodes in the system. Note that, for the same value of system load, different algorithms will result in different system utilization due to the differences in the number of nodes that they allocate to each job.

Scheduling Results. We first evaluate malleable KNEE and HELL mechanisms, as illustrated in Fig. 12. Fig. 12a shows the results with jobs drawn uniformly at random from all four DNN architectures, and Fig. 12b illustrates the results with jobs only from larger DNNs (AlexNet and VGG19). SRPT produces the worst performance since it wastes resources for small marginal gains; Waterfill also performs poorly, since it runs too many jobs simultaneously and does not favor the completion of short jobs. The KNEE and HELL heuristics achieve similar performance when the system load is low, largely due to the benefits of the filling mechanism which allows the HELL heuristic to allocate a similar number of nodes per job as the KNEE heuristic. When the system load is high, the KNEE heuristic performs worse than the HELL heuristic. Examination of our simulation traces confirms that this is due to KNEE allocating more nodes to a job than HELL, resulting in fewer jobs being executed simultaneously. The more aggressive allocation of nodes to jobs by KNEE is due to the use of small values of α (0.01).

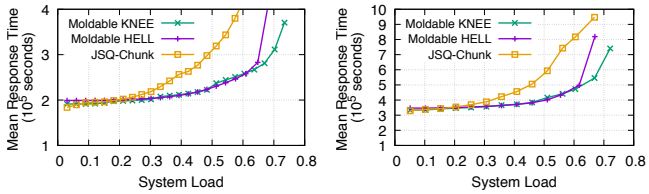
TABLE III: Synthetic workload collected by [35] from previous literature

Name	Total Examples $\|\mathcal{D}\|$	Model Size	Forward+Backward TFLOPS/batch of 1024	Epochs
NiN [37]	50,000 (CIFAR-10, CIFAR-100)	30 MB	6.7	200
GoogLeNet [2]	1.2×10^6 (ILSVRC 2014)	54 MB	9.7	200
AlexNet [38]	1.2×10^6 (ILSVRC 2012)	249 MB	7.0	90
VGG19 [28]	1.2×10^6 (ILSVRC 2012, ILSVRC 2014)	575 MB	120	74



(a) Workload with four DNNs (b) Workload with two large DNNs

Fig. 12: The performance of malleable job scheduling



(a) Workload with four DNNs (b) Workload with two large DNNs

Fig. 13: The performance of moldable job scheduling

We note that both policies perform better than JSQ-Chunk with multiple speedup functions [15]. In our experiments, the HELL mechanism with filling performs better when the system load is high for all types of training jobs.

Next, we compare the KNEE and HELL mechanisms for moldable scheduling. We do not include the two baseline algorithms, as they were already shown to produce worse allocations on the (easier) problem of malleable scheduling. Fig. 13a shows the results with jobs drawn uniformly at random from all four DNN architectures, and Fig. 13b illustrates the results with jobs only from larger DNNs (AlexNet and VGG19). As shown in both figures, moldable HELL and KNEE perform similarly when the system utilization is low, but the KNEE mechanism outperforms the HELL mechanism at higher loads. This difference is due to the fact that the moldable KNEE mechanism is able to execute more jobs simultaneously by allocating fewer nodes to each job at higher loads (due to the load-dependent α), while the HELL mechanism waits for the same, ideal number of nodes for each job (irrespective of the load). Both heuristics perform better than JSQ-Chunk with multiple speedup functions.

Summary. In order to reduce the response time, it is important to allocate a proper number of workers to each job. If this number can be modified during execution, the HELL mechanism with filling achieves better response time under all situations (in on our experiments). However, the complexity of changing the degree of parallelism may not be negligible. Thus, we considered the corresponding moldable scheduling mechanisms, and showed that the KNEE mechanism has greater flexibility to adjust its allocation under different system loads with respect to response time, as illustrated in Fig. 13.

VI. Extension to Early Termination

As noted earlier, one special property of DNN training jobs is that users submit many jobs with different hyperparameter settings, monitor their progress, and then terminate jobs (corresponding to less promising combinations) before the end of their execution. In this section we explore one possible direction to account for early termination during scheduling.

A. Scheduling Heuristic with Early Termination

To reduce mean response time in this scenario, one approach is to account for the current job progress during scheduling. We define the current *progress* of job i at time t as $p_i^t = 1 - \frac{J_i^t}{J_i}$, where J_i is the initial job size and J_i^t is the remaining job size at time t. A newly-submitted job has progress $p_i^t = 0$; intuitively, it should have higher priority over long-running jobs as it is more likely to be terminated by the user. On the other hand, jobs with short remaining time should also have high priority. To combine both criteria, we propose to use, during scheduling, service times adjusted as follows:

$$S_i^t(w) = \frac{J_i^t}{X(w)} \left(\frac{1}{1 - p_i^t} \right)^\gamma$$

where w is the number of workers and $\gamma \geq 0$.

Using a large value of γ gives higher priority to new jobs, but penalizes those that completed a large fraction, resulting in (potentially) higher mean response time. The optimal choice of γ depends on the behavior of users in the cluster. Accounting for this behavior is the subject of ongoing research.

B. Evaluation of Early Termination Heuristic

Fig. 14 depicts the Cumulative Distribution Function (CDF) of response time under the malleable HELL mechanism when different values of γ are used. As shown in the figure, our extension speeds up the early fraction of a job when $\gamma > 1$. For instance, without the extension, the mean time to complete 40% of the job requires around 41 hours, while our extension with $\gamma = 1.05$ results in only ≈ 26.8 hours, which reduces the time by 35%. However, this type of improvement comes at the cost of increasing the mean response time for completing the entire job. Thus, benefits depend on the type of jobs running in the system. If users are interested in exploring a greater number of hyper-parameter values and models, and less sensitive to the time required to obtain final results, the

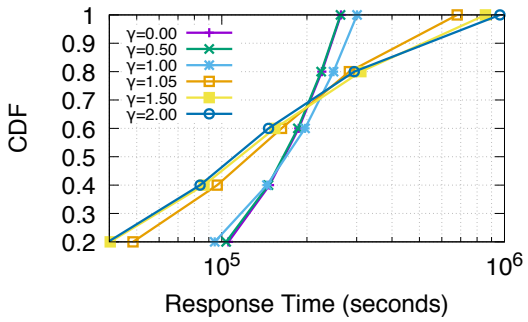


Fig. 14: Job response time CDF for early termination

scheduler can use a large value of γ. Otherwise, a small γ (e.g., between 1 and 1.05 in our experiment) is preferable.

VII. Extension of Distributed SGD Models

In Section III, we addressed the problem of throughput prediction for asynchronous SGD clusters where, after processing each mini-batch of training examples, worker nodes exchange gradients and up-to-date parameters with a single server node, using TCP over Ethernet. While these settings cover an important class of asynchronous SGD approaches, many variants are relevant for scheduling of large-scale DNN training workloads.

We believe that one of the most important is the use of *multiple parameter servers*: as evident from our measurements, very large DNN models (such as ResNet or VGG) quickly saturate the receiving interface of the parameter server when many asynchronous workers transmit their gradients. The extension of our queueing model to multiple parameter servers is not trivial, as each worker processes the next mini-batch only after receiving up-to-date DNN parameters from all server nodes, thus introducing a synchronization point. In Fig. 15 we illustrate a simple extension of our models to multiple parameter servers: the solutions of Eq. (1) and Eq. (2) are evaluated for reduced uplink, downlink, and parameter server times; in particular, when m parameter servers are assigned to a job, the uplink service time is modeled as S_k^U/m, the downlink service time is modeled as S_k^D/m, and the service time of the parameter server is modeled as S_k^S/m. This model assumes a uniform split of the parameters among server nodes, and synchronous server completions at different server nodes.

Validating the model in the same GPU-enabled cloud environment, we observe encouraging results with $m = 2$ servers for InceptionV3 and ResNet-152, while, for VGG16 and for all configurations with $m = 3$ parameter servers, our models severely overestimate the throughout measured using TensorFlow. Analyzing `tcpdump` logs of each parameter server, we observe an unbalanced split of VGG16 parameters among parameter servers (for $m = 3$, one of the servers holds 4 times more parameters). This anomaly, due to the greedy parameter allocation algorithm used by TensorFlow, causes a performance bottleneck in the system and a lower-than-expected throughput in Figs. 15e and 15f. To tackle these situations, we plan to extend our profiling procedure by instantiating server processes (without running actual DNN training) for varying values of m, in order to analyze the exact split of the DNN weights among multiple parameter servers.

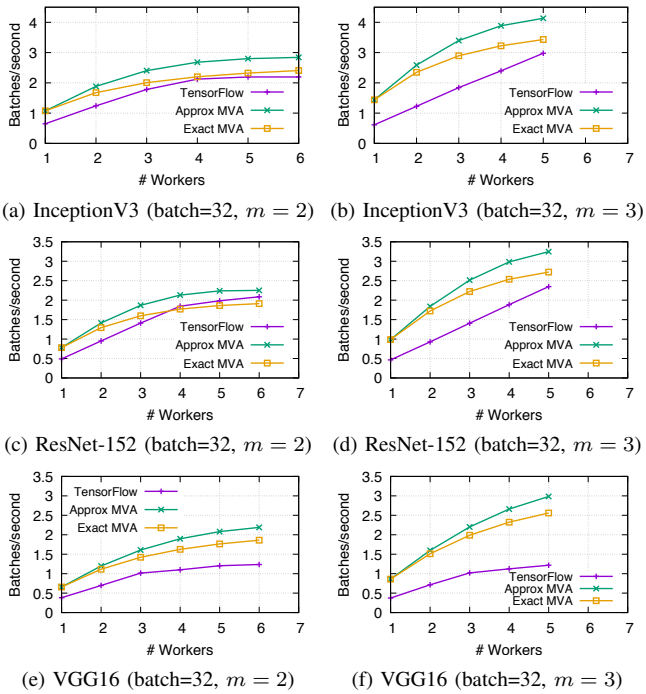
Given the implementation-dependent nature of these phenomena, we plan to validate our models on other machine learning frameworks, such as MXNet. Future work also includes modeling and validation of asynchronous SGD when GPUs are connected by dedicated Infiniband links (instead of TCP over Ethernet) or located on the same node and communicating over PCIExpress.

VIII. Conclusions

We focused on reducing the mean response time of machine learning jobs in a shared distributed computing environment. To this end, we developed a performance model for estimating the throughput of a distributed training job as a function of the number of workers allocated to it. Based on the throughput estimation, we proposed and evaluated scheduling heuristics that utilize resources efficiently in order to reduce the mean job response time. We also considered an extension for providing quick feedback to users, to facilitate early termination of jobs corresponding to non-promising DNN models, as early job termination is a desirable feature for machine learning training applications.

Acknowledgements

This work was supported in part by the NSF CCF-1763747 award. The authors would like to thank anonymous referees and Hardik Surana for useful feedback that helped improve the final version of this paper.



(a) InceptionV3 (batch=32, $m = 2$) (b) InceptionV3 (batch=32, $m = 3$)

(c) ResNet-152 (batch=32, $m = 2$) (d) ResNet-152 (batch=32, $m = 3$)

(e) VGG16 (batch=32, $m = 2$) (f) VGG16 (batch=32, $m = 3$)

Fig. 15: Throughput estimation in cloud GPU cluster with m parameters servers and varying number of workers

REFERENCES

[1] Y. LeCun, Y. Bengio, and G. Hinton, "Deep learning," *Nature*, vol. 521, no. 7553, pp. 436–444, May 2015.

[2] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich, "Going deeper with convolutions," in *CVPR'15*, 2015, pp. 1–9.

[3] M. Abadi and al., "TensorFlow: Large-scale machine learning on heterogeneous systems," 2015, software available from tensorflow.org.

[4] Caffe2: A New Lightweight, Modular, and Scalable Deep Learning Framework. [Online]. Available: https://caffe2.ai

[5] T. Chen, M. Li, Y. Li, M. Lin, N. Wang, M. Wang, T. Xiao, B. Xu, C. Zhang, and Z. Zhang, "Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems," *CoRR*, vol. abs/1512.01274, 2015. [Online]. Available: http://arxiv.org/abs/1512.01274

[6] J. Dean, G. Corrado, R. Monga, K. Chen, M. Devin, M. Mao, A. Senior, P. Tucker, K. Yang, Q. V. Le *et al.*, "Large scale distributed deep networks," in *Advances in neural information processing systems*, 2012.

[7] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard *et al.*, "Tensorflow: A system for large-scale machine learning." in *OSDI*, vol. 16, 2016, pp. 265–283.

[8] K. Jansen and L. Porkolab, "Computing optimal preemptive schedules for parallel tasks: linear programming approaches," *Mathematical programming*, vol. 95, no. 3, pp. 617–630, 2003.

[9] N.-Q. Nguyen, F. Yalaoui, L. Amodeo, H. Chehade, and P. Toggenburger, "Solving a malleable jobs scheduling problem to minimize total weighted completion times by mixed integer linear programming models," in *ACIIDS'16*. Springer, 2016, pp. 286–295.

[10] S. Chretien, J.-M. Nicod, L. Philippe, V. Rehn-Sonigo, and L. Toch, "Job scheduling using successive linear programming approximations of a sparse model." in *Euro-Par*, vol. 12. Springer, 2012, pp. 116–127.

[11] J. Błażewicz, M. Machowiak, J. Weglarz, M. Y. Kovalyov, and D. Trystram, "Scheduling malleable tasks on parallel processors to minimize the makespan," *Annals of Operations Research*, 2004.

[12] D. G. Feitelson, "Job scheduling in multiprogrammed parallel systems," 1997.

[13] J. Du and J. Y.-T. Leung, "Complexity of scheduling parallel task systems," *SIAM Journal on Discrete Mathematics*, 1989.

[14] K. C. Sevcik, "Application scheduling and processor allocation in multiprogrammed parallel processing systems," *Performance Evaluation*, vol. 19, no. 2-3, pp. 107–140, 1994.

[15] B. Berg, J. L. Dorsman, and M. Harchol-Balter, "Towards optimality in parallel scheduling," *CoRR*, vol. abs/1707.07097, 2017.

[16] D. L. Eager, J. Zahorjan, and E. D. Lazowska, "Speedup versus efficiency in parallel systems," *IEEE Transactions on Computers*, vol. 38, no. 3, pp. 408–423, 1989.

[17] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "ImageNet Classification with Deep Convolutional Neural Networks," in *NIPS Proceedings*, 2012, pp. 1106–1114.

[18] S. Venkataraman, Z. Yang, M. J. Franklin, B. Recht, and I. Stoica, "Ernest: Efficient performance prediction for large-scale advanced analytics." in *NSDI*, 2016, pp. 363–378.

[19] A. Ulanov, A. Simanovsky, and M. Marwah, "Modeling scalability of distributed machine learning," in *ICDE'17*. IEEE, 2017, pp. 1249–1254.

[20] H. Zhang, L. Stafman, A. Or, and M. J. Freedman, "SLAQ: quality-driven scheduling for distributed machine learning," in *SoCC Proceedings*, 2017, pp. 390–404.

[21] L. Kleinrock, *Queueing Systems*. Wiley, 1975.

[22] L. Bottou, "Stochastic gradient descent tricks," in *Neural Networks: Tricks of the Trade - Second Edition*, 2012, pp. 421–436.

[23] M. Li, D. G. Andersen, J. W. Park, A. J. Smola, A. Ahmed, V. Josifovski, J. Long, E. J. Shekita, and B.-Y. Su, "Scaling distributed machine learning with the parameter server." in *OSDI*, 2014.

[24] T. M. Chilimbi, Y. Suzue, J. Apacible, and K. Kalyanaraman, "Project adam: Building an efficient and scalable deep learning training system." in *OSDI*, vol. 14, 2014, pp. 571–582.

[25] I. Mitliagkas, C. Zhang, S. Hadjis, and C. Ré, "Asynchrony begets momentum, with an application to deep learning," in *54th Annual Allerton Conference on Communication, Control, and Computing*, 2016, pp. 997–1004.

[26] J. Chen, R. Monga, S. Bengio, and R. Jozefowicz, "Revisiting distributed synchronous sgd," *arXiv preprint arXiv:1604.00981*, 2016.

[27] C. Szegedy, V. Vanhoucke, S. Ioffe, J. Shlens, and Z. Wojna, "Rethinking the inception architecture for computer vision," in *CVPR'16*, 2016.

[28] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," *arXiv preprint arXiv:1409.1556*, 2014.

[29] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *CVPR'16*, 2016, pp. 770–778.

[30] M. Alizadeh, S. Yang, M. Sharif, S. Katti, N. McKeown, B. Prabhakar, and S. Shenker, "pfabric: Minimal near-optimal datacenter transport," in *ACM SIGCOMM Comp. Comm. Review*, vol. 43, no. 4, 2013.

[31] M. Reiser, "A Queueing Network Analysis of Computer Communication Networks with Window Flow Control," *IEEE Transactions on Communications*, vol. 27, no. 8, pp. 1199–1209, Aug 1979.

[32] M. Reiser and S. S. Lavenberg, "Mean-value analysis of closed multichain queuing networks," *Journal of the ACM (JACM)*, 1980.

[33] tf_cnn_benchmarks: High performance benchmarks. [Online]. Available: https://www.tensorflow.org/performance/benchmarks

[34] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei, "ImageNet: A Large-Scale Hierarchical Image Database," in *CVPR09*, 2009.

[35] F. N. Iandola, M. W. Moskewicz, K. Ashraf, and K. Keutzer, "Firecaffe: near-linear acceleration of deep neural network training on compute clusters," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2016, pp. 2592–2600.

[36] S. Hadjis, C. Zhang, I. Mitliagkas, D. Iter, and C. Ré, "Omnivore: An optimizer for multi-device deep learning on cpus and gpus," *arXiv preprint arXiv:1606.04487*, 2016.

[37] M. Lin, Q. Chen, and S. Yan, "Network in network," *arXiv preprint arXiv:1312.4400*, 2013.

[38] A. Krizhevsky, "One weird trick for parallelizing convolutional neural networks," *arXiv preprint arXiv:1404.5997*, 2014.

[39] G. Sabin, M. Lang, and P. Sadayappan, "Moldable parallel job scheduling using job efficiency: An iterative approach," in *Workshop on Job Scheduling Strategies for Parallel Processing*. Springer, 2006.