RESOURCE ELASTICITY IN DISTRIBUTED DEEP LEARNING

Andrew Or [1] **Haoyu Zhang** [1 2] **Michael J. Freedman** [1]

ABSTRACT

Elasticity—scaling out or in depending upon resource demand or availability—allows a system to improve its efficiency or performance. This leads to potentially significant cost savings and shorter job completion times. However, elasticity is not possible in today's distributed deep learning deployments, in large part because the most widely used frameworks such as TensorFlow are built assuming that resource allocation must be fixed throughout the lifetime of the job.

In this work, we demonstrate that these assumptions are not fundamental to distributed deep learning and present the first autoscaling engine for these workloads. Our system takes into account cost as well as scaling efficiency when making scaling decisions. As a side benefit, we reuse the same autoscaling mechanisms to remove persistent stragglers. We evaluate our system by training ResNet on CIFAR-10 and ImageNet, and we find a reduction in job completion time of up to 45% and a reduction in GPU time of up to 85.1%.

1 INTRODUCTION

Distributed learning today often over- or under-provisions resources. The state-of-the-art approach of assigning resources to a job is largely manual: an expert estimates what is an appropriate set of resources for a given job based on past experiences with similar jobs. For example, for training ResNet (He et al., 2016) on ImageNet (Deng et al., 2009), it is common to choose a number of GPUs such that the batch size per GPU is 64 or 128 images (Jia et al., 2018; Ying et al., 2018; Sun et al., 2019). For workloads that are not well known, these guidelines do not exist and users have to resort to trial-and-error to find an appropriate set of resources for their jobs.

However, a trial-and-error approach is expensive. Every iteration destroys all processes along with their in-memory program state, such as preprocessed input samples and the computation graph, and this can take minutes to rebuild. Further, deciding how much resources to assign to a given job requires knowing the scaling characteristics of the job. This is difficult to estimate ahead of time without sufficient experience with the job itself or similar jobs.

For these reasons, users often rely on suboptimal but fixed sets of resources for their jobs. This means their jobs are potentially either running more slowly than what the users can afford (in terms of paying for extra resources), or wasting resources that do not actually contribute to observable progress. Further, even if the amount of resources assigned to a job is optimal, the *set* of resources may not be, for example, when there are persistent stragglers in the system (see Figure 1). Existing approaches handle stragglers by overprovisioning resources, but never actually remove the stragglers (Abadi et al., 2016; Karakus et al., 2017).

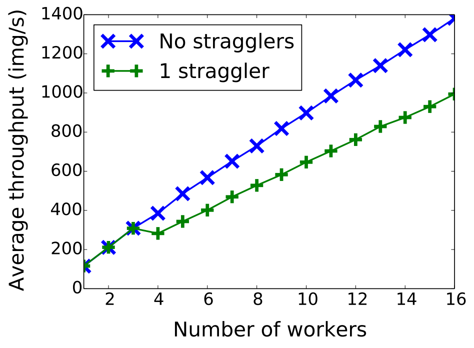


Figure 1. Even a single straggler can persistently hurt throughput scaling in synchronous SGD training. After the 4th worker, which is 33% slower than the other workers on average, the difference between the two curves widens as more workers are added.

1.1 Resource elasticity

Contrary to existing practice, this paper argues that resource allocation in distributed learning should be elastic. A job should request more resources if throughput feedback is positive, and relinquish resources if training progress would not be affected by the loss of these resources.

[1]Department of Compute Science, Princeton University, Princeton, NJ, USA [2]Google AI, Mountain View, CA, USA. Correspondence to: Andrew Or <andrewor@cs.princeton.edu>.

Proceedings of the 3^{rd} MLSys Conference, Austin, TX, USA, 2020. Copyright 2020 by the author(s).

In cloud environments, the system can incrementally find the most cost efficient set of resources for a job, while ensuring that the user is not paying for extra resources that do not contribute to training progress due to scaling limits. In multi-tenant environments, the system can share resources more efficiently with other jobs running in the cluster and take advantage of diurnal cycles. In both cases, straggler mitigation comes for free; with the appropriate statistics, we can simply replace workers that are consistently slow.

The idea of resource elasticity has already been widely explored in many other areas such as cloud services (AWS), distributed batch processing (Dean & Ghemawat, 2008; Zaharia et al., 2012), stream processing (Gedik et al., 2013), and cluster management (Rensin, 2015). Applying it in the context of distributed learning can reap similar benefits.

1.2 Main challenges

One of the main reasons why many users still rely on the manual trial-and-error process today is because the notion of static resource allocation is baked into state-of-the-art distributed learning systems. TensorFlow (Abadi et al., 2016), a widely popular deep learning framework, embeds a job's cluster membership information into a computation graph that cannot be easily modified once training begins. Synchronization between workers relies on mechanisms that wait for a fixed number of tokens or messages that corresponds to the number of workers in the system.

Other popular frameworks such as PyTorch (Paszke et al., 2017) and MXNet (Chen et al., 2015) have similar constraints. Although PyTorch supports dynamic graphs, they are only dynamic with respect to inputs and operations, not with respect to the number of processes in the system. The lack of native support for dynamic resource allocation in these frameworks poses a significant barrier for users who wish to utilize their resources more efficiently.

Realizing resource elasticity for distributed learning is more than just an engineering effort, however. There are fundamental differences between distributed learning and other areas, such as distributed batch processing, that already support it.

First, because these existing areas are largely driven by dynamic workloads, resource utilization is often used as a metric for scaling. In contrast, resource demands in distributed learning are largely static throughout the lifetime of the job; workers are often highly utilized and never idle. This means we need to design our own scaling heuristics instead of just reuse existing ones.

Second, in distributed learning, scaling out generally involves increasing the overall batch size across all the workers in the system, but this affects the convergence of the model (Keskar et al., 2016). Our goal is to provide flexible resource allocations without affecting the quality of training.

1.3 Autoscaling engine for distributed learning

To realize these goals, we built an autoscaling engine on top of TensorFlow, using Horovod (Sergeev & Del Balso, 2018) as the underlying communication mechanism. This architecture decouples computation from synchronization across workers in the system. During changes in resource allocation, our system reuses existing processes and saves all relevant program state in memory to minimize idle time. In addition to building such a system, this paper makes the following contributions:

- Outline the main architectural limitations in state-of-the-art distributed learning systems that prevent resource elasticity.
- Design new scaling heuristics for distributed learning that take both throughput and cost into account.
- Present the first distributed learning system that handles persistent stragglers without overprovisioning resources.

This work is primarily focused on synchronous stochastic gradient descent (SGD) training using the allreduce parameter update architecture (Patarasuk & Yuan, 2009). However, the techniques presented also apply to other forms of distributed learning. We do not focus on the multi-tenant use case in depth, as the problems there are already partially solved by SLAQ (Zhang et al., 2017b), a cluster scheduler that gives more resources to jobs that converge faster. Rather, the goal of this paper is to take the first step in bringing attention to the need for resource elasticity in distributed learning.

2 Background

In this section, we elaborate on the main hurdles for resource elasticity in distributed deep learning systems today. We begin by discussing the design limitations in TensorFlow that makes autoscaling difficult (§2.1), then turn to the scaling complications posed by the batch size and discuss how these complications affect autoscaling (§2.2).

2.1 Architectural limitations in TensorFlow

The common practice for training deep learning models with TensorFlow involves replicating the model graph across workers in the system. Model parameters are mirrored across the workers and synchronized at the end of each step. Operations in the model graph that cross device boundaries are connected via special `Send` and `Receive` operations that TensorFlow adds to the graph.

Today, there is no easy way to change cluster membership at runtime. This would require replacing existing `Send` and `Receive` operations with new ones that properly reconnect the new set of workers. However, this is non-trivial because once training begins, the framework has already performed a series of optimizations such as rewriting the graphs, building the input data pipeline, and potentially JIT compiling for accelerators (XLA).

Further, synchronization primitives in TensorFlow rest on the assumption that the number of processes is fixed throughout the lifetime of the program. For example, `SyncReplicasOptimizer` uses fixed-sized queues to wait on all workers at the end of each step. The newer `MultiWorkerMirroredStrategy` API, which is the recommended method for distribution in TensorFlow 2.0, makes similar assumptions. Model parameters created under this strategy's scope are automatically mirrored across the specific replicas that were configured in the beginning. Expanding the cluster requires rebuilding the model graph under a new distribution strategy for the new set of workers.

Similar limitations also exist in other widely used frameworks. Thus, to build an autoscaling system for distributed learning workloads, we must either resolve these architectural constraints or find ways to work around them. We will return to this in more detail in §6.

2.2 How to set the batch size?

Depending on the context, the term batch size can refer to either the number of samples assigned to each device, or the total number of samples across all devices. In this paper, we refer to the former as the *local* batch size and the latter as the *global* batch size, and use the term *minibatch* to refer to the global batch size's worth of data processed in each step.

In homogeneous systems, the global batch size is simply the local batch size multiplied by the number of devices in the system. Therefore, finding the right number of devices for a given job is tantamount to finding the right values for these batch sizes. Today, this process is often driven by the constraints on both batch sizes.

Constraints on the local batch size. Performance of accelerators such as GPUs and TPUs rely on processing large batches of data for better parallelism. However, device memory is limited, and all input data and the intermediate computation results in a minibatch must fit in memory. This sets a hard upper limit for the local batch size. For example, for training ResNet-50 on ImageNet, an NVIDIA V100 GPU can fit up to 170 samples at float32 precision. Although there is no lower limit, resource utilization deterioriates significantly as the local batch size decreases.

Constraints on the global batch size. Recent work has shown that using large global batch sizes can affect the convergence of the model (Keskar et al., 2016; Smith & Le, 2017; Hoffer et al., 2017), and the effects of this have been observed at scale (Goyal et al., 2017; Jia et al., 2018; Mikami et al., 2018; Sun et al., 2019). For common workloads, there are often well known thresholds for the global batch size before training begins to diverge. For example, this threshold is 8192 for training ResNet on ImageNet, assuming no additional optimization techniques are applied. However, finding this threshold for arbitrary workloads is an open problem that we will not address in this paper.

Thus, finding an efficient amount of resources for a given job involves searching within these constraints. The search space can be large, however. Even after fixing the global batch size for a given workload, there are still many configurations to explore. For example, how much less efficient is 256x32 (number of GPUs times local batch size) vs 64x128? Is the tradeoff worth the extra cost? These questions are difficult to answer without sufficient experience with the job, and they are common for users deploying their models even without elasticity.

Implications for scaling. How should the batch sizes change when new workers are added to the system? There are two ways of handling this. The first fixes the local batch size, which preserves the per device efficiency but allows the global batch size to climb, and this may affect convergence (weak scaling). The second fixes the global batch size, which decouples convergence from scaling, but sacrifices per device throughput since the local batch size must fall as the cluster expands (strong scaling).

Our system takes the middle ground: we accept a maximum global batch size from the user and fix the local batch size until this threshold is crossed. If this limit is not provided, the system will assume that either the model convergence will not be significantly affected under large global batch sizes, or the user considers the potential degradation in model quality acceptable.

3 System Overview

In a typical distributed learning setup, minibatches are set to be small enough such that the system as a whole can exploit good parallelism. Thus, the time to process each minibatch is generally short (subsecond to seconds) and runtime statistics available at the end of each minibatch can be accessed frequently.

The autoscaling engine has two major components: scaling heuristics (§4) and straggler detection (§5) (Figure 2). It collects these runtime statistics to build a distribution of throughputs for each individual worker and for each cluster size. With these distributions, the engine can make fine-grained scaling decisions at the minibatch granularity. In practice, it is more reasonable to make a decision every

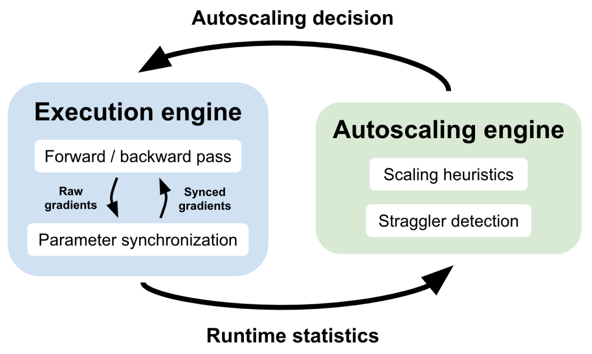


Figure 2. Autoscaling engine collects runtime statistics to make fine-grained autoscaling decisions.

n (e.g., 10) minibatches to ensure that we have sufficient numbers of data points.

Once a scaling decision has been made, the system needs to only update the parameter synchronization portion of the execution engine. Other in-memory state used for computing gradients, such as preprocessed samples, model parameters, and computation graphs, can be reused as is. This allows the system to minimize idle time when responding to changes in resource allocations.

4 Scaling Heuristics

An autoscaling system relies on scaling heuristics to determine when to add or remove how many workers. Scaling heuristics consists of two components: a scaling schedule that decides when to add or remove how many workers, and a set of scaling conditions that decides whether it is still efficient to keep scaling. We first discuss the former while assuming the existence of the latter.

4.1 Scaling schedule

Our system uses a simple schedule that adds K workers every N batches until the scaling conditions are violated. The unit of scaling K depends on the workload and system constraints; for example, a reasonable value for K is the number of GPUs on a machine, where each worker is assigned a single GPU. The scaling interval N controls how much information the system is given before making an autoscaling decision. A large N means decisions are more reliable because more runtime statistics are collected, but delays finding a more efficient cluster size. In practice, even a small value such as $N = 10$ is often sufficient.

Figure 3 shows how the schedule adds and removes workers using the scaling conditions. The system always starts with only one data point on the throughput curve, which is not enough to make any decisions. Therefore, the first step is always to add K workers (or remove K workers if the

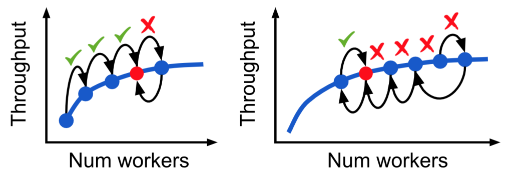


Figure 3. Scaling schedule for scaling out (left) and scaling in (right). Autoscaling decisions follow the direction of the arrows. The check mark or red X between two points represents whether scaling conditions were met when scaling from the first point to the second. The red dot represents the final cluster size the schedule settled on.

cluster size is already at the upper limit). If the scaling conditions passed, then we continue adding until this is no longer true (left figure).

If the scaling conditions failed, then it means it is not efficient to expand the cluster. We must keep searching, however, because the most efficient point may be smaller than our current size. Thus, when this happens, the system will jump to next lowest point that it does not have runtime statistics for, as shown in the right figure of Figure 3. This process continues until the scaling conditions begin to pass.

4.2 Scaling conditions

Scaling conditions refer to the conditions for which a worker should be added to the system. In general, the system will try to keep the number of workers at the point before the conditions first begin to fail.

4.2.1 *Throughput scaling efficiency*

The first metric our system considers is incremental throughput scaling efficiency, defined as follows:

$$s_{k,d} = \frac{\Delta R / \Delta k}{r_k}$$

$$\begin{aligned}
k &= \text{current number of workers} \\
d &= \text{number of workers to add} = \Delta k \\
R_k &= \text{aggregate throughput with k workers} \\
r_k &= \text{per worker throughput with k workers} = R_k / k \\
\Delta R &= R_{k+d} - R_k
\end{aligned}$$

Then, the scaling condition is simply whether this term is above a certain threshold:

$$s_{k,d} > S, S \in [0, 1]$$

Intuitively, this term measures how much each extra worker contributes to the overall performance of the system relative

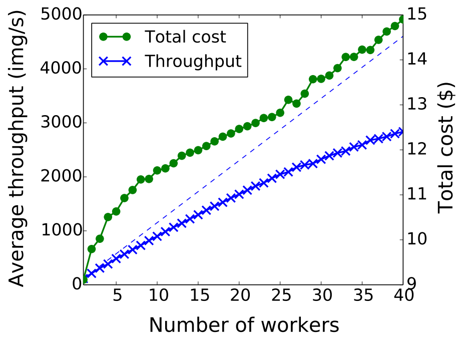


Figure 4. An example of how total cost and throughput scale with cluster size. Dotted line represents perfect scaling for throughput.

to how much each existing worker contributes. For example, in the simple case when $d = 1$, this expression becomes

$$s_{k,1} = \frac{R_{k+1} - R_k}{r_k}$$

Suppose $R_4 = 400$, then $R_5 = 500$ means the new worker contributes the same amount as each existing worker and the scaling efficiency is 1 (perfect scaling), and $R_5 = 450$ means the scaling efficiency is only 0.5. A negative $s_{k,d}$ means the aggregate throughput actually *dropped* as a result of adding d workers. This can happen if the increase in communication costs outweigh the benefits of increased parallelism, for instance.

Note that unlike normal definitions of scaling efficiency, $s_{k,d}$ is relative to the previous number of workers instead of relative to a single worker. This is because we often do not have the data point for running with a single worker, as these jobs are distributed.

4.2.2 *Utility vs cost*

To many users, metrics like dollar cost and job completion time are more intuitive to reason about than scaling efficiency. If the user can provide an estimate of the utility gained from a given job in terms of dollar amount, then the system can use this information to find a cluster size that maximizes utility while minimizing cost.

Total cost. We define the expected total cost at any given point of the job as the following:

$$C = C_p + t_k * b * k * P$$

$$\begin{aligned} C_p &= \text{past cost, the amount spent on this job so far} \\ k &= \text{current number of workers} \\ t_k &= \text{step time, a function of number of workers} \\ b &= \text{number of remaining batches} \\ P &= \text{price of an instance per time unit, a constant} \end{aligned}$$

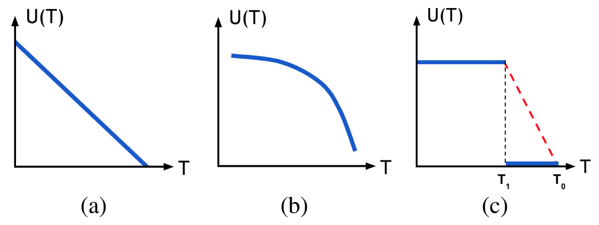


Figure 5. Example utility functions provided by the user. Y-axis is utility in terms of absolute dollars and x-axis is total training time. Utility functions are monotonically decreasing. In cases when jumps are present, such as in step functions, the system modifies the functions to replace the gaps with linear functions, as shown by the red dotted line in 5(c).

Figure 4 plots the total cost and throughput for a job training ResNet on CIFAR-10 on a CPU cluster. The machines used are most similar to m5.4xlarge instances on EC2, which have 16 cores and 64GB memory each and cost \$0.768 per hour when this paper was written. Thus, we set $P = 0.768/3600$ since we express step time in terms of seconds.

Finding the number of workers that generates the most value depends on (1) the total cost C of running the job with that many workers, as shown above, (2) the total completion time T of the job, and (3) how much completing a job within T is worth in dollar amount, denoted as utility $U(T)$.

Utility function. $U(T)$ is a monotonically decreasing function; a rational user should always prefer shorter completion times. Figure 5 lists common forms of $U(T)$. A linearly decreasing function indicates a constant preference for lowering T regardless of the value of T. An concave downward function indicates a higher preference to lowering T when T is high, but this preference diminishes as T decreases. A step function can be used to express a target completion time for the job.

Given a utility function and a complete total cost curve, the user can directly compare the two to find the optimal number of workers. More specifically, since total completion time $T = T_p + t_k * b$, where T_p is the total time spent on the job so far, T is a function of the number of workers k, and the user can search across all possible values of k to find the point that minimizes $U(T(k)) - C(k)$.

Unfortunately, the total cost curve is not available in the beginning of the job. Instead, the autoscaling engine must discover this curve incrementally. In this case, directly comparing the values of $U(T(k))$ and $C(k)$ is potentially problematic; a simple scaling condition such as $U(T(k)) > C(k)$ will not work in cases when the utility function is flat. For example, if the utility in the region $T < T_1$ of Figure 5(c) is always greater than the total cost, then we will keep adding workers to the system even though total cost is increasing but utility is not.

Scaling condition. Instead, when deciding whether to add a worker, what the system should compare is the *marginal* utility and the *marginal* cost:

$$U(T(k+1)) - U(T(k)) > C(k+1) - C(k)$$

If this condition is true, then the $(k+1)$th worker is worth more than it costs, and so the overall system is better off compared to having only k workers. This is the scaling condition our autoscaling engine uses.

To make this work for step functions, we must additionally bridge the gaps between the steps, as shown in the red dotted line in 5(c). We add a line that begins at the rightmost point of the higher step (T_1) and ends at either the rightmost point of the lower step (in the case of 3 or more steps) or the initial T (T_0). This enables the system to transition between steps.

Since a utility function explicitly specifies the user's preferences, the utility scaling condition should override the decision made by the throughput scaling condition. If a utility function is not provided, however, the system will fall back to using the throughput scaling condition.

5 STRAGGLER DETECTION

Stragglers are a common source of slowdown in distributed systems. In synchronous distributed training, even a single straggler can prevent the entire cluster from advancing to the next step due to synchronization barriers at the end of each minibatch. Using per-worker runtime statistics, the autoscaling engine can easily keep track of which workers are consistently slower than others on average.

Our system runs a simple straggler detection algorithm at the end of each step (Algorithm 1). A worker is considered a potential straggler if its throughput as a fraction of the median throughput falls below a threshold. Intuitively, this threshold represents the minimum throughput (relative to the median) that the system will tolerate. A threshold of 1 means all workers below the median will be treated as stragglers, while a threshold of 0 means no workers will be treated as stragglers. One reasonable threshold is 0.8, which means the system will tolerate at most a 1.25x increase in total training time caused by stragglers.

Removing a worker as soon as its throughput fraction falls below this threshold is too aggressive. Our goal is to detect persistent stragglers, not workers that happen to run slowly for one or two batches. One solution is to remove the worker only if it has been consistently slow for N batches in a row. However, this does not detect the case when a persistent straggler happens to run normally once in a while, e.g. on every $N-1$'th batch.

```
alpha = 0.1
fraction_threshold = 0.8
fraction_averages = {}

# Called at the end of every step
def record_stats(per_worker_throughputs):
  m = median(per_worker_throughputs)
  for t in per_worker_throughputs:
    fraction = t / m
    fraction_averages[worker] =
      alpha * fraction + (1 - alpha) *
          fraction_averages[worker]

# Called every N steps for each worker
def is_straggler(worker):
  return fraction_averages[worker] <
      fraction_threshold
```

Algorithm 1. Straggler detection.

To address this problem, our system maintains an exponential weighted moving average (EWMA) of the throughput fraction for each worker. A worker will be treated as a persistent straggler only if its *average* throughput fraction falls below the threshold. Small interruptions will only adjust the average instead of marking the worker as not a straggler.

Now we have a mechanism to detect persistent stragglers. To ensure the number of workers stays the same, we add a replacement worker to the system when removing a straggler. As an optimization, our system delays the actual removal of the straggler to when the replacement joins. This ensures that the system's performance at any given time is no worse than if the straggler is not removed.

6 IMPLEMENTATION

As described in section 2, the architectural choices made in TensorFlow pose significant challenges to realizing resource elasticity out of the box. This section describes how we worked around these challenges when building an autoscaling engine on top of this framework.

Checkpoint restart. Before implementing our system, we considered a coarse-grained alternative of achieving resource elasticity that is available out-of-the-box. This approach saves the model to checkpoints, kills all existing processes, spawns new ones on the new set of resources, and restores from the saved checkpoints. However, this approach is too costly (§7.2) so we quickly dismissed it as a viable mechanism for our autoscaling engine.

Bypassing TensorFlow distribution logic. Our system treats TensorFlow processes as individual programs, linking them only through gradient synchronization using Horovod, a third-party allreduce library. Each worker is unaware of the existence of the other workers, finishing each step by applying a black-box (allreduce) function to its own gradients before applying them to its own model.

This architecture decouples the computation graph from the number of workers in the system. When a new worker joins, each existing worker only needs to detach the old allreduce operation from its graph and reattach the new one. Note that this is significantly easier to do than replacing the internal `Send` and `Receive` operations because the allreduce operation, as an external function applied to a worker's gradients, is a much narrower interface. The remainder of the graph does not need to change, and so we do not need to rebuild the model graph.

Minimizing transition time. Every second of time spent on transitioning to a new resource allocation is wasted time that could have been used for training the model. Thus, we wish to minimize the transition time.

When a worker process first starts, it must perform a few time consuming tasks before training the first batch, such as loading libraries, building the model graph, preprocessing the first batch of data, and warming up the GPUs assigned to the process, if any. Depending on the workload, these tasks can take minutes altogether. In an autoscaling system, it is unacceptable to have all resources go idle for minutes every time the system adjusts a job's resource allocation.

To minimize the transition time, our system delays when a new worker joins to until after it has performed all of the above by running a few batches on its own. This brings the transition time down to a few seconds for most workloads.

Bootstrapping new workers. When a new worker joins the system, it must update its model parameters to the existing values before it can begin training, otherwise the gradients it computes will not make sense to the other workers. Our system bootstraps new workers by having them synchronously fetch different slices of the model parameters from different existing workers in parallel. As we will see in the next section, this adds a few more seconds to the transition time for training ResNet-50 on ImageNet.

7 Evaluation

We ran our experiments in two environments. The first is a cluster of 60 machines, each with 16 Intel Xeon CPUs operating at 2.6GHz, 64GB of memory, and a 1 Gbps network connection. The second is a cluster of 8 machines, each with 8 NVIDIA V100 GPUs, 64 Intel Xeon CPUs operating at 2.2Ghz, 250GB of memory, and a 16 Gbps network connection. End-to-end evaluation is run on the GPU cluster while other benchmarks are run on the CPU cluster.

7.1 End-to-end

Figures 6 and 7 compare training with and without using the autoscaling engine we built. To evaluate how our system reacts to different workloads, we train on two datasets of vastly different sizes, CIFAR-10 and ImageNet.

In this experiment, `static` refers to using a constant number of GPUs throughout the entire job, while `autoscaling` refers to starting with a certain number of GPUs but potentially settling on a different number that is more resource efficient. We use the throughput scaling condition with a scaling efficiency threshold of 0.1 and configure our system to scale on 4 GPU increments.

7.1.1 CIFAR-10

We train ResNet-56 on CIFAR-10 for 200 epochs using a fixed global batch size of 1024. The number of GPUs used in each trial varies from 4 to 24, and each worker is assigned one GPU.

This workload is most efficiently run on 4 or fewer GPUs. As shown in the `static` line in Figure 6(a), completion time actually increases as more GPUs are added. This is due in large part to the large drop in performance that results from having to use smaller local batch sizes.

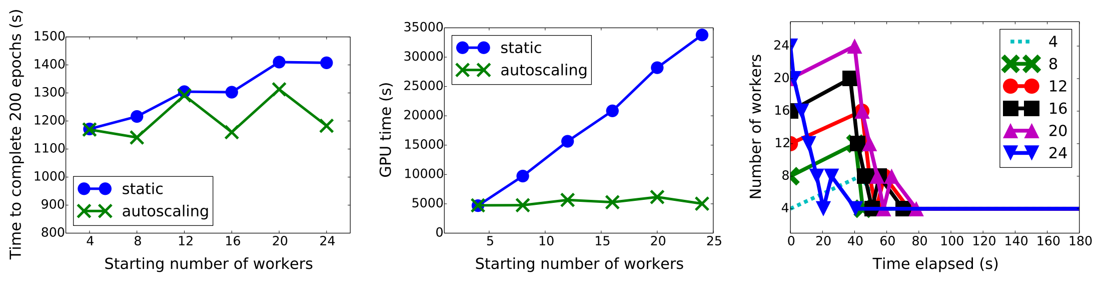
Our autoscaling engine recognizes this and converges to 4 GPUs in all trials. By doing so, we reduce the total completion time by 8.23% on average and up to 16.0%. The real gain, however, comes from the potential cost savings of from discarding GPUs that do not contribute to additional performance. In particular, our system reduces the total GPU time, which scales with cost, by 58.6% on average and up to 85.1% (Figure 6(b)).

An autoscaling system must respond to misallocation of resources quickly in order to reduce cost. Figure 6(c) shows that all trials converge to 4 workers within 61.0 seconds of training and up to 78.4 seconds. This duration is dominated by the time it takes for new workers to bootstrap, which is unavoidable. Note that in the mean time existing workers continue to train and keep the resources busy.

7.1.2 ImageNet

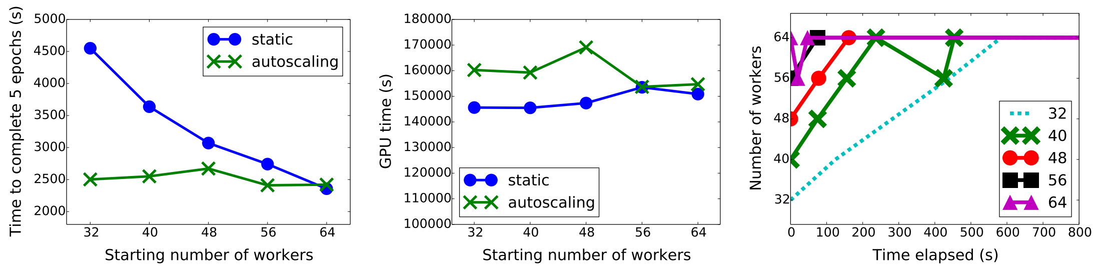
We train ResNet-50 on ImageNet using a maximum global batch size of 4096. Due to the size of the dataset, we only train this model for 5 epochs, which is by far long enough for the number of workers to converge in `autoscaling` mode. Unlike in the CIFAR-10 experiment, we fix the local batch size while letting global batch size rise within a limit of 4096. This allows the system to scale more efficiently, since the per GPU efficiency is not affected.

Figure 7(a) shows that within the range of 32 and 64 GPUs, completion time is minimized with 64 GPUs. This is also



(a) Completion time (b) GPU time (c) Number of workers

Figure 6. Training ResNet-56 on CIFAR-10. In figure 6(c), each line refers to a different starting number of workers.



(a) Completion time (b) GPU time (c) Number of workers

Figure 7. Training ResNet-50 on ImageNet. In figure 7(c) each line refers to a different starting number of workers.

the number of GPUs our autoscaling engine converges to. Our system reduces the total completion time by 19.4% on average and up to 45.0%. For longer jobs such as running the same experiment for 100 epochs, we expect to see a much more dramatic improvement, since the time spent on running on sets of resources that are less efficient (e.g. with 32 workers) will be smaller as a percentage.

However, unlike in the CIFAR-10 experiment, the average GPU time increases by 7.39% on average and up to 14.7% (at 48 workers). This happens when the increase in performance is not enough to offset the cost of the extra workers. From Figure 7(b), we can see that the GPU time is higher even relative to running 64 workers in `static` mode. The average increase of 5.66% is caused by a combination of running at a suboptimal number of workers for a period of time, and the overhead from the idle time during changes in resource allocation.

Scaling from the initial set of GPUs to 64 GPUs took 264 seconds on average and up to 583 seconds (Figure 7(c)). The time to converging to the target number of workers is much longer than in the CIFAR-10 experiment. This is primarily because many trials in this experiment scale out multiple times, and each time the system needs to wait for the new workers to load the libraries, build the graph, fetch existing model parameters etc. Note that scaling only needs to happen in the beginning so the overheads are less observable for long jobs. For example, if we train 100 epochs starting with 32 workers, the time to reach 64 GPUs (583 seconds) will only be 1.16% of the total training time ($\approx 100/5 * 2502$ seconds).

	CIFAR-10	ImageNet
autoscaling (+)	3.179	6.813
autoscaling (-)	2.612	4.376
checkpoint restart	72.756*	81.186*

Table 1. Average idle time during transition in seconds. For autoscaling, (+) refers to adding a worker while (-) refers to removing one. For checkpoint restart, the transition time is the same in both cases. (*) does not include CUDA library loading time, which can add another minute.

7.2 Transition time

As described in section 6, transition time refers to how long resources in the cluster go idle for while the job transitions to a new resource allocation. Minimizing this duration is crucial for ensuring high resource utilization.

Table 1 compares the transition time using the checkpoint restart strawman with those using our autoscaling engine. For training ResNet on both CIFAR-10 and ImageNet, transition time in our system is no more than a few seconds. Adding workers takes longer because the new worker must fetch model parameters from existing workers. For ImageNet, this adds a few extra seconds to the transition time.

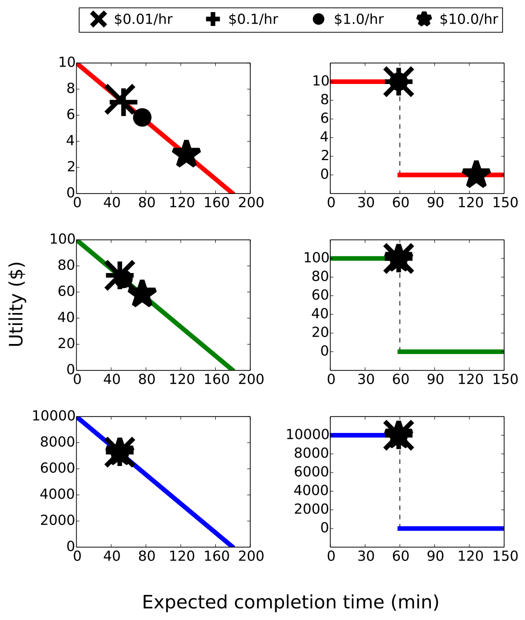


Figure 8. Higher utility lowers completion times for a wider range of worker prices. Each point represents a trial with a different price per worker per time unit. The left column plots the linear function while the right column plots the step function. Lower plots use higher utility.

Compared to checkpoint restart, however, our system has significantly shorter transition times, up to 95.6% reduction for CIFAR-10 and 91.6% reduction for ImageNet.

7.3 Utility functions

In this section, we evaluate two utility functions used by our autoscaling engine. As described in 4.2.2, the linear function represents a constant utility gain for lowering the total completion time regardless of what the completion time is, while the step function can be used to express a target completion time. This experiment trains ResNet-56 on CIFAR-10 for 50 epochs, using a global batch size of 1024 in all trials.

Figure 8 plots these utility functions with the total completion times of running with different prices per worker per time unit. In both the linear and step functions, lower utility values cause a wider spread of completion times across different price points. In particular, an overly expensive price of $10/hour for each worker causes our system to converge to a lower number of workers since the utility gain is not worth it. This inflates completion time by 2.5x for the linear function and 2x for the step function, compared to running with $0.01/hour.

In both functions, when the value of the utility provided by the user is high, the difference in price matters less. In the linear function, when the maximum utility is $10000,

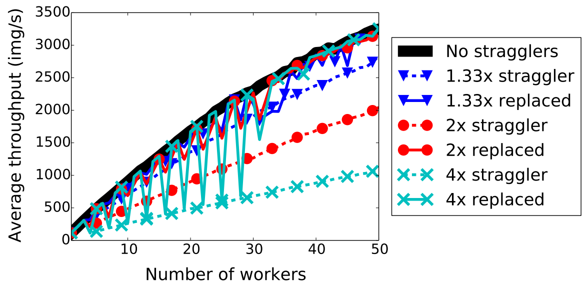


Figure 9. Every 4th worker is a straggler, up to 10 stragglers. Stragglers are removed shortly after they are introduced.

all price points complete within 50 minutes, the fastest for this experiment. If the user provides a step utility function with a different target (60 minutes in figure 8), however, our system converges to that target instead of the minimum. Intuitively, this is because further reducing the completion time is not worth the extra cost of the additional workers, since utility is flat below the target completion time. In general, once the system has reached the highest tier of the function, it will not make any further attempts to scale out.

7.4 Straggler removal

As a byproduct of autoscaling, straggler removal ensures that a job is not only using the right amount of resources, but also using the resources allocated to it efficiently. In this experiment, we train ResNet-56 on CIFAR-10 for 100 epochs using a global batch size of 1024. We ignore the scaling condition and spawn 1 worker every time after running 10 batches on a new cluster size. Additionally, we introduce 1 straggler into the system for every 3 normal workers added, up to 10 stragglers. Slowdown refers to computation slowdown; a 2x straggler is simulated by assigning twice as many samples to that worker per step.

Figure 9 shows that our system responds stragglers quickly. Every time a straggler is introduced there will be a downward spike in throughput. However, these spikes are short; a straggler is already marked for removal by the time the new worker in the next round joins. In this experiment, the average lifetime of a straggler is about 1-2 minutes, since that is the amount of time it takes for a new worker to join after being spawned.

Figure 10 shows that the replacing the stragglers in this manner has a large effect on the completion time of the job. In particular, our straggler removal strategy reduces the completion time by 51.4%, 34.1%, and 19.9% for a 4x, 2x, and 1.33x slowdown respectively. Compared to running without stragglers, our strategy has similar completion times; all trials were within 12.5% of the baseline.

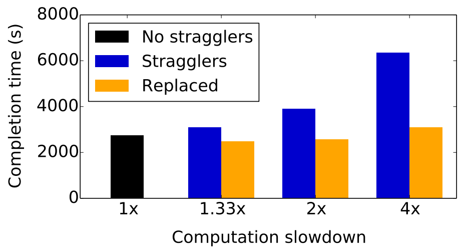


Figure 10. Replacing stragglers with new workers yields completion times similar to running with no stragglers in the first place.

8 Related Work

Elasticity in machine learning. The idea of elasticity has been explored in the context of distributed machine learning. (Huang et al., 2015) considers elasticity in the context of declarative machine learning on MapReduce. However, many of the techniques presented are designed specifically for the master-worker architecture in MapReduce—which itself already provides the mechanisms for elasticity—and so do not apply to modern deep learning workloads.

(Qiao et al., 2018) explores elasticity in the context of CPU-based machine learning using the parameter server architecture. Their discussion of multi-tenant priority as a potential scaling heuristic is complementary to our work. However, the low level event-driven API they propose is incompatible with state-of-the-art deep learning frameworks.

(Harlap et al., 2017) exploits dynamic pricing on public clouds in order to lower costs for machine learning workloads through elasticity. Although they specifically target the parameter server architecture and does not consider deep learning workloads, many of their ideas can be applied alongside those presented in this work.

Autoscaling systems. Resource elasticity and autoscaling are ubiquitous outside machine learning. Production systems in many areas such as cloud computing (AWS; Azure; GCE), batch data processing (Dean & Ghemawat, 2008; Zaharia et al., 2012) and cluster management (Vavilapalli et al., 2013; Hindman et al., 2011; Rensin, 2015), already deploy autoscaling to cut cost and increase efficiency. Here we will discuss two examples to highlight the differences between these workloads and those in distributed learning.

Dynamic resource allocation in Apache Spark (Or, 2014) scales out exponentially while there are tasks pending to be executed, and scales in when an executor (worker) is idle. Autoscaling in Kubernetes (Szczepkowski & Wielgus, 2016) scales in and out to maintain an average CPU utilization of 50% across all pods. In distributed learning, there are always pending tasks, workers are never idle, and resource utilization is stable across minibatches. Therefore, these scaling heuristics do not apply and our system must devise ones that are more specific to our workloads. Scaling schedules such as the exponential increase in Apache Spark may help reduce scaling overheads. We leave exploring alternative schedules as future work.

Straggler mitigation. Distributed synchronous SGD is vulnerable to stragglers. As a system scales, their effects will be more and more pronounced. State-of-the-art techniques of handling them include using backup workers (Abadi et al., 2016; Chen et al., 2016), redundant data encoding (Karakus et al., 2017), bounded staleness (Ho et al., 2013), and work reassignment (Harlap et al., 2016).

However, these are all variants of overprovisioning. For example, assigning backup workers means a small fraction of the computed results will be wasted. Further, in the event of persistent stragglers, these techniques do not address the root cause by removing these stragglers. We argue this is largely because of the widespread assumptions that resources must be fixed in distributed learning workloads. Our system can be used alongside with these techniques to handle persistent stragglers.

Utility functions. Utility functions are widely used in economics (Ratchford, 1982) and artificial intelligence (Russell & Norvig, 2016) to specify preferences for certain actions. In computing systems, they have been used for VM resource management (Minarolli & Freisleben, 2011), live video analytics (Zhang et al., 2017a), and stream management (Carney et al., 2003).

9 Conclusion

In this paper, we presented the first autoscaling system for distributed deep learning. As new types of models and workloads arise, such a system will help bridge the gap between unfamiliarity with a given job to quickly being able to run it with high resource efficiency. Although we have addressed the main challenges that prevent resource elasticity today, there are still many remaining questions that need to be answered by the broader community.

Resource elasticity is the first step of a broader effort to tune systems parameters using live performance feedback from training. In the future, it is worth exploring whether other systems parameters, such as synchronization mechanism, device placement, and types of parallelism, can be autotuned dynamically to improve the performance of the system.

References

Abadi, M., Barham, P., Chen, J., Chen, Z., Davis, A., Dean, J., Devin, M., Ghemawat, S., Irving, G., Isard, M., et al. Tensorflow: A system for large-scale machine learning. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pp. 265–283, 2016.

AWS. Auto Scaling: `https://aws.amazon.com/autoscaling/`.

Azure. Autoscale: `https://azure.microsoft.com/en-us/features/autoscale/`.

Carney, D., Çetintemel, U., Rasin, A., Zdonik, S., Cherniack, M., and Stonebraker, M. Operator scheduling in a data stream manager. In *Proceedings of the 29th international conference on Very large data bases-Volume 29*, pp. 838–849. VLDB Endowment, 2003.

Chen, J., Pan, X., Monga, R., Bengio, S., and Jozefowicz, R. Revisiting distributed synchronous sgd. *arXiv preprint arXiv:1604.00981*, 2016.

Chen, T., Li, M., Li, Y., Lin, M., Wang, N., Wang, M., Xiao, T., Xu, B., Zhang, C., and Zhang, Z. Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems. *arXiv preprint arXiv:1512.01274*, 2015.

Dean, J. and Ghemawat, S. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

Deng, J., Dong, W., Socher, R., Li, L.-J., Li, K., and Fei-Fei, L. Imagenet: A large-scale hierarchical image database. In *2009 IEEE conference on computer vision and pattern recognition*, pp. 248–255. Ieee, 2009.

GCE. Autoscaling groups of instances: `https://cloud.google.com/compute/docs/autoscaler/`.

Gedik, B., Schneider, S., Hirzel, M., and Wu, K.-L. Elastic scaling for data stream processing. *IEEE Transactions on Parallel and Distributed Systems*, 25(6):1447–1463, 2013.

Goyal, P., Dollár, P., Girshick, R., Noordhuis, P., Wesolowski, L., Kyrola, A., Tulloch, A., Jia, Y., and He, K. Accurate, large minibatch sgd: Training imagenet in 1 hour. *arXiv preprint arXiv:1706.02677*, 2017.

Harlap, A., Cui, H., Dai, W., Wei, J., Ganger, G. R., Gibbons, P. B., Gibson, G. A., and Xing, E. P. Addressing the straggler problem for iterative convergent parallel ml. In *Proceedings of the Seventh ACM Symposium on Cloud Computing*, pp. 98–111. ACM, 2016.

Harlap, A., Tumanov, A., Chung, A., Ganger, G. R., and Gibbons, P. B. Proteus: agile ml elasticity through tiered reliability in dynamic resource markets. In *Proceedings of the Twelfth European Conference on Computer Systems*, pp. 589–604, 2017.

He, K., Zhang, X., Ren, S., and Sun, J. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 770–778, 2016.

Hindman, B., Konwinski, A., Zaharia, M., Ghodsi, A., Joseph, A. D., Katz, R. H., Shenker, S., and Stoica, I. Mesos: A platform for fine-grained resource sharing in the data center. In *NSDI*, volume 11, pp. 22–22, 2011.

Ho, Q., Cipar, J., Cui, H., Lee, S., Kim, J. K., Gibbons, P. B., Gibson, G. A., Ganger, G., and Xing, E. P. More effective distributed ml via a stale synchronous parallel parameter server. In *Advances in neural information processing systems*, pp. 1223–1231, 2013.

Hoffer, E., Hubara, I., and Soudry, D. Train longer, generalize better: closing the generalization gap in large batch training of neural networks. In *Advances in Neural Information Processing Systems*, pp. 1731–1741, 2017.

Huang, B., Boehm, M., Tian, Y., Reinwald, B., Tatikonda, S., and Reiss, F. R. Resource elasticity for large-scale machine learning. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, pp. 137–152, 2015.

Jia, X., Song, S., He, W., Wang, Y., Rong, H., Zhou, F., Xie, L., Guo, Z., Yang, Y., Yu, L., et al. Highly scalable deep learning training system with mixed-precision: Training imagenet in four minutes. *arXiv preprint arXiv:1807.11205*, 2018.

Karakus, C., Sun, Y., Diggavi, S., and Yin, W. Straggler mitigation in distributed optimization through data encoding. In *Advances in Neural Information Processing Systems*, pp. 5434–5442, 2017.

Keskar, N. S., Mudigere, D., Nocedal, J., Smelyanskiy, M., and Tang, P. T. P. On large-batch training for deep learning: Generalization gap and sharp minima. *arXiv preprint arXiv:1609.04836*, 2016.

Mikami, H., Suganuma, H., U-chupala, P., Tanaka, Y., and Kageyama, Y. Massively distributed SGD: ImageNet/ResNet-50 training in a flash. *arXiv preprint arXiv:1811.05233*, 2018.

Minarolli, D. and Freisleben, B. Utility-based resource allocation for virtual machines in cloud computing. In *2011 IEEE symposium on computers and communications (ISCC)*, pp. 410–417. IEEE, 2011.

Or, A. Apache Spark Dynamic Resource Allocation: `https://spark.apache.org/docs/latest/job-scheduling.html#dynamic-resource-allocation`, 2014.

Paszke, A., Gross, S., Chintala, S., Chanan, G., Yang, E., DeVito, Z., Lin, Z., Desmaison, A., Antiga, L., and Lerer, A. Automatic differentiation in PyTorch. 2017.

Patarasuk, P. and Yuan, X. Bandwidth optimal all-reduce algorithms for clusters of workstations. *Journal of Parallel and Distributed Computing*, 69(2):117–124, 2009.

Qiao, A., Aghayev, A., Yu, W., Chen, H., Ho, Q., Gibson, G. A., and Xing, E. P. Litz: Elastic framework for high-performance distributed machine learning. In *2018 {USENIX} Annual Technical Conference ({USENIX}{ATC} 18)*, pp. 631–644, 2018.

Ratchford, B. T. Cost-benefit models for explaining consumer choice and information seeking behavior. *Management Science*, 28(2):197–212, 1982.

Rensin, D. K. *Kubernetes - Scheduling the Future at Cloud Scale*. 1005 Gravenstein Highway North Sebastopol, CA 95472, 2015. URL `http://www.oreilly.com/webops-perf/free/kubernetes.csp`.

Russell, S. J. and Norvig, P. *Artificial intelligence: a modern approach*. Malaysia; Pearson Education Limited,, 2016.

Sergeev, A. and Del Balso, M. Horovod: fast and easy distributed deep learning in TensorFlow. *arXiv preprint arXiv:1802.05799*, 2018.

Smith, S. L. and Le, Q. V. A bayesian perspective on generalization and stochastic gradient descent. *arXiv preprint arXiv:1710.06451*, 2017.

Sun, P., Feng, W., Han, R., Yan, S., and Wen, Y. Optimizing network performance for distributed DNN training on GPU clusters: Imagenet/alexnet training in 1.5 minutes. *arXiv preprint arXiv:1902.06855*, 2019.

Szczepkowski, J. and Wielgus, M. Autoscaling in Kubernetes: `https://kubernetes.io/blog/2016/07/autoscaling-in-kubernetes/`, 2016.

Vavilapalli, V. K., Murthy, A. C., Douglas, C., Agarwal, S., Konar, M., Evans, R., Graves, T., Lowe, J., Shah, H., Seth, S., et al. Apache hadoop yarn: Yet another resource negotiator. In *Proceedings of the 4th annual Symposium on Cloud Computing*, pp. 5. ACM, 2013.

XLA, T. Optimizing Compiler for TensorFlow: `https://www.tensorflow.org/xla`.

Ying, C., Kumar, S., Chen, D., Wang, T., and Cheng, Y. Image classification at supercomputer scale. *arXiv preprint arXiv:1811.06992*, 2018.

Zaharia, M., Chowdhury, M., Das, T., Dave, A., Ma, J., McCauley, M., Franklin, M. J., Shenker, S., and Stoica, I. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*, pp. 2–2. USENIX Association, 2012.

Zhang, H., Ananthanarayanan, G., Bodik, P., Philipose, M., Bahl, P., and Freedman, M. J. Live video analytics at scale with approximation and delay-tolerance. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, pp. 377–392, 2017a.

Zhang, H., Stafman, L., Or, A., and Freedman, M. J. Slaq: quality-driven scheduling for distributed machine learning. In *Proceedings of the 2017 Symposium on Cloud Computing*, pp. 390–404. ACM, 2017b.