TICTAC: ACCELERATING DISTRIBUTED DEEP LEARNING WITH COMMUNICATION SCHEDULING

Sayed Hadi Hashemi [*1] **Sangeetha Abdu Jyothi** [*1] **Roy H Campbell** [1]

ABSTRACT

State-of-the-art deep learning systems rely on iterative distributed training to tackle the increasing complexity of models and input data. In this work, we identify an opportunity for accelerating distributed DNN training in systems that rely on graph representation for computation, such as TensorFlow and PyTorch, through communication scheduling. We develop a system, TicTac, that reduces the iteration time by identifying and enforcing parameter transfers in the order in which the parameters are consumed by the underlying computational model, thereby guaranteeing near-optimal overlap of communication and computation. Our system is implemented over TensorFlow and enforces the optimal ordering by prioritization of parameter transfers at the Parameter Server in data parallel training. TicTac requires no changes to the model or developer inputs and improves the throughput by up to 37.7% in inference and 19.2% in training, while also reducing straggler effect by up to $2.3\times$. Our code is publicly available.

1 INTRODUCTION

Deep learning has grown significantly in the past decade, fuelled by the flexibility of development offered by machine learning frameworks, availability of rich data, and readily accessible distributed high-performance computing. The computational cost of training sophisticated deep learning models has long outgrown the capabilities of a single high-end machine, leading to distributed training being the norm in a typical AI pipeline. Training a deep learning model is an iterative job which may take days to weeks in high-end clusters today.

Computational graphs are used to represent the training jobs in state-of-the-art systems (Abadi et al., 2016; Chen et al., 2015; Paszke et al., 2017). In the commonly-used Model Replica or data parallel mode of training, the input data is partitioned and processed at participating workers using identical computational graphs. Each iteration typically lasts milliseconds to seconds. At the end of each iteration, servers exchange a relatively large amount of data associated with parameter updates to aggregate the results of the iteration. This communication overhead has a substantial impact on throughput of the system and also limits its scalability (Sridharan et al., 2018; Alistarh et al., 2017). Even a small improvement in communication overhead can improve the learning time by hours in these long-running learning jobs.

The iteration time in deep learning systems depends on the time taken by (i) computation, (ii) communication and (iii) the overlap between the two. When workers receive the parameters from the parameter server at the beginning of each iteration, all parameters are not used simultaneously; they are consumed based on the dependencies in the underlying DAG. While one particular schedule of parameter transfers (over the complete set of parameters in a given model in a single iteration) may facilitate faster computation, another may cause blockage. Hence, identifying the best schedule of parameter transfers is critical for reducing the blocking on computation (determined by DAG dependencies), and in turn improving the overlap and the iteration time.

We observe that the schedule of data transfers in current systems (Abadi et al., 2016; Chen et al., 2015; Paszke et al., 2017) is determined arbitrarily during execution without considering the impact on overlap. We quantify the observed combinations in TensorFlow and find that in a trial with 1000 iterations on ResNet-V2-50, every iteration had a unique order of received parameters which has not been observed previously. This random order of parameter transfers at workers has two performance implications. First, the iteration time, and in turn throughput (number of samples processed per second), suffers significantly due to suboptimal overlap. Second, even in the same iteration, multiple workers might follow different schedules of data transfers, leading to stragglers during synchronized training.

[*]Equal contribution [1]Department of Computer Science, University of Illinois at Urbana-Champaign, Urbana, IL. Correspondence to: Sayed Hadi Hashemi <hashemi3@illinois.edu>.

Proceedings of the 2^{nd} SysML Conference, Palo Alto, CA, USA, 2019.

Past work has attempted to address this issue by enforcing the same order of parameter transfers at all workers. However, these solutions are restricted to earlier systems with layer-by-layer model representation (Arnold, 2016; Cui et al., 2016; Zhang et al., 2017) where finding the optimal order of execution is trivial (Cui et al., 2014). In modern systems with DAG representation (Abadi et al., 2016; Paszke et al., 2017), this is a non-trivial challenge.

In this work, we devise a systematic methodology for deriving near-optimal schedules of parameter transfers through critical path analysis on the underlying computational graph. This allows maximal overlap of computation and communication and prevents stragglers arising from random order of parameter transfers at workers. We also develop a lightweight resource-level enforcement mechanism over TensorFlow (Abadi et al., 2016). These techniques form the core of our system, TicTac, which achieves substantial performance improvement while requiring no changes in the model or developer inputs.

In summary, we make the following contributions:

- We identify an opportunity for improving performance in state-of-the-art deep learning systems with Parameter Server-based aggregation through prioritized parameter transfers (§2).
- We define a metric to quantify the efficiency of a given execution: the overlap coefficient (§3).
- We propose two heuristics, TIC and TAC, for near-optimal scheduling of computation and communication in Model Replica with Parameter Server.
- We implement our system over TensorFlow (§ 5). The code is publicly available. [1]
- We extensively evaluate the performance of our system in GPU and high-end CPU environments under training and inference of DNN models and show that throughput can be improved by up to 37% (§6).

2 Background and Motivation

Our system focuses on network optimization in deep learning frameworks with DAG representation of computational graphs (Abadi et al., 2016; Paszke et al., 2017), Model Replica (MR) mode of distribution and Parameter Servers. The performance improvement provided by TicTac is beneficial in two key environments. First, it improves throughput and iteration time in clud environment with commodity hardware or on-demand clusters where high resiliency is critical (workers may be preempted). Second, in online reinforcement learning with workers for training and separate active agents for inference, enforced ordering can improve the inference time. In this environment, the active agents are reading parameters from the PS or decentralized workers as shown in Figure 3. While decentralized aggregation techniques (such as all-reduce and Horovod (Sergeev & Balso, 2018)) are gaining traction in high performance networking, TicTac does not address such systems and is focused on PS.

In this section, we give a brief overview of deep learning systems, prior techniques proposed in these systems to mitigate network overhead, and opportunities for further optimization.

2.1 Network Optimization in DNN training

In deep learning systems, high GPU utilization can be achieved in two ways: (i) when total communication time is less than or equal to the computation time and (ii) with efficient overlap of communication and computation. Several techniques have been proposed to improve GPU utilization.

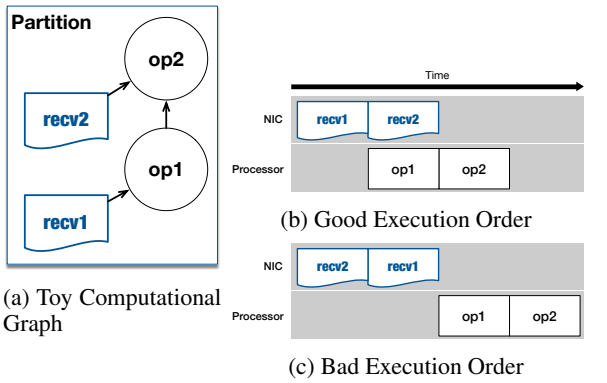
Increasing computation time: The fraction of computation time relative to communication time can be increased by increasing the batch size (Iandola et al., 2016). However, this approach suffers from decreased accuracy (Keskar et al., 2016) and may not be generally applicable under resource constraints. (Goyal et al., 2017; Cho et al., 2017; You et al., 2017; Akiba et al., 2017).

Decreasing communication time: Solutions for reducing network communication have taken multiple approaches — modifying the machine learning algorithm to reduce communication cost (Alistarh et al., 2017; Wen et al., 2017; Zhang et al., 2017), reducing the precision of parameter representation (Vanhoucke et al., 2011; Courbariaux et al., 2015; Gupta et al., 2015), changing the network primitives to collective (e.g. all reduce) (Goyal et al., 2017; Cho et al., 2017; Amodei et al., 2015; You et al., 2017; Akiba et al., 2017) or broadcast (Zhang et al., 2017).

Smarter interleaving of computation and communication: Several layer-by-layer systems (Arnold, 2016; Cui et al., 2016; Zhang et al., 2017), where the models are sequential and obtaining the order is trivial (Cui et al., 2014), adopt this approach. These solutions are not applicable to current DAG-based systems such as TensorFlow (Abadi et al., 2016) and PyTorch (Paszke et al., 2017). The inter-resource dependency considered in (Cui et al., 2016) (with GPU memory) and in (Zhang et al., 2017) (with network) is constrained to layer-by-layer models.

In this work, we focus on *improving the iteration time through better and predictable overlap of communication and computation.* Techniques for optimizing communication and communication time are orthogonal to our system and may be used in parallel with TicTac.

[1] https://github.com/xldrx/tictac



(a) Toy Computational Graph

(b) Good Execution Order

(c) Bad Execution Order

Figure 1: Impact of multi-resource operation ordering on performance

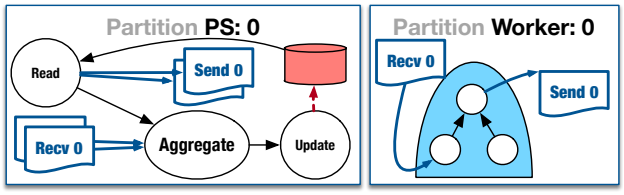


Figure 2: Distributed execution of Model-Replica with Parameter Server

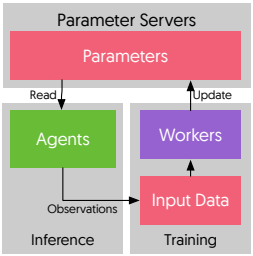


Figure 3: A general reinforcement learning setup

2.2 Opportunity for Optimization

We demonstrate the opportunity for accelerating DNN training through a better understanding of the internal computational model in TensorFlow which is a Directed Acyclic Graph (DAG). The parameter transfers are denoted by $send$ and $recv$ operations in the DAG. In MR, each worker has an identical copy of the computational DAG. In the **worker DAG**, all `recv` ops are roots and $send$ ops are leaves. Thus $recv$ ops can block the initialization of a computation branch in the DAG. Since the activation of various branches of computation in the DAG is dependent on the $recv$ at the root of the branch, the ordering in MR can be reduced to the ordering of `recv` ops in workers. DAG at the PS is different from that at workers. **PS DAG** has five ops per parameter: aggregation, $send$, $recv$, read, and update. Since $send$ and $recv$ at the PS are not blocked by computation, our focus is on the worker DAG.

In the simple DAG shown in Figure 1a, a sample worker DAG, there are two possible schedules for parameter transfers. If $recv_1$ (parameter 1 transfer from PS to the worker) happens before $recv_2$ (parameter 2 transfer), it reduces the blocking on computation time and improves the overlap. The reverse order results in increased iteration time due to blocking on computation. Thus, in a distributed environment, network can block computation based on dependencies in the DAG. This can lead to under-utilization of computational capacity, in turn resulting in sub-optimal performance. In addition, variation in iteration time caused by random order of parameter transfers across multiple workers can lead to straggling effect.

The impact of poor overlap can be significant in DNN training due to complexity of state-of-the-art models. For instance, ResNet-v2-152 (He et al., 2016) has 363 parameters with an aggregate size of 229.5MB. The computational graph associated with this neural network has 4655 operations in the TensorFlow framework. Finding the optimal schedule in this complex DAG involves evaluating 363! combinations. We run 1000 iterations of learning over ResNet-v2-50, Inception-v3 and VGG-16 networks and observe the order of network transfers at a single worker. The observed order of parameter transfer is unique in ResNet-v2-50 and Inception-v3 networks across the 1000 runs. In VGG-16, we observe 493 unique combinations across 1000 runs.

2.3 Comparison with Other Distributed Systems

It is worth noting that deep learning systems with computational graphs are fundamentally different from graph processing systems (Malewicz et al., 2010; Hoque & Gupta, 2013; Xin et al., 2013). In deep learning, the graph is a representation of the computation to be done on the input data. In graph processing systems, the graph itself is the input to the computation. As a result, graphs in DNN frameworks are a few orders of magnitude smaller than a typical large-scale graph processing system. Iterations in DNN frameworks are identical, and network communication pattern is fixed. This may not be true for graph processing systems.

In stream processing systems, the relationship between processing elements are represented using graphs. These systems allow pipelining, with different partitions of input data being processed in different elements along the pipeline at the same time. In contrast, DNN frameworks process the entire batch of input at a processing element at a worker. Pipelining is not employed in this environment. Hence, optimizations proposed for stream processing cannot be borrowed here.

3 Quantifying Performance

In this section, we explore methods for quantitatively comparing the efficiency of multiple schedules. Towards this goal, we formally define the scheduling problem and investigate the feasibility of finding an optimal solution. Finally,

we define a metric that is used to quantify the efficiency of a schedule.

3.1 Scheduling Problem

The objective is to find the optimal schedule of network transfers that minimizes the iteration time by improving the communication/computation overlap. The network transfers of parameters ($recv$ ops) are roots in the computational graph at the worker. The branch of computation ops dependent on a $recv$ op can be executed only after the network transfer is completed. Thus, the order of network transfers can determine the order of computation as well as the extent of overlap. We focus on improving the overlap, and in turn the iteration time, by choosing a near-optimal schedule of parameter transfers.

The inputs to this optimization problem are: (a) *the worker DAG*, and (b) *a time oracle*. The time oracle ($Time(op)$) predicts the execution time of a given op. For computation ops, this indicates the elapsed time on a computation resource. For communication ops, this represents the transfer time on the communication medium. We compute the time assuming that the resource is dedicated to the op under consideration.

The output of the scheduling algorithm is a feasible schedule of ops in the DAG tagger by priorities. Ops in a computational DAG may have multiple feasible topological orders. However, some of them may result in a bad iteration time (as explained in Figure 1). We want to limit the execution path to take the one that improves the training performance. We achieve this with priority numbers. Priority number is a positive integer assigned to an op in the DAG. A higher priority op is given a lower priority number. An op may not be assigned a priority if it need not be ordered. Multiple ops may be assigned the same priority if their relative order is insignificant.

The order is enforced in the following manner. When we need to select a new op from the ready-to-execute queue, we randomly choose from among the set of ops that contain the lowest priority number and those without any priority number. It is worth noting that priority only specifies relative order among candidate ops in the ready-to-execute queue at a given resource, and the resulting order will still respect the topological order specified by the DAG.

The problem of finding the optimal schedule is NP-hard. A simpler version of the optimal execution problem with homogeneous hardware can be formally defined as follow (Using notion in (Pinedo, 2008)): $P_m|M_i, prec|C_{max}$

In this formulation, P_m represents multiple parallel resources with identical performance. M_i assigns the operations to specific resources, i.e., computation ops vs. communication. $prec$ describes the dependency relation of ops in the DAG. The C_{max} represents the goal of scheduling is to minimize the last node completion time.

This problem is still open (Brucker & Knust, 2007) and simpler cases are proven to be NP-Hard. While there exist approximations for relaxed versions of this problem, to the best of our knowledge, there is no solution or approximation with guaranteed bounds for our original problem.

3.2 Defining Overlap Coefficient

The two major contributors to total DNN iteration time (T) are network transfer time or the communication time (N) and the computation time (C). Since the computation and communication may overlap, the total time $T \leq N + C$. Given a GPU/CPU/TPU environment, we assume the computation time, C, to be constant. We ignore the case of computation stragglers and focus on communication.

We define two metrics that define the DNN iteration time: (a) the communication/computation ratio, ρ and (b) the overlap coefficient, α. The ratio of communication to computation, denoted by ρ, determines the extent of benefits achievable. When $\rho < 1$, communication time is smaller than the total computation time, providing ample opportunity for running GPUs at high utilization.

The second factor affecting the GPU utilization is the overlap coefficient, $\alpha = \frac{N+C-T}{min(N,C)}$. $N + C$ is the iteration time when there is no overlap, and T is the actual iteration time. The difference between these quantities is the extent of overlap. The maximum overlap possible is given by $min(N, C)$, which is achieved when the smaller quantity completely overlaps with the large quantity. The difference is normalized by this factor to obtain the overlap coefficient, $\alpha \in [0, 1]$.

The GPU utilization ($U = \frac{C}{T}$) can be represented in terms of these coefficients:

$$U = \frac{C}{N + C - \alpha * min(N, C)} = \frac{1}{1 + \rho - \alpha * min(\rho, 1)}$$

The goal of our scheduling algorithms is to achieve high GPU efficiency by maximizing α, i.e., increasing the overlap of communication and computation. The impact of our scheduling algorithm on α, and in turn the GPU utilization is plotted in Figure 4 using Inception v3 with 2 workers and 1 PS as an example.

4 Scheduling Algorithms

In this section, we present two heuristics to derive the optimal schedule of $recv$ ops using a given worker DAG (§3). The intuition behind our heuristics is to prioritize transfers that speed up the critical path in the DAG by reducing blocking on computation caused by parameter transfers.

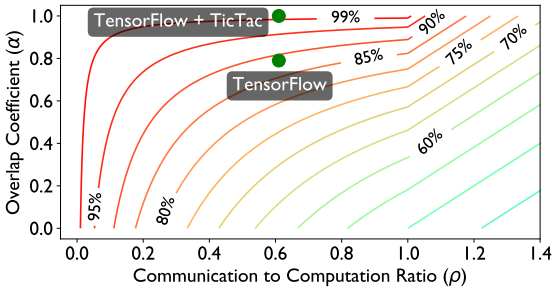


Figure 4: Improvement in GPU utilization with TicTac

Timing-Independent Communication scheduling (TIC): In TIC, we assign priorities based only on vertex dependencies in the DAG (ignoring the execution time of each op). Higher priorities are given to transfers which are least blocking on computation. In this algorithm, we ignore the time oracle, $Time$, and assume all ops have equal cost.

Timing-Aware Communication scheduling (TAC): In this algorithm, we prioritize transfers that maximize α by using information on (a) vertex dependencies among ops specified by the computational DAG, and (b) execution time of each op estimated with time oracle.

4.1 Op properties

Before delving into the algorithms, we define properties associated with ops that are used in the scheduling algorithms. The inputs are the worker dataflow DAG (G), a time oracle ($Time$), available communication channels on a device (C) and a set of outstanding (to-be-activated) $recv$s ops (R). We assume that $recv$ ops not in R have their corresponding transfers completed. These properties are updated using the algorithm 1.

Communication Dependency (op.dep): This is the set of $recv$ ops that an op is directly or transitively dependent on. For example, in figure 1a, $op_2.dep = \{recv_1, recv_2\}$. We extract the communication dependencies using a depth-first post-fix graph traversal on the DAG.

Communication Time (Op.M): Communication time of an op is the total network transfer time required to complete that op. For a $recv$ op, this is the time required to complete its corresponding transfer, given by $Time(recvOp)$. For other ops, this is the total time to complete all outstanding dependent transfers, given by $\sum_{r \in op.dep \cap R} Time(r)$. For example, in Figure 1a, $op_1.M = Time(recv_1)$ and $op_2.M = Time(recv_1) + Time(recv_2)$.

For **recv ops**, we define two additional properties.

Algorithm 1: Property Update Algorithm

```
// Update properties for the given the set of
   outstanding read ops R
Function UpdateProperties(G, Time, R):
    foreach op ∈ G do
        op.M ← Σ_{∀r∈op.dep∩R} Time(r);
    end
    foreach op ∈ R do
        op.P ← 0;
        op.M+ ← +∞;
    end
    foreach op ∈ G − R do
        D ← op.dep ∩ R;
        if |D| = 1 then
            ∀r ∈ D : r.P ← r.P + Time(op);
        end
        if |D| > 1 then
            ∀r ∈ D : r.M+ ← min{r.M+, op.M};
        end
    end
end
```

Directly-Dependent Compute Load (recvOp.P): This property represents the computational benefit of completing a $recv$ op. More specifically, it is the total $Time(op)$ for all ops that can be activated only by completing this $recvOp$, but not without it. These ops are those whose communication dependencies contain only this outstanding $recvOp$ (it is admissible to have communication dependencies on other completed $recv$ operations). For example, in Figure 1a, $recv_1.P = Time(op_1)$ and $recv_2.P = 0$ since no op can execute with completion of only $recv_2$.

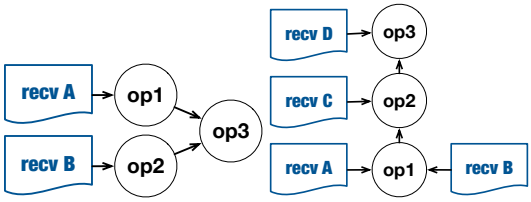
Impending Communication Load (recvOp.M$^+$): This property helps us to identify candidate $recv$ ops to be activated, given the current $recv$ is completed. In more detail, it is the minimum communication cost to activate a computation op which has multiple $recv$ dependencies including the one under consideration. For example, in Figure 1a, $read_1.M^+ = read_2.M^+ = Cost(read_1) + Cost(read_2)$. Please note that $recvOp.M^+$ includes the communication time of that recvOp.

4.2 Timing-Independent Communication Scheduling (TIC)

The goal of this algorithm is to prioritize those transfers which reduces blocking on network transfers. Our intuition is that information on DAG structure alone can provide significant improvement.

To achieve this goal, we define a generic time function which only uses the number of communication ops instead of time taken by an op. We use this simple cost function to generate the schedule in Timing-Independent Communication scheduling (TIC).

General Time Oracle: We define a simple universal time



(a) Case 1 (b) Case 2

Figure 5: Sample DAG

oracle as follows:

$$Time_{General}(op) = \begin{cases} 0 & \text{if } op \text{ is not recv} \\ 1 & \text{if } op \text{ is recv} \end{cases} \quad (1)$$

The complete solution is given in Algorithm 2.

Algorithm 2: Timing-Independent Communication Scheduling (TIC)

```
1  Function TIC(G)
2      FindDependencies(G) ;
3      UpdateProperties(G,R,Time={Computation: 0, Communication: 1});
4      ∀op in G, if op is recv : op.priority ← op.M+;
5  end
```

4.3 Timing-Aware Communication Scheduling (TAC)

The goal of this algorithm is to prioritize those transfers which reduces the blocking of computation, i.e., speeding up transfers on the critical path. To achieve this goal, the algorithm focuses on two cases. First, it considers the opportunity for overlapping communication and computation. Second, in the case of equal overlap or absence of it, it looks at the impending transfers to choose one which eliminates the computation block sooner.

To better describe the logic, we begin with an example for each case.

Case 1: In Figure 5a, when deciding between two read ops, A and B, A should precede B iff:

$$\begin{aligned} \mathbf{A \prec B} &\iff T(A \to B) < T(B \to A) \\ &\iff M_A + \max\{P_A, M_B\} + P_B < M_B + \max\{P_B, M_A\} + P_A \\ &\iff M_A + P_A + M_B - \min\{P_A, M_B\} + P_B < \\ &\qquad M_B + P_B + M_A - \min\{P_B, M_A\} + P_A \\ &\iff \min\{P_B, M_A\} < \min\{P_A, M_B\} \end{aligned}$$

Therefore:

$$A \prec B \to \min\{P_B, M_A\} < \min\{P_A, M_B\} \quad (2)$$

Case 2: In Figure 5b, when all $recv$ ops are outstanding, their P is 0, making them equivalent under the comparison in Equation 2. Obviously, $recv_A$ and $recv_B$ should precede other $recv$s. Hence, we use M^+ to break the ties: $recv_A.M^+ = recv_B.M^+ = Time(recv_A) + Time(recv_B) < recv_C.M^+ < recv_D.M^+$.

Comparator: We combine results from the two cases to make a comparator that extends to multiple read ops. This is an approximate induction, which may not be correct in general. The result is the `Comparator` function in algorithm 3. It is easy to prove that this function is transitive and can be used for partial ordering.

The ordering algorithm takes a partition graph on a worker, calculates the communication dependencies, then while there is an outstanding $recv$ op, it updates properties, finds the smallest $recv$ op with respect to the comparator and then removes the $recv$ from the outstanding set and assign it a higher priority relative to others.

Algorithm 3: Timing-Aware Communication Scheduling (TAC)

```
   // Compare two given recv ops
1  Function Comparator(Op_A,Op_B): Bool
2      A ← min(P_A, M_B);
3      B ← min(P_B, M_A);
4      if A ≠ B then
5          return A < B
6      else
7          return M_A^+ < M_B^+
8      end
9  end
10 Function TAC(G,Time)
11     FindDependencies(G) ;
12     R ← {op|∀op in G, op is recv};
13     count ← 0;
14     while R is not empty do
15         UpdateProperties(G,R,Time);
16         Find the minimum op from R wrt Comparator;
17         Remove op from R;
18         op.priority ← count;
19         count ← count + 1;
20     end
21 end
```

5 System Design

In this section, we provide a brief overview of the system design and implementation.

The system has four main components: the tracing module, the time oracle estimator, the ordering wizard, and the enforcement module (shown in Figure 6).

Tracing Module: This module collects runtime stats from an execution, which is later fed to the time oracle estimator.

Time Oracle: The time oracle is responsible for estimating the runtime of each op in the system based on the execution timing stats. Note that the runtime may vary depending on the platform, device characteristics, input data and even across iterations on the same hardware/software. We execute each operation 5 times and measure the time taken in each run. Our Time Oracle implementation chooses the minimum of all measured runs for a given op as the time for that op.

Ordering Wizard: This module is responsible for assign-

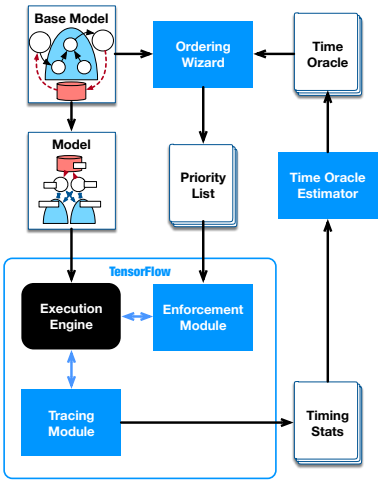


Figure 6: System Design. Components of our system are in blue sharp-edged rectangles.

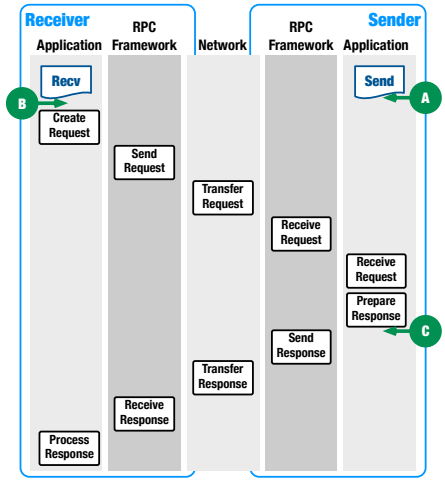


Figure 7: Life time of a network transfer.

ing priorities to recv ops on a single worker. The schedule may be computed based on TIC or TAC. In TAC, the ordering module relies on the time estimated by the time oracle. In TIC, the order is determined based on the DAG alone. The estimated priorities are sent to the enforcement module. The priority list is calculated offline before the execution; all iterations follow the same order.

Enforcement Module: This module takes as input the priority list computed by the ordering module and enforces this order on the network transfers per worker.

5.1 Implementation

We implement our system over TensorFlow 1.8. We describe our implementation in detail.

Time Oracle: We use the TensorFlow internal tracer to measure the time of computation ops. We extend the capability (115 LOC C++) of this tracer to collect information on network transfer at all workers. Our code is publicly available (https://github.com/xldrx/tictac).

Ordering Wizard: We implement TIC and TAC as offline analyzers (250 LOC in Python). The implementation takes time oracle and base model in the TensorFlow DAG format and generates the priority of recv ops.

Enforcing: The enforcement module is implemented over the gRPC submodule of TensorFlow (40LOC in C++).

gRPC provides one channel per worker-PS pair with all transfers between the pair sent to the same queue. Only one transfer can be active at a given moment for each channel. A network transfer over gRPC in TensorFlow involves multiple stages as shown in Figure 7. When a $recv$ op is activated at the receiver, it sends a request for transfer to the sender. If the $send$ op is also active at the sender, the transfer may be initiated by gRPC. In this dataflow, there are three possible candidate locations for enforcing ordering — at the receiver before the request is initiated, at the sender before the $send$ op is activated or at the sender before the transfer is sent to gRPC. Alternatively, this may also be enforced as a direct dependency in the DAG.

We implement the enforcement module at the sender, i.e. the PS, before the transfer is sent to gRPC. This choice is guided by several practical concerns. Enforcing directly on the DAG is conservative since each transfer has to wait for the completion of the previous transfer. This prevents pipelining and drastically reduces the communication throughput. Ordering the activation of recv or $send$ ops is not sufficient since it could change throughout the data flow. For example, a larger transfer request may take longer to reach the response state on the sender side. During this interval, a smaller transfer with lower priority may catch up.

For the purpose of enforcement, the priorities are sequentially assigned to an integer in the range of $[0, n)$. Thus, the priority number of a transfer represents the number of transfers that have to complete before it. The sender (PS server) maintains a counter for each worker per iteration which is incremented when a corresponding transfer is handed to the gRPC. Before a transfer is handed to the gRPC, it is blocked until the corresponding counter reaches the normalized priority number.

During experiments, we notice that gRPC may not always process transfers in the order they are queued. This affects the performance of our ordering in some cases. However, the number of such occurrences at the gRPC level are very few. In Inception model (one of the tested models), this error was 0.5% in TIC and 0.4% in TAC.

6 RESULTS

In this section, we evaluate TicTac under a wide range of inputs/system parameters to answer the following questions:

- How does TicTac perform with scale out of workers?
- How is TicTac affected by the number of parameter servers?
- How does the benefits accrued with TicTac change with the communication and computation cost?
- How well do the proposed heuristics perform in terms of consistency and straggler mitigation?

Setup: We use in-graph replication for Distributed TensorFlow (Google) with synchronized training and synthetic input data.

We test TicTac under two environments. (a) **Cloud GPU environment(env_G):** We use Standard NC6 virtual machines (6 cores, 56 GB memory, 1 X Nvidia K80 GPU with 12GB memory) on Azure cloud environment. For parameter servers we used Standard F64s v2 (CPU Only, 64 cores, 128 GB memory). (b) **High-end CPU cluster (env_C):** We use a commodity cluster (32 core, 64GB memory, 1GbE network). In both environments, we test 2 to 16 workers and 1 to 4 PS. For understanding the impact of batch size, we test the networks with the standard batch size multiplied by factors [0.5, 1, 2]. We tested our method on 10 well-known models (details of models in Table 1 in Appendix).

We evaluate the performance under two workloads: training and inference. In training, we use Stochastic Gradient Descent (SGD) as optimizer. The training workload is identical to the training jobs used in practice. We emulate the inference workload of agents in reinforcement learning with online training. In this environment, parameter servers store the parameters which are updated by a set of training worker nodes (which we do not consider in the inference workload). The inference agents are responsible for reading the parameters from the PS and running the inference (this is the phase we evaluate in this workload).

In each test, we discard the first 2 iterations to limit the warm-up effect (initialization of GPUs, cache etc). This is necessary since the first iteration takes much longer compared to the rest. We record the next 10 iterations. For throughput, we report the mean across 10 iterations; for straggler effect and overlap coefficient we report the maximum. Computing the TIC and TAC heuristics takes approximately 10 seconds. Note that these heuristics are computed *before* the training/inference begins. Hence, this will not add overhead during the execution.

We use Imagenet Dataset for our experiments. We evaluated both synthetic and real data and observed less than 3% difference in iteration time on a single machine. The data is read in the TFRecord format from a shared NFS-connected Azure storage, samples are resized, augmented, and prefetched during training. TicTac does not alter the computational flow of the model; it only chooses one of the feasible orders of network transfers. Hence, it does not affect the accuracy of training (shown in Figure 8).

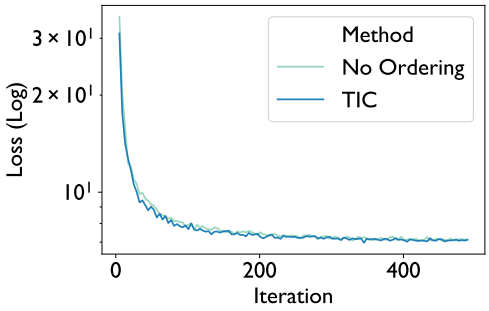


Figure 8: Loss value throughout the first 500 iterations of training InceptionV3 on ImageNet.

Next, we compare the performance metrics across various heuristics. Specifically, we evaluate throughput, overlap coefficient, and prevalence of stragglers (slow workers that force others to wait, thereby increasing the iteration time). Performance of TIC is only marginally worse compared to TAC (shown in Figure 15 in Appendix). This indicates that, for current models, DAG-level information is sufficient for obtaining a near-optimal scheduling. However, we expect the gap between TIC and TAC to increase as complexity of models increases.

We attempted to compare TicTac with Poseidon (Zhang et al., 2017). However, only the binaries of Poseidon are publicly available. In our experiments, Poseidon performed extremely poorly compared to TicTac, and even vanilla TensorFlow 1.8. Since Poseidon is based on older version of TensorFlow (TensorFlow 1.0) and CUDA (8.0), we were unable to account the poor performance to their methodology. Hence, we exclude the results since the comparison is inconclusive. Additionally, since order extraction is not explained in their paper, we were unable to reimplement their strategy.

6.1 Throughput

Scaling the number of workers: In Figure 9, we evaluate the impact of scaling the number of workers with the number of PS to workers fixed to the ratio 1:4. We obtain up to 37.7% of speed up in throughput across networks. The gains are measured relative to the baseline — no scheduling. Larger networks have higher performance gains. The speed up depends on two factors — communication load and extent of overlap. As the number of workers increases, the communication load increases in PS. When the com-

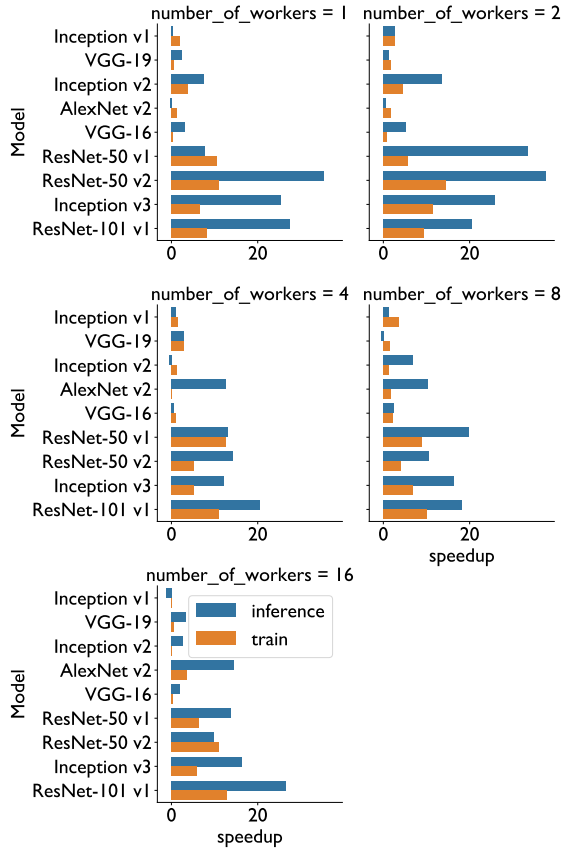


Figure 9: Impact of scaling the number of workers on throughput. The gains are measure with respect to the baseline (no scheduling). Measured on env_G with PS:Workers in the ratio 1:4.

munication load increases, scheduling can provide benefits through better overlap until a threshold. When the communication load is much higher than the computation load, the impact of overlap diminishes. Hence, beyond this threshold, the benefits accrued with scheduling reduces. This threshold varies across models. Also, the gains are measured with respect to the baseline which chooses a random schedule, leading to variations in performance. Hence, we observe varying trends across networks based on the network-specific characteristics. In small networks, with small number of workers and parameter servers, the overhead associated with scheduling may overshadow the benefits of better overlap. In such rare cases, we observe a slow down of up to 4.2%. This shows that scheduling network transfers may be disabled in small networks at small training and inference sizes.

Scaling the number of Parameter Servers: In Figure 10, we evaluate the impact of scaling the number of parameter servers with 8 workers in env_G (Cloud with GPU) across various networks. In general, we obtain higher gains in the inference phase than training. Even in the presence of

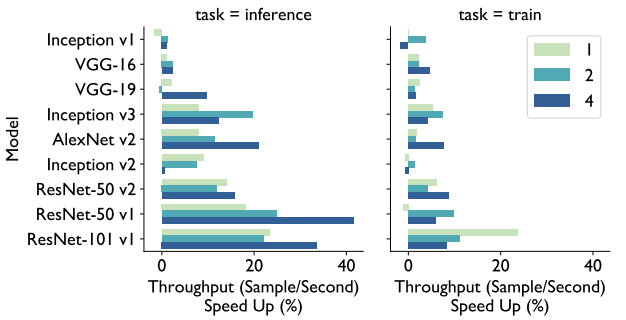


Figure 10: Impact of scaling the number of Parameter Servers on env_G cloud GPU environment with 8 workers.

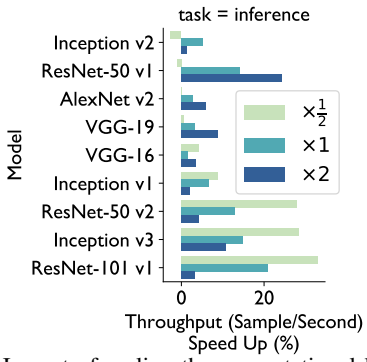


Figure 11: Impact of scaling the computational load on env_G cloud GPU environment with 4 workers.

multiple parameter servers, enforcing ordering with TicTac provides significant performance improvement.

Scaling the computational load: In Figure 11, we show the impact of varying computational load by testing each model with the prescribed batch size multiplied by three factors — 0.5, 1, 2. There are two factors affecting the scaling of computation load — computation time and opportunity for overlap. The relative ratio of communication and computation determines the opportunity for overlap. As the batch size increases, the computation time increases. If the communication time is higher (compared to the computation time), increase in computation time increases the opportunity for overlap. If communication time is smaller than computation time, scaling will reduce throughput as the opportunity for overlap reduces.

Scalability with network size:: We show the improvement in throughput (samples/second) achieved with TIC compared to the baseline with no scheduling in Figure 12. We observe that larger networks obtain higher gains. This can be attributed to larger variance in parameter transfer orders in larger DAGs in the absence of scheduling.

6.2 Overlap Coefficient

To validate the overlap coefficient metric, we run training of Inception v2 1000 times each with and without the scheduling algorithm, TAC in env_C. The overlap coeffi-

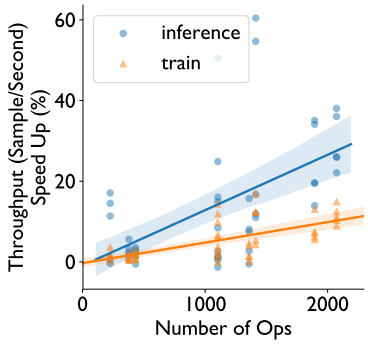


Figure 12: Throughput speedup with training and inference as a function of DAG size represented in number of ops

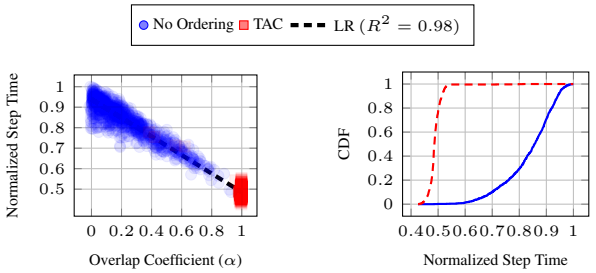


Figure 13: In env_C, on Inception v2, (a) Regression test of Scheduling Efficiency and Normalized Step Time, (b) Step Time Comparison across Scheduling Mechanisms.

cient can predict step time accurately, with a high R^2 score of 0.98, as seen in Figure 13 (a). This proves that most of the variation in iteration time arises from random schedules in parameter transfers. We also observe that in the absence of enforced scheduling, the step time and overlap coefficient have a large variance. With scheduling, the step time is reduced and the variance is minimal. Moreover, most runs have an overlap coefficient approaching 1, indicating near-optimal scheduling in TAC.

6.3 Performance Consistency

In Figure 13 (b), we compare the consistency in performance obtained with and without scheduling (TAC) in inference on InceptionV2 with 1000 runs in env_C. We see that TAC has consistent performance, denoted by a sharp curve in the CDF. The baseline (no scheduling), on the other hand, has a large variance. For comparison, 95^{th} percentile of normalized step time in the baseline and TAC are respectively 0.63403 and 0.99825.

Straggler Effect: : Performance inconsistency creates straggling worker effect when multiple workers have different makespan. As a result, all workers have to wait for the slowest one. We quantify the straggler time as the maximum time spent by any worker in waiting to the total iteration time (represented in percentage).

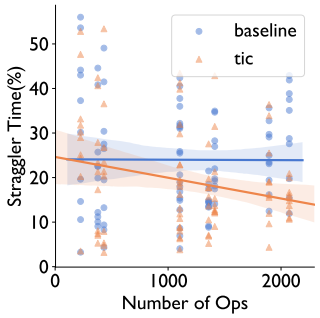


Figure 14: Effect of stragglers with TIC in the GPU enviornment, env_G.

In Figure 14, we show the impact of stragglers. Straggler effect is caused by two factors: system-level performance variations and efficiency of scheduling on individual workers. In the baseline, workers follow arbitrary scheduling. Hence, a worker with a bad order forces other workers into a long wait, more than 50% of the total iteration time in some cases. On average, scheduling limits straggler effect with larger benefits in bigger DNNs (higher number of ops). Enforcing *any* order reduces straggler effect regardless of the quality of the chosen order.

7 Conclusion

In this work, we elucidate the importance of communication scheduling in distributed deep learning systems. We devised a metric for quantifying the efficiency of a given schedule of data transfers and developed two heuristics for efficient scheduling. Through extensive testing of these heuristics across a variety of workloads, we demonstrated that significant gains are achievable through communication scheduling. For a typical DNN training which runs for days to weeks, 20% improvement in iteration time can save significant compute power.

Our study encourages further research in network scheduling for parameter server as well as other unexplored aggregation techniques such as all reduce. In future, we can also take into account additional metrics such as congestion from the network fabric for better network performance. These results also provide motivation for extending the scheduling to additional resources types such as memory and storage.

8 Acknowledgement

We thank Paul Barham and Brighten Godfrey for their feedback. This material is based on work supported by the National Science Foundation under Grant No. 1725729. Azure Cloud resources used in this paper is provided through Microsoft Azure Sponsorship.

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 *OSDI*, volume 16, pp. 265–283, 2016.

Akiba, T., Suzuki, S., and Fukuda, K. Extremely Large Minibatch SGD: Training ResNet-50 on ImageNet in 15 Minutes. *CoRR*, abs/1711.04325, 2017. URL http://arxiv.org/abs/1711.04325.

Alistarh, D., Grubic, D., Li, J., Tomioka, R., and Vojnovic, M. QSGD: Communication-Efficient SGD via Gradient Quantization and Encoding. In *Advances in Neural Information Processing Systems*, pp. 1707–1718, 2017.

Amodei, D., Anubhai, R., Battenberg, E., Case, C., Casper, J., Catanzaro, B., Chen, J., Chrzanowski, M., Coates, A., Diamos, G., Elsen, E., Engel, J., Fan, L., Fougner, C., Han, T., Hannun, A. Y., Jun, B., LeGresley, P., Lin, L., Narang, S., Ng, A. Y., Ozair, S., Prenger, R., Raiman, J., Satheesh, S., Seetapun, D., Sengupta, S., Wang, Y., Wang, Z., Wang, C., Xiao, B., Yogatama, D., Zhan, J., and Zhu, Z. Deep Speech 2: End-to-End Speech Recognition in English and Mandarin. *CoRR*, abs/1512.02595, 2015. URL http://arxiv.org/abs/1512.02595.

Arnold, S. An Introduction to Distributed Deep Learning. https://seba1511.com/dist_blog/, 2016.

Brucker, P. and Knust, S. Complexity results for scheduling problems, 2007.

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.

Cho, M., Finkler, U., Kumar, S., Kung, D., Saxena, V., and Sreedhar, D. PowerAI DDL. *arXiv preprint arXiv:1708.02188*, 2017.

Courbariaux, M., Bengio, Y., and David, J.-P. Binaryconnect: Training deep neural networks with binary weights during propagations. In *Advances in neural information processing systems*, pp. 3123–3131, 2015.

Cui, H., Tumanov, A., Wei, J., Xu, L., Dai, W., Haber-Kucharsky, J., Ho, Q., Ganger, G. R., Gibbons, P. B., Gibson, G. A., et al. Exploiting iterative-ness for parallel ML computations. In *Proceedings of the ACM Symposium on Cloud Computing*, pp. 1–14. ACM, 2014.

Cui, H., Zhang, H., Ganger, G. R., Gibbons, P. B., and Xing, E. P. GeePS: Scalable deep learning on distributed GPUs with a GPU-specialized parameter server. In *Proceedings of the Eleventh European Conference on Computer Systems*, pp. 4. ACM, 2016.

Google. Distributed tensorflow. https://www.tensorflow.org/deploy/distributed.

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.

Gupta, S., Agrawal, A., Gopalakrishnan, K., and Narayanan, P. Deep learning with limited numerical precision. In *International Conference on Machine Learning*, pp. 1737–1746, 2015.

He, K., Zhang, X., Ren, S., and Sun, J. Deep residual learning for image recognition. *CoRR*, abs/1512.03385, 2015. URL http://arxiv.org/abs/1512.03385.

He, K., Zhang, X., Ren, S., and Sun, J. Identity mappings in deep residual networks. *CoRR*, abs/1603.05027, 2016. URL http://arxiv.org/abs/1603.05027.

Hoque, I. and Gupta, I. LFGraph: Simple and fast distributed graph analytics. In *Proceedings of the First ACM SIGOPS Conference on Timely Results in Operating Systems*, pp. 9. ACM, 2013.

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

Ioffe, S. and Szegedy, C. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *CoRR*, abs/1502.03167, 2015. URL http://arxiv.org/abs/1502.03167.

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.

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

Malewicz, G., Austern, M. H., Bik, A. J., Dehnert, J. C., Horn, I., Leiser, N., and Czajkowski, G. Pregel: a system for large-scale graph processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, pp. 135–146. ACM, 2010.

Paszke, A., Gross, S., Chintala, S., and Chanan, G. PyTorch: Tensors and dynamic neural networks in Python with strong GPU acceleration, 2017.

Pinedo, M. L. *Scheduling: Theory, Algorithms, and Systems*. Springer Publishing Company, Incorporated, 3rd edition, 2008. ISBN 0387789340, 9780387789347.

Sergeev, A. and Balso, M. D. Horovod: fast and easy distributed deep learning in tensorflow. *CoRR*, abs/1802.05799, 2018. URL http://arxiv.org/abs/1802.05799.

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

Sridharan, S., Vaidyanathan, K., Kalamkar, D., Das, D., Smorkalov, M. E., Shiryaev, M., Mudigere, D., Mellempudi, N., Avancha, S., Kaul, B., et al. On Scale-out Deep Learning Training for Cloud and HPC. *arXiv preprint arXiv:1801.08030*, 2018.

Szegedy, C., Liu, W., Jia, Y., Sermanet, P., Reed, S. E., Anguelov, D., Erhan, D., Vanhoucke, V., and Rabinovich, A. Going deeper with convolutions. *CoRR*, abs/1409.4842, 2014. URL http://arxiv.org/abs/1409.4842.

Szegedy, C., Vanhoucke, V., Ioffe, S., Shlens, J., and Wojna, Z. Rethinking the inception architecture for computer vision. *CoRR*, abs/1512.00567, 2015. URL http://arxiv.org/abs/1512.00567.

Vanhoucke, V., Senior, A., and Mao, M. Z. Improving the speed of neural networks on CPUs. In *Proc. Deep Learning and Unsupervised Feature Learning NIPS Workshop*, volume 1, pp. 4. Citeseer, 2011.

Wen, W., Xu, C., Yan, F., Wu, C., Wang, Y., Chen, Y., and Li, H. Terngrad: Ternary gradients to reduce communication in distributed deep learning. In *Advances in Neural Information Processing Systems*, pp. 1508–1518, 2017.

Xin, R. S., Gonzalez, J. E., Franklin, M. J., and Stoica, I. Graphx: A resilient distributed graph system on spark. In *First International Workshop on Graph Data Management Experiences and Systems*, pp. 2. ACM, 2013.

You, Y., Zhang, Z., Hsieh, C., and Demmel, J. 100-epoch ImageNet Training with AlexNet in 24 Minutes. *CoRR*, abs/1709.05011, 2017. URL http://arxiv.org/abs/1709.05011.

Zhang, H., Zheng, Z., Xu, S., Dai, W., Ho, Q., Liang, X., Hu, Z., Wei, J., Xie, P., and Xing, E. P. Poseidon: An Efficient Communication Architecture for Distributed Deep Learning on GPU Clusters. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, pp. 181–193, Santa Clara, CA, 2017. USENIX Association. ISBN 978-1-931971-38-6. URL https://www.usenix.org/conference/atc17/technical-sessions/presentation/zhang.

APPENDIX

A DNN MODELS

In Table 1, we present the model characteristics of 10 deep learning models used in our evaluation. The number of parameters, total size of all parameters, number of computational operations in inference mode and training mode, and the standard batch size are given below.

Neural Network Model	#Par	Total Par Size (MiB)	#Ops Inference/ Training	Batch Size
AlexNet v2 (Krizhevsky, 2014)	16	191.89	235/483	512
Inception v1 (Szegedy et al., 2014)	116	25.24	1114/2246	128
Inception v2 (Ioffe & Szegedy, 2015)	141	42.64	1369/2706	128
Inception v3 (Szegedy et al., 2015)	196	103.54	1904/3672	32
ResNet-50 v1 (He et al., 2015)	108	97.39	1114/2096	32
ResNet-101 v1 (He et al., 2015)	210	169.74	2083/3898	64
ResNet-50 v2 (He et al., 2016)	125	97.45	1423/2813	64
ResNet-101 v2 (He et al., 2016)	244	169.86	2749/5380	32
VGG-16 (Simonyan & Zisserman, 2014)	32	527.79	388/758	32
VGG-19 (Simonyan & Zisserman, 2014)	38	548.05	442/857	32

Table 1: DNN model characteristics

B TIC VS. TAC

In Figure 15, we plot the increase in throughput achieved with scheduling in env_C with and without the scheduling schemes (TIC and TAC). We observe that both TIC and TAC offer significant speedup compared to the baseline (no scheduling). Performance of TIC is comparable to that of TAC indicating that we can achieve improved performance without relying on runtime statistics in current models.

Due to the simplicity of TIC algorithm, we use it as the representative algorithm for scheduling in the cloud GPU environment (env_G).

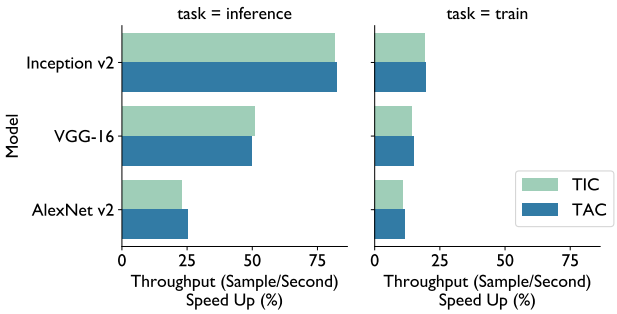


Figure 15: Increase in throughput with the scheduling schemes (TIC and TAC) compared to the baseline (no scheduling). Measured on env_C (CPU-Only).