Supplementary Material
AutoSync: Learning to Synchronize for Data-Parallel Distributed Deep Learning

Hao Zhang†, Yuan Li†, Zhijie Deng, Xiaodan Liang, Lawrence Carin, Eric P. Xing
Petuum Inc., Carnegie Mellon University, Duke University, Tsinghua University
{hao.zhang,christy.li,xiaodan.liang,eric.xing}@petuum.com, dzj17@mails.tsinghua.edu.cn, lcarin@duke.edu

The supplementary material is organized as follows:

- §1 explains the rationale of estimating $T$ using $T = \max(T_{\text{comp}}, T_{\text{sync}})$ (Section 3.1 of the main paper).
- §2 provides more detailed explanations about the predefined modeling of $T^P_{\text{server}}$ and $z^P$ in addition to the contents in Section 3.1 of the main paper.
- §3 provides details on the predefined modeling for $T^C$ and the extraction of domain-agnostic features $z^C$ (corresponding to Section 3.1 of the main paper).
- §4 gives detailed formulations of the RNN simulator (Section 3.1 of the main paper).
- §5 gives detailed formulations of the GAT simulator (Section 3.1 of the main paper).
- §6 provides detailed algorithms for the two knowledge constraints $c_{lb}$ and $c_{am}$ (Section 3.2 of the main paper).
- §7 elaborates the method for selecting the final set of candidate strategies for trial execution, and the approaches to estimates similarities between strategies (Section 3.2 of the main paper).
- §8 gives details about the AutoSync system implementations and baseline systems information (Section 5 of the main paper).
- §9 lists all the 11 models ($\mathcal{G}$) and 14 cluster specifications ($\mathcal{D}$) we have experimented with throughout our research, and provides information about the public dataset we are going to release. (Section 5 of the main paper).
- §10 provides some hyperparameter information about the simulator training.

1 Rationale of $T = \max(T_{\text{comp}}, T_{\text{sync}})$

In the main paper, we model the per-iteration runtime $T$ of parallelizing $\mathcal{G}$ on $\mathcal{D}$ using two contributing components: computation time $T_{\text{comp}}$, and parameter synchronization time $T_{\text{sync}}$, and obtain the per-iteration time via:

$$T = \max(T_{\text{comp}}, T_{\text{sync}}).$$  (1)

The rationale behind Eq. 1 is as follows: (1) Since many runtime systems (e.g. TensorFlow [1] or PyTorch [3]) introduce scheduling or parallelization between communication and computation, in practice, there are significant overlaps between the two components; (2) in data-parallel training, it is commonly observed that one component usually dominates the other [4]. These make using the maximum of them as the estimation of the total time reasonable.

2 More Explanations for $T_{PS}^{\text{server}}$

For modeling $T_{\text{sync}}$ of variables, we have introduced two hyperparameters: network overhead denoted as $\phi$, and GPU kernel memory latency denoted as $\delta$ which includes device-host transfer time and GPU kernel overhead. As $\delta$ is relatively slow compared with other time consumption, we treat it as a constant that does not scale with variable size. Synchronizing $v_i$ via PS ($v_i \in V_{PS}$) involves: (1) workers send gradients to servers, (2) servers update parameters, (3) servers send the updated parameters to workers, where (2) is negotiable and (1)(3) are symmetric processes and cost the same amount of time. We denote the original size (e.g. byte size) of the gradient of the variable $v_i$ as $m_i$ (note $m_i$ takes into consideration the sparsity of the gradients when applicable), and assume it will apply the encoding/decoding scheme $c_i \in s_i$, so the actual size of the message (related to $v_i$) to be transferred across devices is $c_i(m_i)$, where we also use $c_i$ to denote the compression function that reduces the original size $m_i$ to $c_i(m_i)$.

Let $w_i$ denote the number of workers involved in synchronizing $v_i$. The parameter transfer process involves transferring data between the GPU device memory and the host memory (RAM) within the same machine, and between the host memory across machines. The first process introduces GPU kernel latency and device-host communication. The second process introduces network overhead (e.g., latency) and network communication. Hence, the communication time $T_{\text{server}}^{PS}$ on the server hosting $v_i$, indexed as $j$, is

$$T_{\text{server}}^{PS}(v_i) = \sum_{k=1}^{w_i} \|d(j,k) \cdot r_{i,k}^p \|_{b_{j,k}} + \sum_{k=1}^{w_i} \|d(j,k) \cdot r_{i,k}^p \|_{c_i(m_i)} + \phi + \delta,$$

where $r_{i,k}$ is the number of replicas of $G$ on worker $k$ if the worker $k$ has multiple GPUs that host multiple replicas of $G$ respectively; $\|d(j,k) \cdot r_{i,k}^p \|_{b_{j,k}}$ and $\|d(j,k) \cdot r_{i,k}^p \|_{c_i(m_i)}$ are true when server $j$ and worker $k$ locate on different machines and when hierarchical reduction is used, respectively. To interpret Eq. 2, the first term corresponds to sending messages from each worker $k$ to the server $j$ (and vise versa). The second term captures network overheads that scale linearly with the number of workers, or with the number of replicas when $\|d(j,k) \cdot r_{i,k}^p \|_{b_{j,k}}$ is false. The third term captures the constant GPU memcpy latency.

The estimation $T_{\text{server}}^{PS}(v_i)$ can be accumulated across all $v_i \in V_{PS}$ to obtain the total synchronization time taken for all PS variables. We also derive the domain-agnostic features $z^{PS}$ based on Eq. 2, which has been elaborated in the main paper.

3 Predefined Modeling for $T_{\text{sync}}^{CC}$ and Domain-agnostic Features $z^{CC}$

In the main paper and §2, we have demonstrated the predefined modeling for communicating variables which adopt the PS communication architecture ($v_i \in V_{PS}$). Next, we develop the model for variables using collective communication (CC).

For $v_i \in V_{CC}$, we model 5 mostly used collective primitives: AllReduce, ReduceScatter, AllGather, Broadcast and Reduce [12]. Take an example when there are $w$ workers and the device order in the substrategy $s_i$ is a ring [14]. Each primitive sends and receives $\frac{2(w-1)}{w}$, $\frac{w-1}{w}$, $\frac{w-1}{w}$, 1, 1 times, respectively, in its applicable scenario for parameter synchronization (e.g. AllReduce for dense gradients or AllGather for sparse gradients [14]). Therefore, we can obtain $T_{\text{sync}}^{CC}(v_i)$ using the following formula:

$$T_{\text{sync}}^{CC}(v_i) = \mathbb{I}_1 \frac{2(w_i - 1)c_i(m_i)}{w_ib_m} + \mathbb{I}_2 \frac{(w_i - 1)c_i(m_i)}{w_ib_m} + \mathbb{I}_3 \frac{c_i(m_i)}{b_m} + w_i \cdot \phi + \delta,$$

where $b_m = \min_{(k_1,k_2) \in \text{ring}} b_{k_1,k_2}$ denotes the lowest bandwidth between devices in the ring, since the throughput of a ring is restricted by the lowest bandwidth in the network [19]. $\mathbb{I}_1, \mathbb{I}_2, \mathbb{I}_3$ are true when AllReduce, ReduceScatter and AllGather, Broadcast and Reduce are activated, respectively. The formula is derived based on counting how many times each message (i.e. gradients) needs to be passed across the ring, taking into considerations both network transfer overhead as well as the device-host memory swap latency. The total synchronization time for variables assigned with collective communication is $T_{\text{sync}}^{CC}(V_{CC}) = \sum_{i=1}^{\mid V_{CC} \mid} T_{\text{sync}}^{CC}(v_i)$. 

2
In a similar way as in §2, the domain-agnostic features for \( v_i \in V^{CC} \) are \( z_i^{CC} = [\text{network transfer}, \text{coefficient of } \phi, \text{coefficient of } \delta] \) from Eq. 3, and the global features of \( T_{sync}^{CC} \) are \( z^{CC} = \max\{z_i^{CC} \text{ for } v_i \in V^{CC}\} \oplus \sum\{z_i^{CC} \text{ for } v_i \in V^{CC}\}/|V^{CC}|. \)

4 Detailed Formulation for the RNN-based Simulator

The formulation of applying RNN with both predefined and raw features on runtime prediction can be written as

\[
\begin{align*}
    z_i &= \text{MLP}_i(z_i^{\text{raw}} \oplus z_i^{\text{pre}}), \\
    h_{i+1} &= \text{LSTM}(h_i, z_i), \\
    h'_{i+1} &= \text{LSTM}(h'_i, z_i \mid_{V'_{G,\theta}}^{\text{pre}}), \\
    \hat{f} &= \text{MLP}_o([h_1 \mid_{V'_{G,\theta}}, h'_1 \mid_{V'_{G,\theta}}]),
\end{align*}
\]

where \( \text{MLP}_i \) converts input features to a hidden representation, and \( \text{MLP}_o \) converts encoded features after LSTM fusion to the output logits. Note that the LSTM walks through each \( v_i \in V'_{G,\theta} \) strictly following their original forward (backward) order in the computational graph, so as to inject this information into the modeling. Note the RNN works on the transformed graph \( \{V'_{G,\theta}, E'_{G}\} \), which additionally has partitioned variables compared to the original graph \( \{V_{G,\theta}, E_{G}\} \).

5 Detailed Formulation for the GAT-based Simulator

The formulation of applying GAT with both predefined and raw features on runtime estimation can be written as

\[
\begin{align*}
    \{z_i\}_{i=1}^{\mid V'_{G,\theta}} &= \text{GAT}((z_i^{\text{raw}} \oplus z_i^{\text{pre}}, \text{ for } v_i \in V'_{G,\theta}), E'_{G}), \\
    z &= \frac{1}{\mid V'_{G,\theta}} \sum((z_i)_{i=1}^{V'_{G,\theta}}), \\
    \hat{f} &= q(z),
\end{align*}
\]

where \( \text{GAT} \) denotes the graph attention operations \([18]\) applied on the graph \( \{V'_{G,\theta}, E'_{G}\} \) with \( z_i^{\text{raw}} \oplus z_i^{\text{pre}} \) as node features. \( q \) is a linear layer that converts hidden representation to runtime estimation.

6 Details of the Knowledge Constraints \( c_{lb} \) and \( c_{am} \)

The two knowledge constraints, load balancing constraint \( c_{lb} \) and adjacent merging constraint \( c_{am} \), are implemented together with the strategy sampler as regulations. In particular, the strategy sampler first generates a partial strategy \( S \), leaving two fields, variable placement (PS-based) and the group (collective-based) undecided. A global load balancer \( (c_{lb}) \) and group assigner \( (c_{am}) \) assign their values using randomized and approximate solutions, illustrated in Algorithm 1 and Algorithm 2, respectively. Note both constraints work on the set of variables \( V'_{G,\theta} \) with partitioned shards, instead of the original set of variables \( V_{G,\theta} \).

The two constraints help filter strategies that are obviously inefficient, but approximately maintain global load balance on the cluster and adjacent merging structures correlated with the forward-backward propagation order. Their effectiveness has been validated in §5.1 in the main paper.

7 Select Final Candidate Strategies for Trial Execution

This section provides details on the selection of the final set of strategy candidates for trial execution. As stated in §3.1 of the main paper, we minimize a weighted sum of \( \hat{f} \) and internal similarity to select the final set of strategy candidates for trial execution. Formally, we solve the following optimization
Algorithm 1: Implementation of the load balance constraint \( c_{lb} \)

**Inputs:** \( S, D, V'_{G,\theta} \)

**Function** load_balance_constraint\( (S, D, V'_{G,\theta}) : \)

1. Set loads of each device in \( D \) as \( l_{cur} = [0, ..., 0] \)
2. Sum up the bandwidth of devices in \( D \): \( b_{all} = \sum_{i=1}^{|D|} b_{i,i} \)
3. Sort \( V'_{G,\theta} \) by byte size in descending order and get \( V'_l \)
4. for \( v \) in \( V'_{G,\theta} \) do
   5. \( l_{all} = \sum_{i=1}^{|D|} l_{cur}[i] \)
   6. \( logits = l_{all} * [b_{1,1}/b_{all}, ..., b_{|V'_{G,\theta}|}/b_{all}] - l_{cur} \)
   7. \( j \sim \text{Categorical(softmax(logits))} \)
   8. Set the placement of \( v \) in \( S \) as \( d_j \in D \)
   9. \( l_{cur}[j] = l_{cur}[j] + \text{byte_size}(v) \)

Algorithm 2: Implementation of the adjacent merge constraint \( c_{am} \)

**Inputs:** \( D, V'_{G,\theta}, G \), the number of total collective merging groups \( N \)

**Function** adjacent_merge_constraint\( (D, V'_{G,\theta}, S, N) : \)

1. Sort \( V'_{G,\theta} \) based on their location in the backpropagation order indicated by \( G \), from input to loss function, and get \( V'_G \)
2. Set loads \( l_{cur} = [0, ..., 0] \) corresponding to \( N \) merge groups
3. \( l_{avg} = \frac{1}{N} \sum_{v \in V'_G} \text{byte_size}(v) \)
4. \( g = 0 \)
5. for \( v \) in \( V'_G \) do
   6. if \( l_{cur}[g] \geq l_{avg} \) then
      7. \( g = g + 1 \)
   8. if \( l_{cur}[g] < l_{avg} \) then
   9. Set the group of \( v \) in \( S \) as \( g \)
  10. \( l_{cur}[g] = l_{cur}[g] + \text{byte_size}(v) \)

problem:

\[
\min_{\{S_i\}_{i=1}^M} \sum_{i=1}^M \tilde{f}(S_i) + \alpha \sum_{i=1}^K \sum_{j=1}^K \text{sim}(S_i, S_j), \tag{11}
\]

where \( \{S_i\}_{i=1}^M \) are the \( M \) qualified strategies filtered first using the simulator score, \( \alpha \) denotes a trade-off coefficient, and sim is a pairwise similarity function between strategies. Solving Eq. 11 helps deliver a set of candidates that trade off between low predicted runtime (exploitation) and low similarity (exploration). The above problem is a typical problem of submodular minimization, and we resort to the greedy algorithm for an approximate solution [10].

**Estimating the similarity between strategies.** Since we do not have a continuous representation of strategies, we have developed two approaches to estimate the similarities between strategies. The first approach estimates the similarity of the two strategies \( S_1, S_2 \) by comparing each sub strategy \( s_1, s_2 \in S_1 \) and \( s_2, s_2 \in S_2 \) corresponding to the variable \( v_i \), and counting how many choices of each synchronization-affecting factor are the same, and use the final count as a measure of their similarity (higher is more similar). The second approach takes the cosine similarity between the hidden outputs of the simulator, whose inputs are \( S_1 \) and \( S_2 \), as a proxy of the strategy similarity. Empirically, we find the two approaches perform similarly, but the first similarity function does not depend on the simulator thus can be used when no trained simulator is available (e.g. for the baseline AutoSync(-s)).
8 More Details on AutoSync Implementations and Baseline Systems

In this section, we provide more details about the experimental setup, including the implementations of AutoSync backend for strategy generation and application, and the details about the two baseline systems: Horovod and PS.

AutoSync System Implementation. Frameworks such as TensorFlow offer a flexible distributed runtime [1, 20] for evaluating dataflow graphs on distributed clusters [1, 20]. On top of TF 2.0, we have built a composable dataflow graph rewriting system, called AutoDist (https://github.com/petuum/autodist), that allows composing different synchronization aspects, such as synchronizer, placement and partitioning of variables, etc. as synchronization strategies. More precisely, given the single-node dataflow graph \( G \) extracted from user code, AutoDist can rewrite \( G \) by modifying nodes, edges and their attributes, into a distributed graph \( G' \) with all required semantics from \( S \). It then launches TensorFlow distributed runtime to execute \( G' \) on the cluster of nodes from \( R \). This allows evaluating different \( S \) using a common distributed engine and eliminating performance variations caused by different runtime. We skip the detailed descriptions about how AutoDist is designed as they are out of the scope of this paper.

Based on AutoDist, AutoSync is implemented a synchronization strategy auto-optimizer component therein. The specific code of AutoSync can be found at https://github.com/petuum/autodist/tree/master/autodist.

System Setup. All the baselines and AutoSync rely on TensorFlow 2.0 as the distributed runtime. For the experiment results reported in this paper, we compile TF2.0 with CUDA10.0, CUDNN 7.1, and NCCL 2.4.7, and uses gRPC for network communication.

Baseline Systems. In the main paper we introduce two strong hand-optimized synchronization systems as external baselines. We provide more details about the two systems.

Horovod [14] is one of the most adopted open source synchronization systems for data-parallel distributed ML. It uses AllReduce (AllGather) to synchronize dense (sparse) gradients of all model variables, and utilizes BO to autotune the merge scheme for multiple collective operations based on collected trial data in warm-up runs. Details of the optimization can be found at https://github.com/horovod/horovod/blob/master/docs/autotune.rst. Per our experiments on our cluster setup, it reports up to 2x speedup than a Google-provided parameter server implementation based on distributed TensorFlow as runtime (details at https://github.com/tensorflow/examples/blob/master/community/en/docs/deploy/distributed.md), on multiple CNNs such as ResNet101 and InceptionV3. Horovod uses NCCL 2.4.7 for collective communication, which is the same with AutoSync.

PS is a highly tuned parameter server implementation with multiple optimizations from recent PS literature incorporated. We elaborate a few notable optimizations: (1) it maintains load balance by partitioning large variables and evenly placing the shards across servers depending on their available bandwidth (i.e. correlates to its maximum bandwidth and the current load as a parameter server); (2) it uses the BO algorithm to decide the partitioning size following ByteScheduler [13]; (3) it communicates sparse gradients (IndexSlices in TensorFlow) using Gather and Scatter primitives (instead of Reduce and Broadcast) [9] to reduce communication overhead; (4) it performs hierarchical Reduce (or Gather) and Broadcast (or Scatter) on nodes with more than 1 GPUs to reduce network traffic. The optimizations are implemented on top of TensorFlow 2.0 as well, so the backend also relies on distributed TensorFlow 2.0 for distributed execution, same with AutoSync.

9 Descriptions of the Experiment Setup and Public Dataset

9.1 Model and Cluster Specifications

Throughout our research in this paper, we have conducted experiments on 10 models from different DL families, and 14 cluster configurations based on our in-house cluster and AWS. We provide additional details about the models and cluster specifications below and in Table 1 and Table 2.

Models. Table 1 list the details of the 11 models we have experimented with, including 5 CNN models for image classification (IC), a transformer-based model for neural machine translation (MT), two versions of the neural collaborative filtering model for recommendation systems, and four BERT
<table>
<thead>
<tr>
<th>Model</th>
<th>Task</th>
<th>Training data</th>
<th>bs/gpu</th>
<th>#params</th>
</tr>
</thead>
<tbody>
<tr>
<td>InceptionV3 [16]</td>
<td>IC</td>
<td>ImageNet</td>
<td>32</td>
<td>24M</td>
</tr>
<tr>
<td>VGG16 [15]</td>
<td>IC</td>
<td>ImageNet</td>
<td>32</td>
<td>138M</td>
</tr>
<tr>
<td>DenseNet121 [8]</td>
<td>IC</td>
<td>ImageNet</td>
<td>32</td>
<td>8M</td>
</tr>
<tr>
<td>Transformer [17]</td>
<td>MT</td>
<td>WMT’14 ende</td>
<td>5K</td>
<td>62M</td>
</tr>
<tr>
<td>NCF-dense [7]</td>
<td>CF</td>
<td>MovieLens-20Mx16x32</td>
<td>256</td>
<td>122M</td>
</tr>
<tr>
<td>NCF-sparse [7]</td>
<td>CF</td>
<td>MovieLens-20Mx16x32</td>
<td>256</td>
<td>122M</td>
</tr>
</tbody>
</table>

Table 1: Models we have experimented with. Their implementations are from the tensorflow/models repository. IC: image classification, MT: machine translation, LM: language modeling.

variants for language modeling. Their single-node implementations are from the official open source repository at https://github.com/tensorflow/models. Their training data, and number of parameters are listed in Table 1. We use default training settings suggested by MLPerf [11]. The per-GPU batch size is indicated in Table 1 as well.

It is worth noting that, for the neural collaborative filtering (NCF) model, we follow MLPerf [11] and use an enlarged version (x16x32) of the model and training set – which expands the original MovieLens-20M dataset [5] with 16x more users and 32x more movies. Details can be found at https://github.com/mlperf/training/tree/master/recommendation/pytorch#Getting-the-expanded-dataset. To test the system’s capability, we also experiment with two versions of NCF: NCF-dense, which uses dense tensors to represent the gradients of embedding variables in the model, and NCF-sparse, which uses sparse tensors (IndexSlices in TensorFlow) to represent the gradients instead.

Clusters. We focus on GPU clusters since it is the main setup for distributed DL training. During our research, we have tested on 14 different cluster configurations as listed in Table 2: Cluster A includes maximally 16 nodes, each equipped with a GeForce TITAN X GPU, an Intel 16-core CPU and 64GB RAM, interconnected via a 40-Gigabit Ethernet switch; Cluster B, based on AWS, consists of up to 8 node, each node is one of the g3.4xlarge (1x Tesla M60 GPU and 10GbE Ethernet), g3.16xlarge (4x M60, 25GbE), g4dn.2xlarge (1x NVIDIA T4 GPU and 25GbE Ethernet), g4dn.12xlarge (4x T4, 50GbE) instance types. Details about the configurations of AWS instances can be found at https://aws.amazon.com/ec2/instance-types/g4/ and https://aws.amazon.com/ec2/instance-types/g3/. Due to AWS constraints, they all have 10GbE single-flow bandwidth. On top of these two clusters, we list all the resource specifications we have experimented with, including cluster setup, number of GPUs on each cluster node, and bandwidth information, in Table 2.

To avoid confusion, it is worth noting that the setup A2 and B7 in Table 2 correspond to the Cluster A and Cluster B referred to in the main paper.

9.2 Public Datasets

We have collected a dataset containing nearly 10k data points of \{(G, D, S), f\}, where G is one of the DL models in Table 1, D is one of the cluster setups from Table 2, S is randomly sampled strategy from the proposed strategy space, and f is the groundtruth runtime collected via real distributed execution.

The dataset contains strategies sampled for all 11 models, ranging from fixed-formed strategies such as those used in specialized systems, and randomly explored strategies by AutoSync during the strategy auto-optimization. The data are organized into multiple folders where each folder corresponds to a domain of \((G, D)\). Hence the dataset used in the main paper is a subset containing several domains belonging to the entire collected dataset.
<table>
<thead>
<tr>
<th>$D$</th>
<th>Setup</th>
<th>GPU distribution</th>
<th>Bandwidth spec</th>
</tr>
</thead>
<tbody>
<tr>
<td>A1</td>
<td>16x Cluster A nodes</td>
<td>[1] x 16</td>
<td>40GbE</td>
</tr>
<tr>
<td>A2</td>
<td>11x Cluster A nodes</td>
<td>[1] x 11</td>
<td>40GbE</td>
</tr>
<tr>
<td>B1</td>
<td>2x g3.16</td>
<td>[4] x 2</td>
<td>25 GbE</td>
</tr>
<tr>
<td>B2</td>
<td>3x g3.16</td>
<td>[4] x 3</td>
<td>25 GbE</td>
</tr>
<tr>
<td>B3</td>
<td>4x g3.16</td>
<td>[4] x 4</td>
<td>25 GbE</td>
</tr>
<tr>
<td>B4</td>
<td>1x g4dn.12</td>
<td>[4] x 1</td>
<td>50 GbE</td>
</tr>
<tr>
<td>B5</td>
<td>2x g4dn.12</td>
<td>[4] x 2</td>
<td>50 GbE</td>
</tr>
<tr>
<td>B6</td>
<td>3x g4dn.12</td>
<td>[4] x 3</td>
<td>50 GbE</td>
</tr>
<tr>
<td>B7</td>
<td>4x g4dn.12</td>
<td>[4] x 4</td>
<td>50 GbE</td>
</tr>
<tr>
<td>B8</td>
<td>8x g4dn.12</td>
<td>[4] x 8</td>
<td>50 GbE</td>
</tr>
<tr>
<td>B9</td>
<td>1x g3.4, 1x g3.16</td>
<td>[1, 4]</td>
<td>10/25 GbE</td>
</tr>
<tr>
<td>B10</td>
<td>1x g3.16, 1x g4dn.12</td>
<td>[4] x 2</td>
<td>25/50 GbE</td>
</tr>
<tr>
<td>B11</td>
<td>2x g3.16, 2x g4dn.12</td>
<td>[4] x 4</td>
<td>25/50 GbE</td>
</tr>
<tr>
<td>B12</td>
<td>1x g4dn.2, 1x g4dn.12</td>
<td>[1, 4]</td>
<td>25/50 GbE</td>
</tr>
</tbody>
</table>

Table 2: Cluster specifications we have experimented with, listed with their reference name ($D$), setup information (Setup), GPU distribution and bandwidth specifications (Bandwidth spec).

For quick access, we have provided scripts that read $G$ as dataflow graphs in standard TensorFlow 2.0 format, and read the strategies and runtime into json formats. Instructions on how to access the dataset are provided at https://github.com/petuum/autodist.

10 Simulator Training Dataset and Hyperparameters

For the end-to-end results (Fig.2 in the main paper), we train simulators using runtime data collected on-the-fly during trials, strictly following the workflow illustrated in Fig.1 of the main paper.

For the ablation studies (Table.1-3), we split pre-collected data at auto-optimization of a specific domain (e.g., (NCF-dense, A)) into train/val/test at 70%/15%/15%, respectively, and report the ranking accuracy on the test split (averaged over 3 runs).

Training RNN simulators in all settings use Adam with 1e-3 lr, decayed by 0.3 at the 80th/160th epoch, for 200 epochs. We clipped the gradient norm to 0.25.

References


