Exploring Hidden Dimensions in Parallelizing Convolutional Neural Networks

Zhihao Jia [1] Sina Lin [2] Charles R. Qi [1] Alex Aiken [1]

Abstract

The past few years have witnessed growth in the computational requirements for training deep convolutional neural networks. Current approaches parallelize training onto multiple devices by applying a single parallelization strategy (e.g., data or model parallelism) to all layers in a network. Although easy to reason about, these approaches result in suboptimal runtime performance in large-scale distributed training, since different layers in a network may prefer different parallelization strategies. In this paper, we propose *layer-wise parallelism* that allows each layer in a network to use an individual parallelization strategy. We jointly optimize how each layer is parallelized by solving a graph search problem. Our evaluation shows that layer-wise parallelism outperforms state-of-the-art approaches by increasing training throughput, reducing communication costs, achieving better scalability to multiple GPUs, while maintaining original network accuracy.

1. Introduction

Convolutional neural networks (CNNs) have proven to be general and effective across many tasks including image classification (Krizhevsky et al., 2012; Szegedy et al., 2016), face recognition (Lawrence et al., 1997), text classification (Wang et al., 2012), and game playing (Silver et al., 2016). Their success has resulted in growth in the computational requirements to train today's CNNs, which takes days or even weeks on modern processors (Zeiler & Fergus, 2014; Simonyan & Zisserman, 2014; Szegedy et al., 2016).

Previous work has investigated parallelization techniques to accelerate training. The most common approach is *data parallelism* (Krizhevsky et al., 2012; Simonyan & Zisserman, 2014) that keeps a replica of an entire network on each device and assigns a subset of the training data to each device. Another common approach is *model parallelism* (Mirhoseini et al., 2017; Kim et al., 2017) that divides the network parameters into disjoint subsets and trains each subset on a dedicated device. Both approaches apply a single parallelization strategy (i.e., data or model parallelism) to all layers in a CNN. However, within a CNN, different layers may prefer different parallelization strategies for achieving optimal runtime performance. For example, a densely-connected layer with millions of parameters prefers model parallelism to reduce communication cost for synchronizing parameters, while a convolutional layer typically prefers data parallelism to eliminate data transfers from the previous layer. In addition, some layers may prefer more sophisticated parallelization strategies such as parallelizing in a mixture of multiple data dimensions (see Section 3). Because of the differing characteristics of different layers in a network, applying a single parallelization strategy to all layers usually results in suboptimal runtime performance.

In this paper, we propose *layer-wise parallelism*, which enables each layer in a network to use an individual parallelization strategy. Layer-wise parallelism performs the same computation for each layer as it is defined in the original network and therefore maintains the same network accuracy by design. Compared to existing parallelization approaches, our approach defines a more comprehensive search space of parallelization strategies, which includes data and model parallelism as two special cases. Our goal is to find the parallelization strategies for individual layers to jointly achieve the best possible runtime performance while maintaining the original network accuracy. To formalize the problem, we introduce *parallelization configurations* that define the search space for parallelizing a layer across multiple devices. We propose a cost model that quantitively evaluates the runtime performance of different parallelization strategies. The cost model considers both the computation power of each device and the communication bandwidth between devices. With the cost model, we convert the original problem of choosing parallelization configurations for individual layers to a graph search problem and develop an efficient algorithm to find a globally optimal strategy under the cost model.

We evaluate the runtime performance of layer-wise parallelism with AlexNet (Krizhevsky et al., 2012), VGG-16 (Simonyan & Zisserman, 2014), and Inception-v3 (Szegedy et al., 2016) on the ILSVRC 2012 image classification

[1]Stanford University [2]Microsoft. Correspondence to: Zhihao Jia <zhihao@cs.stanford.edu>.

Proceedings of the 35th International Conference on Machine Learning, Stockholm, Sweden, PMLR 80, 2018. Copyright 2018 by the author(s).

dataset. For distributed training on 16 P100 GPUs (on 4 nodes), layer-wise parallelism is 1.4-2.2× faster than state-of-the-art parallelization strategies. Note that the speedup is achieved without sacrificing network accuracy, since layer-wise parallelism trains the same network as data and model parallelism and uses more efficient parallelization strategies to achieve better runtime performance. In addition, layer-wise parallelism reduces communication costs by 1.3-23.0× compared to data and model parallelism. Finally, we show that layer-wise parallelism achieves better scalability than other parallelization strategies. Scaling the training of Inception-v3 from 1 GPU to 16 GPUs, layer-wise parallelism obtains 15.5× speedup, while other parallelization strategies achieve at most 11.2× speedup.

To summarize, our contributions are:

- We propose layer-wise parallelism, which allows different layers in a network to use individual parallelization configurations.
- We define the search space of possible parallelization configurations for a layer and present a cost model to quantitively evaluate the runtime performance of training a network. Based on the cost model, we develop an efficient algorithm to jointly find a globally optimal parallelization strategy.
- We provide an implementation that supports layer-wise parallelism and show that layer-wise parallelism can increase training throughput by 1.4-2.2× and reduce communication costs by 1.3-23.0× over state-of-the-art approaches while improving scalability.

2. Related Work

Data and model parallelism have been widely used by existing deep learning frameworks (e.g., TensorFlow (Abadi et al., 2016), Caffe2[2], and PyTorch[3]) to parallelize training. Data parallelism (Krizhevsky et al., 2012) keeps a copy of an entire network on each device, which is inefficient for layers with large numbers of network parameters and becomes a scalability bottleneck in large scale distributed training. Model parallelism (Dean et al., 2012) divides network parameters into disjoint subsets and trains each subset on a dedicated device. This reduces communication costs for synchronizing network parameters but exposes limited parallelism.

Krizhevsky (2014) introduces "one weird trick" (OWT) that uses data parallelism for convolutional and pooling layers and switches to model parallelism for fully-connected layers to accelerate training. This achieves better runtime performance than data and model parallelism but is still suboptimal. In this paper, we use OWT parallelism as a baseline in the experiments and show that layer-wise parallelism can further reduce communication costs and improve training performance compared to OWT parallelism.

System optimizations. A number of system-level optimizations have been proposed to accelerate large scale training. Goyal et al. (2017) uses a three-step allreduce operation to optimize communication across devices and aggressively overlaps gradient synchronization with back propagation. Zhang et al. (2017) introduces a hybrid communication scheme to reduce communication costs for gradient synchronization. All these systems are based on data parallelism and are limited in runtime performance by communication costs.

Network parameter reduction. Han et al. (2015) presents an iterative weight pruning method that repeatedly retrains the network while removing weak connections. Alvarez & Salzmann (2016) proposes a network that learns the redundant parameters in each layer and iteratively eliminates the redundant parameters. These approaches improve runtime performance by significantly reducing the number of parameters in a neural network, which results in a modified network and may decrease the network accuracy (as reported in these papers). By contrast, in this paper, we introduce a new approach that accelerates distributed training while maintaining the original network accuracy.

3. Hidden Dimensions in Parallelizing a Layer

Data parallelism parallelizes training by partitioning a training dataset in the sample dimension. However, other dimensions can also be used to parallelize a layer. For example, in standard CNNs for 2D images, data is commonly organized as 4-dimensional tensors (i.e., sample, height, width, and channel). The *sample* dimension includes an index for each image in a training dataset. The *height* and *width* dimensions specify a position in an image. For a particular position, the *channel* dimension indexes different neurons for that position.

In principle, any combination of these dimensions can be used to parallelize a layer, and we should consider both selecting the dimensions to parallelize training and the degree of parallelism in each dimension. Exploring these additional dimensions has the following advantages.

First, parallelizing a layer in other dimensions can reduce execution time. Figure 1 shows the time to process a 2D convolutional layer on 4 GPUs using parallelism in different dimensions. For this layer, data parallelism achieves suboptimal performance.

Second, exploring parallelism in other dimensions can re-

[2]https://caffe2.ai
[3]https://pytorch.org

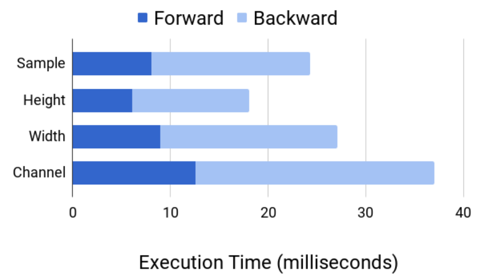
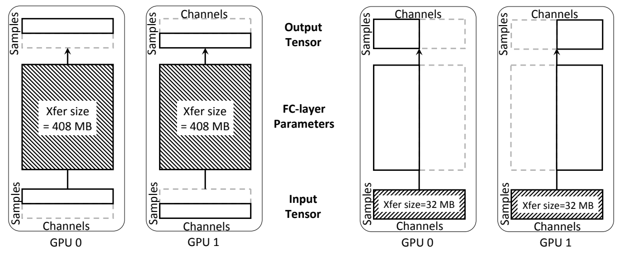


Figure 1. Execution time for parallelizing a convolutional layer (Conv8 in VGG-16 (Simonyan & Zisserman, 2014)) on 4 GPUs by using different dimensions. Parallelizing a layer in other dimensions preserves the same output as parallelizing it in the sample dimension. To achieve this, different GPUs may share some common input data for parallelizations in the height, width, and channel dimensions.



(a) Parallelism in the sample dimension.

(b) Parallelism in the channel dimension.

Figure 2. Different ways to parallelize the first fully-connected layer of VGG-16. Rectangles with solid lines indicate tensors managed by the local GPU, while rectangles with dotted lines are tensors managed by a remote GPU. The shadow rectangles indicate data transfers in each step.

duce communication costs. Figure 2 shows an example of parallelizing a fully-connected layer on two GPUs in different dimensions. In data parallelism (Figure 2a), each GPU synchronizes the gradients of the entire fully-connected layer (shown as the shadow rectangles) in every step. An alternative approach (Figure 2b) parallelizes in the channel dimension, which eliminates parameter synchronization, as different GPUs train disjoint subsets of the parameters, but introduces additional data transfers for input tensors (shown as the shadow rectangles). For this particular case, using parallelism in the channel dimension reduces communication costs by $12\times$.

Third, the *degree* of parallelism (number of parallel devices) is another dimension that affects runtime performance. Different layers have different execution time and communication costs and may prefer different degrees of parallelism. Figure 3 shows the runtime performance of processing two layers in Inception-v3 with different degrees of parallelism. The convolutional layer performs best on 16 GPUs, while the fully-connected layer performs best on 4 GPUs.

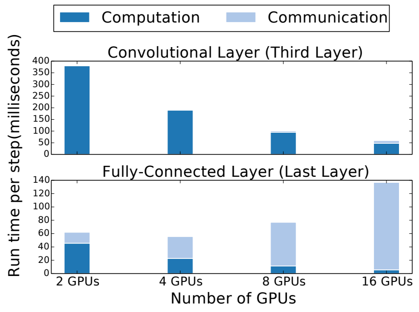


Figure 3. Computation and communication time to process the third layer and the last layer of Inception-v3 using data parallelism.

Table 1. Parallelizable dimensions for different layers. The *length* dimension specifies a position in a 1D image.

Layer	Parallelizable dimensions
Fully-connected	{sample, channel}
1D convolution/pooling	{sample, channel, length}
2D convolution/pooling	{sample, channel, height, width}
3D convolution/pooling	{sample, channel, height, width, depth}

4. Problem Definition

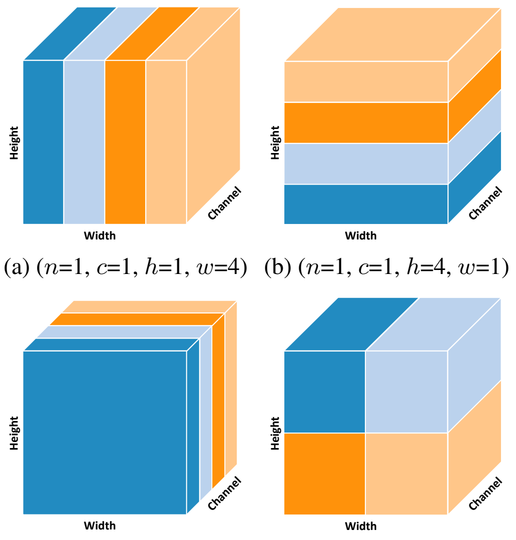
We define the parallelization problem with two graphs. The first is a *device graph* that models all available hardware devices and the connections between them. The second is a *computation graph* that defines the neural network to be mapped onto the device graph.

In a device graph $\mathcal{D}$, each node d_i is a device (e.g., a CPU or a GPU), and each edge (d_i, d_j) is an connection between d_i and d_j with communication bandwidth $b(d_i, d_j)$. In a computation graph $\mathcal{G}$, each node $l_i \in \mathcal{G}$ is a layer in the neural network, and each edge $(l_i, l_j) \in \mathcal{G}$ is a tensor that is an output of layer l_i and an input of layer l_j.

We now define the parallelization of a layer. To parallelize a layer across multiple devices, we assume that different devices can process the layer in parallel without any dependencies. This requires different devices to compute *disjoint* subsets of a layer's output tensor. Therefore, we describe the parallelization of a layer by defining how its output tensor is partitioned.

For a layer l_i, we define its *parallelizable dimensions* $\mathcal{P}_i$ as the set of all divisible dimensions in its output tensor. $\mathcal{P}_i$ includes all dimensions to parallelize the layer l_i. Table 1 shows the parallelizable dimensions for different layers.

A *parallelization configuration* c_i of a layer l_i defines how l_i is parallelized across different devices. For each parallelizable dimension in $\mathcal{P}_i$, c_i includes a positive integer that describes the degree of parallelism in that dimension. For a configuration c_i, the product of the integers over all dimensions is the total degree of parallelism for l_i. We assume equal partitioning in each parallelizable dimension, which provides well-balanced workload among multiple devices. Figure 4 demonstrates some possible configurations for parallelizing a 2D convolutional layer over four devices.



(a) (n=1, c=1, h=1, w=4) (b) (n=1, c=1, h=4, w=1)

(c) (n=1, c=4, h=1, w=1) (d) (n=1, c=1, h=2, w=2)

Figure 4. Example configurations to parallelize a 2D convolutional layer in a single dimension or combinations of multiple dimensions. The figure shows how each training sample is partitioned.

Parallelizing a layer in any configuration produces the same output. This guarantees that all configurations parallelize training on the original network and therefore maintains the original network accuracy.

A *parallelization strategy* $\mathcal{S}$ includes a configuration c_i for each layer $l_i \in \mathcal{G}$. Let $t(\mathcal{G}, \mathcal{D}, \mathcal{S})$ denote the per-iteration execution time to parallelize the computation graph $\mathcal{G}$ on the device graph $\mathcal{D}$ by using strategy $\mathcal{S}$. Our goal is to find a parallelization strategy $\mathcal{S}$ such that the per-iteration training time $t(\mathcal{G}, \mathcal{D}, \mathcal{S})$ is minimized.

5. Method

5.1. Cost Model

We introduce a cost model to quantitively evaluate the runtime performance of different parallelization strategies and use an *dynamic programming based* graph search algorithm to find an optimal parallelization strategy under our cost model. The cost model depends on the following assumptions:

1. For a layer $l_i \in \mathcal{G}$, the time to process l_i is predictable with low variance and is largely independent of the contents of the input data.
2. For each connection (d_i, d_j) between device d_i and d_j with bandwidth b, transferring a tensor of size s from d_i to d_j takes s/b time (i.e., the communication bandwidth can be fully utilized).
3. The runtime system has negligible overhead. A device begins processing a layer as soon as its input tensors are available and the device has finished previous tasks.

Most layers in CNNs are based on dense matrix operations, whose execution time satisfies the first assumption. In addition, the experiments show that our implementation satisfies the second and third assumptions well enough to obtain significant runtime performance improvements.

We define three cost functions on computation graphs:

1. For each layer l_i and its parallelization configuration c_i, $t_C(l_i, c_i)$ is the time to process the layer l_i under configuration c_i. This includes both the forward and back propagation time and is estimated by processing the layer under that configuration multiple times on the device and measuring the average execution time.
2. For each tensor $e = (l_i, l_j)$, $t_X(e, c_i, c_j)$ estimates the time to transfer the input tensors to the target devices, using the size of the data to be moved and the known communication bandwidth.
3. For each layer l_i and its parallelization configuration c_i, $t_S(l_i, c_i)$ is the time to synchronize the parameters in layer l_i after back propagation. To complete parameter synchronization, each device that holds a copy of the parameters for layer l_i transfers its local gradients to a parameter server that stores the up-to-date parameters for layer l_i. After receiving the gradients for layer l_i, the parameter server applies the gradients to the parameters and transfers the updated parameters back to the device. In this process, the communication time is much longer than the execution time to update parameters, therefore we use the communication time to approximate the parameter synchronization time.

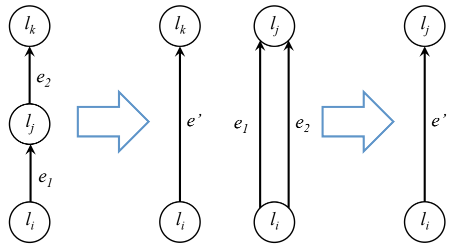
Using the three cost functions above, we define

$$\begin{aligned} t_O(\mathcal{G}, \mathcal{D}, \mathcal{S}) = &\sum_{l_i \in \mathcal{G}} \{t_C(l_i, c_i) + t_S(l_i, c_i)\} \\ &+ \sum_{e=(l_i, l_j) \in \mathcal{G}} t_X(e, c_i, c_j) \end{aligned} \tag{1}$$

$t_O(\mathcal{G}, \mathcal{D}, \mathcal{S})$ estimates the per-step execution time for parallelization strategy $\mathcal{S}$, which includes forward processing, back propagation, and parameter synchronization.

5.2. Graph Search

Equation 1 expresses the problem of finding an optimal parallelization strategy as a graph search problem: our goal is to find a strategy $\mathcal{S}$ so that the overall runtime cost $t_O(\mathcal{G}, \mathcal{D}, \mathcal{S})$ is minimized.



(a) Node elimination. (b) Edge elimination.

Figure 5. Node and edge elimination on a computation graph.

Since layer-wise parallelism allows each layer to use an individual configuration, the number of potential parallelization strategies is exponential in the number of layers in a computation graph, which makes it impractical to enumerate all strategies for large CNNs. However, the CNNs we have seen in practice exhibit strong locality: each layer is only connected to a few layers with similar depths in a computation graph. Based on this observation, we use the following two graph reductions to iteratively simplify a computation graph while preserving optimal parallelization strategies.

Node elimination. If a computation graph $\mathcal{G}$ includes a node l_j with a single in-edge $e_1 = (l_i, l_j)$ and a single out-edge $e_2 = (l_j, l_k)$, a node elimination removes node l_j and the two edges e_1 and e_2 from $\mathcal{G}$, inserts a new edge $e' = (l_i, l_k)$ back into $\mathcal{G}$, and returns the modified graph (Figure 5a). We define $t_X(e', \cdot, \cdot)$ in a way that preserves optimal parallelization strategies (see Theorem 1).

$$t_X(e', c_i, c_k) = \min_{c_j}\{t_C(l_j, c_j) + t_S(l_j, c_j) + t_X(e_1, c_i, c_j) + t_X(e_2, c_j, c_k)\} \tag{2}$$

Intuitively, we use dynamic programming to compute an optimal configuration c_j for node l_j for every possible combination of c_i and c_k and use the cost functions associated with l_j to define $t_X(e', c_i, c_k)$.

Theorem 1. *Assume $\mathcal{G}'$ = NodeElimination($\mathcal{G}$) and l_j is the eliminated node. If $\mathcal{S}_o{}'$ is an optimal parallelization srategy for $\mathcal{G}'$, then $\mathcal{S}_o = \mathcal{S}_o{}' + c_j$ is an optimal parallelization strategy for $\mathcal{G}$, where c_j minimizes Equation 2.*

Edge elimination. If a computation graph $\mathcal{G}$ includes two edges with the same source and destination nodes (i.e., $e_1 = (l_i, l_j)$ and $e_2 = (l_i, l_j)$), an edge elimination removes e_1 and e_2 from $\mathcal{G}$, inserts a new edge $e' = (l_i, l_j)$ into $\mathcal{G}$ (Figure 5b). We define $t_X(e', \cdot, \cdot)$ using $t_X(e_1, \cdot, \cdot)$ and $t_X(e_2, \cdot, \cdot)$.

$$t_X(e', c_i, c_j) = t_X(e_1, c_i, c_j) + t_X(e_2, c_i, c_j) \tag{3}$$

Theorem 2. *Assume $\mathcal{G}'$ = EdgeElimination($\mathcal{G}$), and $\mathcal{S}_o{}'$ is an optimal parallelization strategy of $\mathcal{G}'$, then $\mathcal{S}_o = \mathcal{S}_o{}'$ is an optimal parallelization strategy of $\mathcal{G}$.*

Algorithm 1 Finding Optimal Parallelization Strategy $\mathcal{S}$.

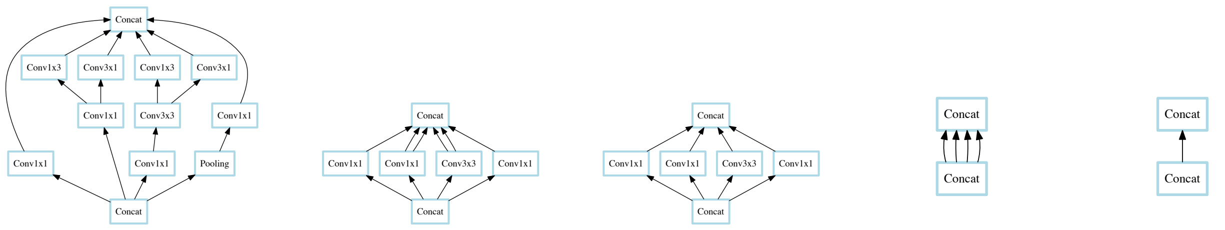
```
1: Input: A computation graph $\mathcal{G}$, a device graph $\mathcal{D}$, and
   precomputed cost functions (i.e., $t_C(\cdot)$, $t_S(\cdot)$ and $t_X(\cdot)$ )
2: Output: A parallelization strategy $\mathcal{S}$ minimizing
   $t_O(\mathcal{G}, \mathcal{D}, \mathcal{S})$
3:
4: $\mathcal{G}^{(0)} = \mathcal{G}$
5: $m = 0$
6: while true do
7:     $\mathcal{G}^{(m+1)}$ = NodeElimination($\mathcal{G}^{(m)}$)
8:     $\mathcal{G}^{(m+2)}$ = EdgeElimination($\mathcal{G}^{(m+1)}$)
9:     if $\mathcal{G}^{(m+2)} = \mathcal{G}^{(m)}$ then
10:        break
11:    end if
12:    $m = m + 2$
13: end while
14: Find the optimal strategy $\mathcal{S}^{(m)}$ for $\mathcal{G}^{(m)}$ by enumerating
    all possible candidate strategies
15: for $i = m$-1 to 0 do
16:     if $\mathcal{G}^{(i+1)}$ = NodeElimination($\mathcal{G}^{(i)}$) then
17:         ▷ Assume $l_j$ is the node eliminated from $\mathcal{G}^{(i)}$
18:         Find $c_j$ that minimizes Equation 1
19:         $\mathcal{S}^{(i)} = \mathcal{S}^{(i+1)} + c_j$
20:     else
21:         $\mathcal{S}^{(i)} = \mathcal{S}^{(i+1)}$
22:     end if
23: end for
24: return $\mathcal{S}^{(0)}$
```

We formally define node and edge eliminations and prove Theorem 1 and 2 in the appendix.[4] The two theorems show that given an optimal parallelization strategy for the modified graph, we can easily construct an optimal strategy for the original graph.

Algorithm 1 shows pseudocode using node and edge eliminations as subroutines to find an optimal parallelization strategy under our cost model. The algorithm first iteratively uses node and edge eliminations to simplify an input computation graph until neither elimination can be applied (lines 4-13). Figure 6 demonstrates how node and edge eliminations are performed on an Inception module (Szegedy et al., 2016).

After the elimination phase, the algorithm enumerates all potential strategies for the final graph $\mathcal{G}^{(m)}$ and chooses $\mathcal{S}^{(m)}$ that minimizes $t_O(\mathcal{G}^{(m)}, \mathcal{D}, \mathcal{S}^{(m)})$ (line 14). After deciding the configuration for each node in $\mathcal{G}^{(m)}$, we then decide the configurations for the eliminated nodes by iteratively undoing the eliminations in reverse order (lines 15-23). Theorem 1 and 2 guarantee that $\mathcal{S}^{(i)}$ is an optimal strategy for

[4] An extended version of this paper with proofs is available at https://arxiv.org/abs/1802.04924.



(a) Initial graph. (b) Node elimination. (c) Edge elimination. (d) Node elimination. (e) Edge elimination.

Figure 6. Iteratively performing node/edge eliminations on an Inception module.

Table 2. Time complexity of Algorithm 1. C is the maximum number of potential configurations for a layer. N and E are the number of nodes and edges in $\mathcal{G}$, respectively. K is the number of nodes in the final graph after node and edge eliminations.

Step	Time Complexity
Performing node and edge eliminations	$O(EC^3)$
Finding the optimal strategy for the final graph	$O(KC^K)$
Undoing node and edge eliminations	$O(EC)$
Overall	$O(EC^3 + KC^K)$

Table 3. Execution time for finding the optimal parallelization strategy for 4 GPUs. Note that the number of nodes in the final graph (i.e., K) is equal to 2 for all networks. For LeNet-5 and AlexNet, two algorithms find the same optimal strategies.

Network	# Layers	Baseline	Our Algorithm
LeNet-5	6	5.6 seconds	0.01 seconds
AlexNet	11	2.1 hours	0.02 seconds
VGG-16	21	$>$ 24 hours	0.1 seconds
Inception-v3	102	$>$ 24 hours	0.4 seconds
Time complexity		$O(EC^N)$	$O(EC^3)$

$\mathcal{G}^{(i)}$ $(0 \leq i \leq m)$. Finally, $\mathcal{S}^{(0)}$ is an optimal parallelization strategy for the original graph $\mathcal{G}$.

Time complexity. Table 2 shows the time complexity of Algorithm 1. Performing a node or edge elimination requires computing Equation 2 or 3 for the inserted edge, which takes $O(C^3)$ and $O(C^2)$ time, respectively. The total number of node and edge eliminations is smaller than E, since an elimination reduces the number of edges in the graph by one. Therefore, the time complexity for performing and undoing node and edge eliminations is $O(EC^3)$ and $O(EC)$, respectively. The algorithm enumerates all possible strategies for the final graph $\mathcal{G}^{(m)}$, which takes $O(KC^K)$ time. The algorithm works efficiently on a wide range of real-world CNNs including AlexNet (Krizhevsky et al., 2012), VGG (Simonyan & Zisserman, 2014), Inception-v3 (Szegedy et al., 2016), and ResNet (He et al., 2016), all of which are reduced to a final graph with only 2 nodes (i.e., $K = 2$).

We compare Algorithm 1 with a baseline algorithm that uses a *depth-first search* algorithm to find an optimal strategy for the original graph $\mathcal{G}$. Table 3 compares the time complexity and actual execution time of the two algorithms. Our algorithm achieves lower time complexity and reduces the execution time by orders of magnitude over the baseline.

6. Experiments

We found that it is non-trivial to parallelize a layer in the height, width, and channel dimensions in existing frameworks (e.g., TensorFlow, PyTorch, and Caffe2), and none provides an interface for controlling parallelization at the granularity of individual layers. Therefore, we implemented our framework in Legion (Bauer et al., 2012), a high-performance parallel runtime for distributed heterogeneous architectures, and use cuDNN (Chetlur et al., 2014) and cuBLAS (cub, 2016) as the underlying libraries to process neural network layers. The following Legion features significantly simplify our implementation. First, Legion supports high-dimensional partitioning that allows us to parallelizing any layer in any combination of the dimensions. Second, Legion permits control of parallelization at the granularity of each layer. Third, Legion allows fine-grain control over the placement of tasks and data. Fourth, the underlying implementation of Legion automatically and systematically overlaps communication with computation and optimizes the path and pipelining of data movement across the machine (Treichler et al., 2014; Jia et al., 2018; 2017).

Benchmarks. We evaluate our approach on three established CNNs. AlexNet (Krizhevsky et al., 2012) is the winner of the ILSVRC-2012 image classification competition. VGG-16 (Simonyan & Zisserman, 2014) improves network accuracy by pushing the depth of the network to 16 weighted layers. Inception-v3 (Szegedy et al., 2016) is a 102-layer deep CNN that uses carefully designed Inception modules to increase the number of layers while maintaining a reasonable computational budget.

Datasets. We evaluate the runtime performance of all three CNNs on the ImageNet-1K dataset (Deng et al., 2009) that consists of 1.2 million images from 1,000 categories.

Baselines. We compare the following parallelization strategies in the experiments.

1. **Data parallelism** is the most common parallelization strategy for large-scale training (Goyal et al., 2017; Abadi et al., 2016). In data parallelism, each device has a copy of the entire network and processes a subset of the training dataset.

2. **Model parallelism.** We use a model parallelism approach (Krizhevsky, 2014) as a baseline, which distributes the network parameters in each layer equally to all devices, providing good load balancing.

3. **OWT parallelism** is designed to reduce communication costs by using data parallelism for convolutional and pooling layers and switching to model parallelism for densely-connected layers.

4. **Layer-wise parallelism.** Given a computation graph and a set of available devices, we run the algorithm described in Section 5 to find a parallelization strategy minimizing Equation 1.

Experimental setup. All experiments were performed on a GPU cluster with 4 compute nodes, each of which is equipped with two Intel 10-core E5-2600 CPUs, 256G main memory, and four NVIDIA Tesla P100 GPUs. GPUs on the same node are connected by NVLink, and nodes are connected over 100Gb/s EDR Infiniband. We use synchronous training and a per-GPU batch size of 32 for all experiments.

To rule out implementation differences, we ran data parallelism experiments in TensorFlow r1.7, PyTorch v0.3, and our implementation and compared the runtime performance. Our Legion-based framework achieves the same or better runtime performance on all three CNNs compared to TensorFlow and PyTorch, and therefore we report the data parallelism performance achieved by our framework.

6.1. Runtime Performance

We compare the training throughput and communication cost among different parallelization strategies. Figure 7 shows the training throughputs with different CNNs and different sets of available devices. Both model and data parallelism scale well on a single node but show limited scalability in distributed training, where the inter-node communication limits runtime performance. OWT parallelism achieves improved runtime performance by switching to model parallelism for densely-connected layers to reduce communication costs. In all experiments, layer-wise parallelism consistently outperforms the other strategies and increases the training throughput by up to 2.2×, 1.5×, and 1.4× for AlexNet, VGG-16, and Inception-v3, respectively.

In addition, layer-wise parallelism achieves better scalability than the other strategies. Scaling the training of the three CNNs from 1 GPU to 16 GPUs (on 4 nodes), layer-wise parallelism achieves 12.2×, 14.8×, and 15.5× speedup for AlexNet, VGG-16, and Inception-v3, respectively, while the best other strategy achieves 6.1×, 10.2×, and 11.2× speedup. Moreover, Figure 7 shows that the layer-wise parallelism can help bridge the runtime performance gap between the ideal training throughputs in linear scale (the red lines) and the actual training throughputs achieved by current parallelization strategies. This shows that layer-wise parallelism is more efficient for large-scale training.

Table 4. Relative difference between estimated execution time $t_O(\mathcal{G}, \mathcal{C})$ and actual execution time $t(\mathcal{G}, \mathcal{C})$.

Available Devices	$(t_O(\mathcal{G},\mathcal{C}) - t(\mathcal{G},\mathcal{C}))/t(\mathcal{G},\mathcal{C})$		
	AlexNet	**VGG-16**	**Inception-v3**
1 GPU (1 node)	1%	0%	1%
2 GPUs (1 node)	4%	3%	5%
4 GPUs (1 node)	-5%	2%	5%
8 GPUs (2 nodes)	2%	6%	9%
16 GPUs (4 nodes)	-1%	7%	6%

Table 5. An optimal parallelization strategy under the cost model for parallelizing VGG-16 on 4 GPUs on a single compute node.

Layers	**Parallelization Configuration**
2 x Conv + Pooling 2 x Conv + Pooling 3 x Conv + Pooling 3 x Conv + Pooling	{n=4, h=1, w=1, c=1}
3 x Conv + Pooling	{n=1, h=2, w=2, c=1}
Fully-connected Fully-connected	{n=1, c=4}
Fully-connected	{n=1, c=2}
Softmax	{n=1, c=1}

Communication cost is another important performance metric in large-scale training. Figure 8 compares the communication costs of different strategies. OWT parallelism eliminates gradient synchronization for densely-connected layers and reduces overall communication costs by 1.1-23.0× compared to data and model parallelism. In addition, Layer-wise parallelism outperforms OWT parallelism by further reducing communication overhead by 1.2-2.5×.

6.2. Cost Model

We compare the estimated execution time $t_O(\mathcal{G}, \mathcal{D}, \mathcal{C})$ projected by our cost model (see Equation 1) with the measured per-step execution time $t(\mathcal{G}, \mathcal{D}, \mathcal{C})$ in the experiments. The results are shown in Table 4. In all experiments, the relative difference between the estimated and the real execution time is within 10%, showing that the cost model can reliably predict a CNN's per-step execution time given the set of available devices and the connections between them.

6.3. Analysis of Optimal Parallelization Strategies

We analyze the optimal parallelization strategies under our cost model and find several similarities among them.

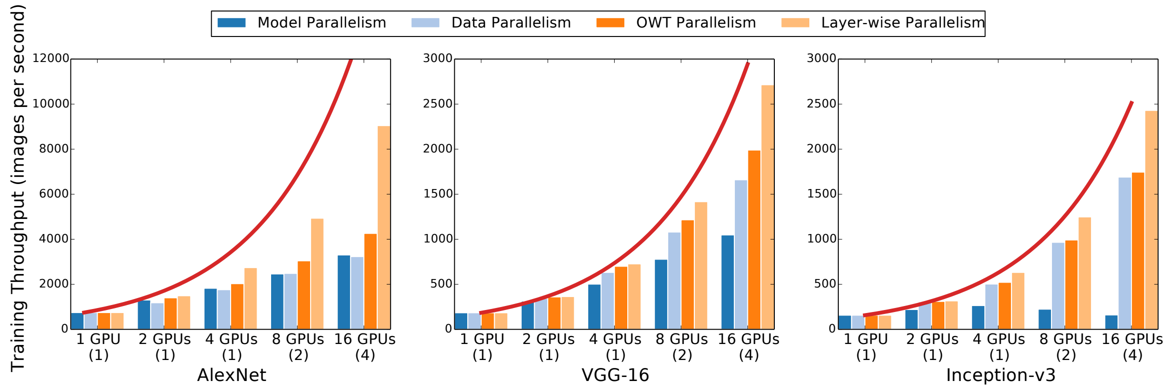


Figure 7. Training throughput (i.e., number of images processed per second) with different parallelization strategies (higher is better). Numbers in parenthesis are the number of compute nodes used in the experiments. The red lines show the training throughput in linear scale (ideal case).

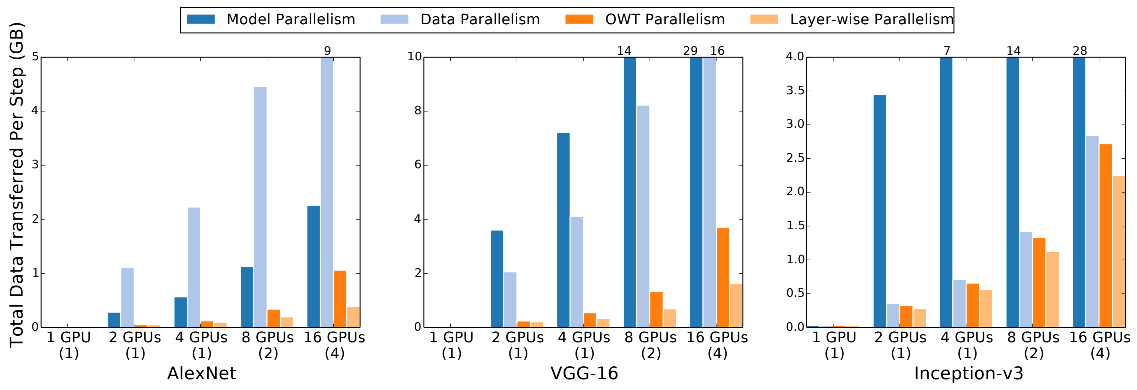


Figure 8. Communication cost (i.e., data transferred in each step) with different parallelization strategies (lower is better).

First, for the beginning layers of a CNN with large height/width dimensions and a small channel dimension, an optimal strategy usually uses data parallelism on all available devices, since the communication costs for synchronizing gradients are much smaller than the communication costs for moving tensors between layers.

Second, deeper layers in a CNN tend to have smaller height/width dimensions and a larger channel dimension. As a result, the costs for moving tensors between different layers decrease, while the costs for synchronizing parameters increase. An optimal strategy adaptively reduces the number of devices for these layers to reduce communication costs to synchronize parameters and opportunistically uses parallelism in the height/width dimensions to achieve better runtime performance.

Finally, for densely-connected layers, an optimal strategy eventually switches to model parallelism on a small number of devices, because synchronizing gradients and transferring tensors are both much more expensive than the execution time for densely-connected layers. This reduces the communication costs for synchronizing parameters and moving tensors at the cost of only using a subset of available devices.

Table 5 shows an optimal strategy under the cost model for parallelizing VGG-16 on 4 GPUs. This strategy first uses parallelism in the sample dimension for the beginning convolutional and pooling layers and then uses parallelism in both the height and width dimensions to accelerate the last three convolutional layers. For the fully-connected layers, it uses parallelism in the channel dimension to reduce communication costs and adaptively decreases the degrees of parallelism.

7. Conclusion

We have introduced layer-wise parallelism, which allows each layer in a CNN to use an individual parallelization configuration. We propose a cost model that quantitively evaluates the runtime performance of different strategies and use a dynamic programming based graph search algorithm to find a globally optimal strategy under the cost model. Our experiments show that layer-wise parallelism significantly outperforms state-of-the-art strategies for CNNs by increasing training throughput, reducing communication costs, and achieving better scalability on larger numbers of devices.

Acknowledgements

This research was supported by NSF grant CCF-1160904 and the Exascale Computing Project (17-SC-20-SC), a collaborative effort of the U.S. Department of Energy Office of Science and the National Nuclear Security Administration.

References

Dense Linear Algebra on GPUs. https://developer.nvidia.com/cublas, 2016.

Abadi, M., Barham, P., Chen, J., Chen, Z., Davis, A., Dean, J., Devin, M., Ghemawat, S., Irving, G., Isard, M., Kudlur, M., Levenberg, J., Monga, R., Moore, S., Murray, D. G., Steiner, B., Tucker, P., Vasudevan, V., Warden, P., Wicke, M., Yu, Y., and Zheng, X. Tensorflow: A system for large-scale machine learning. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI, 2016.

Alvarez, J. M. and Salzmann, M. Learning the number of neurons in deep networks. In *Proceedings of the 29th International Conference on Neural Information Processing Systems*, NIPS, 2016.

Bauer, M., Treichler, S., Slaughter, E., and Aiken, A. Legion: Expressing locality and independence with logical regions. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC, 2012.

Chetlur, S., Woolley, C., Vandermersch, P., Cohen, J., Tran, J., Catanzaro, B., and Shelhamer, E. cudnn: Efficient primitives for deep learning. *CoRR*, abs/1410.0759, 2014. URL http://arxiv.org/abs/1410.0759.

Dean, J., Corrado, G. S., Monga, R., Chen, K., Devin, M., Le, Q. V., Mao, M. Z., Ranzato, M., Senior, A., Tucker, P., Yang, K., and Ng, A. Y. Large scale distributed deep networks. In *Proceedings of the International Conference on Neural Information Processing Systems*, NIPS, 2012.

Deng, J., Dong, W., Socher, R., Li, L.-J., Li, K., and Fei-Fei, L. ImageNet: A large-scale hierarchical image database. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, CVPR, 2009.

Goyal, P., Dollár, P., Girshick, R. B., Noordhuis, P., Wesolowski, L., Kyrola, A., Tulloch, A., Jia, Y., and He, K. Accurate, large minibatch SGD: training imagenet in 1 hour. *CoRR*, abs/1706.02677, 2017. URL http://arxiv.org/abs/1706.02677.

Han, S., Pool, J., Tran, J., and Dally, W. J. Learning both weights and connections for efficient neural networks. In *Proceedings of the 28th International Conference on Neural Information Processing Systems*, NIPS, 2015.

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*, CVPR, 2016.

Jia, Z., Kwon, Y., Shipman, G., McCormick, P., Erez, M., and Aiken, A. A distributed multi-gpu system for fast graph processing. *PVLDB*, 11(3), 2017.

Jia, Z., Treichler, S., Shipman, G., McCormick, P., and Aiken, A. Isometry: A path-based distributed data transfer system. In *Proceedings of the International Conference on Supercomputing*, ICS, 2018.

Kim, J., Park, Y., Kim, G., and Hwang, S. J. SplitNet: Learning to semantically split deep networks for parameter reduction and model parallelization. In *Proceedings of the 34th International Conference on Machine Learning*, ICML, 2017.

Krizhevsky, A. One weird trick for parallelizing convolutional neural networks. *CoRR*, abs/1404.5997, 2014. URL http://arxiv.org/abs/1404.5997.

Krizhevsky, A., Sutskever, I., and Hinton, G. E. ImageNet classification with deep convolutional neural networks. In *Proceedings of the 25th International Conference on Neural Information Processing Systems*, NIPS, 2012.

Lawrence, S., Giles, C. L., Tsoi, A. C., and Back, A. D. Face recognition: A convolutional neural-network approach. *IEEE transactions on neural networks*, 1997.

Mirhoseini, A., Pham, H., Le, Q. V., Steiner, B., Larsen, R., Zhou, Y., Kumar, N., Norouzi, M., Bengio, S., and Dean, J. Device placement optimization with reinforcement learning. In *Proceedings of the 34th International Conference on Machine Learning*, ICML, 2017.

Silver, D., Huang, A., Maddison, C. J., Guez, A., Sifre, L., Van Den Driessche, G., Schrittwieser, J., Antonoglou, I., Panneershelvam, V., Lanctot, M., et al. Mastering the game of go with deep neural networks and tree search. *Nature*, 529:484–489, 2016.

Simonyan, K. and Zisserman, A. Very deep convolutional networks for large-scale image recognition. *CoRR*, abs/1409.1556, 2014. URL http://arxiv.org/abs/1409.1556.

Szegedy, C., Vanhoucke, V., Ioffe, S., Shlens, J., and Wojna, Z. Rethinking the inception architecture for computer vision. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, CVPR, 2016.

Treichler, S., Bauer, M., and Aiken, A. Realm: An event-based low-level runtime for distributed memory architectures. In *Proceedings of the International Conference on*

Parallel Architecture and Compilation Techniques, PACT, 2014.

Wang, T., Wu, D. J., Coates, A., and Ng, A. Y. End-to-end text recognition with convolutional neural networks. In *Proceddings of the 21st International Conference on Pattern Recognition*, ICPR, 2012.

Zeiler, M. D. and Fergus, R. Visualizing and understanding convolutional networks. In *Proceedings of the European Conference on Computer Vision*, ECCV, 2014.

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*, ATC, 2017.

A. Node and Edge Eliminations

We define node and edge eliminations in Algorithm 2.

Algorithm 2 Node and edge eliminations.

```
1:  function NodeElimination($\mathcal{G}$)
2:      if there exists a node $l_j$ with a single in-edge $e_1 = (l_i, l_j)$ and a single out-edge $e_2 = (l_j, l_k)$ then
3:          $e' = (l_i, l_k)$
4:          $\mathcal{G}' = \mathcal{G} - l_j - e_1 - e_2 + e'$
5:          return $\mathcal{G}'$
6:      else
7:          return $\mathcal{G}$
8:      end if
9:  end function
10:
11: function EdgeElimination($\mathcal{G}$)
12:     if there exist two edges $e_1 = (l_i, l_j)$ and $e_2 = (l_i, l_j)$ then
13:         $e' = (l_i, l_j)$
14:         $\mathcal{G}' = \mathcal{G} - e_1 - e_2 + e'$
15:         return $\mathcal{G}'$
16:     else
17:         return $\mathcal{G}$
18:     end if
19: end function
20:
```

Theorem 3. *Assume $\mathcal{G}'$ = NodeElimination($\mathcal{G}$) and l_j is the eliminated layer. If ${\mathcal{S}_o}'$ is an optimal strategy for $\mathcal{G}'$, then $\mathcal{S}_o = {\mathcal{S}_o}' + \widehat{c_j}$ is an optimal strategy for $\mathcal{G}$, where*

$$\widehat{c_j} = \arg\min_{c_j}\{t_C(n_j, c_j) + t_S(n_j, c_j) + t_X(e_1, c_i, c_j) + t_X(e_2, c_j, c_k)\} \tag{4}$$

Proof. It suffices to prove that $t_O(\mathcal{G}, \mathcal{S}_1) \geq t_O(\mathcal{G}, \mathcal{S}_o)$ for any other strategy $\mathcal{S}_1$. We assume layer l_i has parallelization configuration $c_{i1} \in \mathcal{S}_1$. We claim that

$$\begin{aligned} t_O(\mathcal{G}, \mathcal{S}_1) &\geq t_O(\mathcal{G}', \mathcal{S}_1) && (5)\\ &\geq t_O(\mathcal{G}', {\mathcal{S}_o}') && (6)\\ &= t_O(\mathcal{G}, \mathcal{S}_o) && (7)\end{aligned}$$

To prove (5), note that the difference between $t_O(\mathcal{G}, \mathcal{S}_1)$ and $t_O(\mathcal{G}', \mathcal{S}_1)$ is

$$\begin{aligned} &t_O(\mathcal{G}, \mathcal{S}_1) - t_O(\mathcal{G}', \mathcal{S}_1)\\ =&t_C(l_j, c_{j1}) + t_S(l_j, c_{j1}) + t_X(e_1, c_{i1}, c_{j1})\\ &+ t_X(e_2, c_{j1}, c_{k1}) - t_X(e', c_{i1}, c_{k1})\end{aligned} \tag{8}$$

because all other layers except l_j use the same configurations in $t_O(\mathcal{G}, \mathcal{S}_1)$ and $t_O(\mathcal{G}', \mathcal{S}_1)$, and therefore all cost functions unrelated to l_j drop out in the subtraction. The remaining parts are l_j, e_1, and e_2, which no longer exist in $\mathcal{G}'$ after node elimination, and e' that is added to $\mathcal{G}'$. Recall that $t_X(e', \cdot, \cdot)$ is defined as follows.

$$\begin{aligned} t_X(e', c_i, c_k) = \min_{c_j}\{&t_C(l_j, c_j) + t_S(l_j, c_j)\\ &+ t_X(e_1, c_i, c_j) + t_X(e_2, c_j, c_k)\}\end{aligned} \tag{9}$$

Combining (8) and (9), we have $t_O(\mathcal{G}, \mathcal{S}_1) \geq t_O(\mathcal{G}', \mathcal{S}_1)$.

To prove (6), simply observe that the inequality must hold because ${\mathcal{S}_o}'$ is assumed to be an optimal strategy for $\mathcal{G}'$.

To prove (7), the difference between $t_O(\mathcal{G}', {\mathcal{S}_o}')$ and $t_O(\mathcal{G}, \mathcal{S}_o)$ is

$$\begin{aligned} &t_O(\mathcal{G}, \mathcal{S}_o) - t_O(\mathcal{G}', {\mathcal{S}_o}')\\ =&t_C(l_j, \widehat{c_j}) + t_S(l_j, \widehat{c_j}) + t_X(e_1, c_i, \widehat{c_j})\\ &+ t_X(e_2, \widehat{c_j}, c_k) - t_X(e', c_i, c_k)\end{aligned} \tag{10}$$

This is because $\mathcal{S}_o = {\mathcal{S}_o}' + \widehat{c_j}$, and therefore all cost functions unrelated to l_j drop out. We can prove (7) by plugging (4) into (10). □

Theorem 4. *Assume $\mathcal{G}'$ = EdgeElimination($\mathcal{G}$), and ${\mathcal{S}_o}'$ is an optimal strategy for $\mathcal{G}'$, then $\mathcal{S}_o = {\mathcal{S}_o}'$ is an optimal strategy for $\mathcal{G}$.*

Proof. The proof is the same sequence of steps for Theorem 3, but the justification of each step is different.

To prove (5) for Theorem 4, the difference between $t_O(\mathcal{G}, \mathcal{S}_1)$ and $t_O(\mathcal{G}', \mathcal{S}_1)$ is

$$\begin{aligned} &t_O(\mathcal{G}, \mathcal{S}_1) - t_O(\mathcal{G}', \mathcal{S}_1)\\ =&t_X(e_1, c_{i1}, c_{j1}) + t_X(e_2, c_{i1}, c_{j1}) - t_X(e', c_{i1}, c_{j1})\end{aligned} \tag{11}$$

Recall that $t_X(e', \cdot, \cdot)$ is defined as follows:

$$t_X(e', c_i, c_j) = t_X(e_1, c_i, c_j) + t_X(e_2, c_i, c_j) \tag{12}$$

Combining (11) and (12), we have $t_{\mathcal{O}}(\mathcal{G}, \mathcal{S}_1) = t_{\mathcal{O}}(\mathcal{G}', \mathcal{S}_1)$.

For (6), the inequality holds because ${\mathcal{S}_o}'$ is an optimal strategy for $\mathcal{G}'$.

For (7), the difference between $t_{\mathcal{O}}(\mathcal{G}', {\mathcal{S}_o}')$ and $t_{\mathcal{O}}(\mathcal{G}, \mathcal{S}_o)$ is

$$\begin{aligned} & t_{\mathcal{O}}(\mathcal{G}, \mathcal{S}_o) - t_{\mathcal{O}}(\mathcal{G}', {\mathcal{S}_o}') \\ = & t_{\mathcal{X}}(e_1, c_i, c_j) + t_{\mathcal{X}}(e_2, c_i, c_j) - t_{\mathcal{X}}(e', c_i, c_j) \\ = & 0 \end{aligned} \tag{13}$$

□