CASSINI: Network-Aware Job Scheduling in Machine Learning Clusters

Sudarsanan Rajasekaran and Manya Ghobadi, *Massachusetts Institute of Technology;*
Aditya Akella, *UT Austin*

https://www.usenix.org/conference/nsdi24/presentation/rajasekaran

This paper is included in the
Proceedings of the 21st USENIX Symposium on
Networked Systems Design and Implementation.

April 16–18, 2024 • Santa Clara, CA, USA

978-1-939133-39-7

Open access to the Proceedings of the
21st USENIX Symposium on Networked
Systems Design and Implementation
is sponsored by

جامعة الملك عبدالله
للعلوم والتقنية
King Abdullah University of
Science and Technology

CASSINI: Network-Aware Job Scheduling in Machine Learning Clusters

Sudarsanan Rajasekaran[†] *Manya Ghobadi*[†] *Aditya Akella*[‡]

[†]*Massachusetts Institute of Technology* [‡]*UT Austin*

Abstract

We present CASSINI, a network-aware job scheduler for machine learning (ML) clusters. CASSINI introduces a novel geometric abstraction to consider the communication pattern of different jobs while placing them on network links. To do so, CASSINI uses an Affinity graph that finds a series of time-shift values to adjust the communication phases of a subset of jobs such that the communication patterns of jobs sharing the same network link are interleaved with each other. Experiments with 13 common ML models on a 24-server testbed demonstrate that compared to the state-of-the-art ML schedulers, CASSINI improves the average and tail completion time of jobs by up to 1.6× and 2.5×, respectively. Moreover, we show that CASSINI reduces the number of ECN marked packets in the cluster by up to 33×.

1 Introduction

The ever-growing increase in dataset and model sizes of deep learning has created a massive demand for efficient GPU clusters. Several studies have demonstrated that as the number of GPUs increases, the communication overhead of distributed Machine Learning (ML) training workloads quickly takes up a significant portion of training iteration time [12, 15, 28, 33, 45, 47, 55]. Yet state-of-the-art ML schedulers tend to ignore the communication pattern of ML training jobs when placing workers on GPUs.

In this paper, we develop a simple but effective approach, called CASSINI, that integrates with existing ML schedulers to allow them to efficiently place multiple ML jobs on network links while minimizing the chances of network congestion. Our approach requires no special support, such as reservations and priorities, from switches/NICs and does not require any changes to the congestion control protocol.

We demonstrate that for a specific combination of jobs, introducing a small time-shift to delay the start of one of the iterations enables CASSINI to interleave the computation and communication patterns of different jobs, thereby improving the training time. We refer to such combinations of jobs as *compatible* and develop CASSINI as a pluggable module to augment prior ML schedulers to consider a novel *compatibility metric* when determining where to place jobs and control how jobs compete on network links.

Augmenting ML schedulers to take links and servers into account is inherently challenging because jobs are likely to traverse multiple links and may compete with different jobs on different links. To address this challenge, we propose a *geometric abstraction* that leverages the periodic communication pattern of Deep Neural Network (DNN) training workloads. The key idea of our abstraction is to "roll" time around a circle whose perimeter is proportional to the training iteration time of ML jobs. To determine the compatibility score of two (or more) jobs on a link, CASSINI places each job on its corresponding circle and overlays the circles on top of each other. It then uses an optimization formulation to rotate the circles into a position that maximizes interleaving. The rotation angle of each job corresponds to a time-shift value to delay the start of the next immediate training iteration to achieve compatibility.

Looking beyond a single link and extending to jobs running across a topology, we generalize the geometric abstraction to cluster-level by introducing a bipartite Affinity graph whose vertices are a subset of jobs and links. An edge in the Affinity graph indicates a job is traversing a link. We then use a new graph traversal algorithm to find unique time-shifts for all jobs while maintaining their compatibility on all links. Using our geometric abstraction and Affinity graph, we augment Themis [40] and Pollux [50] with ≈1000 lines of code.

To evaluate CASSINI, we build a testbed with 24 servers, each with one NVIDIA A100 GPU [8] and one 50 Gbps RDMA NIC. Our experiments with 13 representative DNN models (VGG11 [26], VGG16 [18], VGG19 [32], ResNet50 [27], WideResNet101 [72], BERT [20], RoBERTa [39], XLM [17], CamemBERT [43], GPT-1 [51], GPT-2 [52], GPT-3 [11], and DLRM [6]) show that CASSINI improves the tail completion time of jobs by up to 2.2× and 2.5×, compared to Themis [40] and Pollux [50], respectively. Moreover, we show that CASSINI reduces the number of ECN marked packets in the cluster by up to 33×.

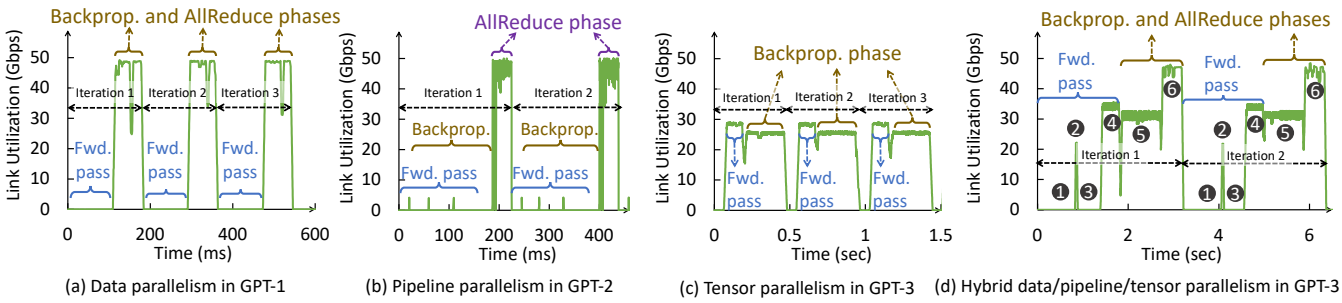


Figure 1: The traffic pattern of different parallelization strategies when training GPT-1, GPT-2, and GPT-3 models.

2 Background and Motivation

2.1 Distributed DNN Training Traffic Pattern

CASSINI is designed for large GPU clusters with hundreds of training jobs distributed with data, pipeline, and/or model parallel training paradigms. To this end, we study the impact of different parallelization strategies on network demand using a series of measurements. Each server in our testbed has one A100 GPU and one ConnectX-5 Mellanox RDMA NIC with 50 Gbps capacity. In all our experiments, we choose batch sizes such that the GPU utilization is higher than 80%, and intra-job pipelining is enabled.

Data parallelism. In data parallel training, the DNN model is copied into the memory of each GPU while the dataset is distributed across them. Figure 1(a) shows the communication pattern of a GPT-1 [51] model (12 layers, 9 GB memory) distributed across four GPU servers using data parallelism. The figure shows the traffic pattern of three back-to-back training iterations. Each iteration contains a forward pass with near-zero network demand, followed by a period of high utilization corresponding to the backpropagation and AllReduce phases.

Model/Pipeline parallelism. In model parallel training, the DNN model is partitioned across workers [29, 35], and parts of the DNN model are computed on different workers. The two common techniques for model parallelism are tensor parallelism and pipeline (or layer) parallelism [10]. In pipeline parallelism, the model is partitioned vertically at the layer boundaries [28, 47]. Figure 1(b) shows the communication pattern of a GPT-2 [52] model (24 layers, 27 GB memory) distributed across two servers using pipeline parallelism. We partition the model vertically in half (i.e., server_1 contains layer_1 to layer_{12}, and server_2 contains layer_{13} to layer_{24}) and use PipeDream's approach [47] to divide the batch size into three minibatches. The three small communication peaks during the forward pass correspond to the activation parameters of these three minibatches. The heavy communication demand following the peaks corresponds to the AllReduce operation between the embedding layers in the model.

Model/Tensor parallelism. Another variant of model parallel training is tensor parallelism [58, 59]. Tensor parallelism techniques partition the model horizontally such that different tensors are distributed across workers [31, 64]. Figure 1(c) shows the communication pattern of a GPT-3 [11] model (96 layers, 35 GB memory) distributed across two servers using tensor parallelism. We partition the model horizontally in half, where each server contains half of all the layers. The figure shows that both forward and backpropagation phases introduce roughly 25 Gbps traffic followed by a short period of near-zero network demand during data loading.

Hybrid data/pipeline/tensor parallelism. Today's DNN training systems tend to use a hybrid of data/pipeline/tensor parallelism to train large DNN models [21, 33, 46, 66]. Figure 1(d) shows the communication pattern of a GPT-3 [11] model (96 layers, 155 GB memory) distributed across eight servers using hybrid data/pipeline/tensor parallelism. We use pipeline parallelism to partition the model's layers vertically into two parts. Then, we divide the layers in each partition horizontally to obtain a total of four partitions. Next, we assign each of these four partitions to a server. Finally, we replicate the same process across another group of four servers and use data parallelism to distribute the data between these two groups of four servers. The figure shows the communication demand of the forward, backpropagation, and AllReduce phases where each phase has a different network demand.

Key takeaways. We repeat the above experiments using common DNN models, such as BERT [20], DLRM [6], WideResNet101 [72], RoBERTa [39], and VGG [62] and observe similar traffic patterns. Our key takeaways are: (*i*) the network demand repeats itself across all iterations, as long as the training hyper-parameters remain the same; (*ii*) the network demand of an iteration may consist of multiple Up and Down phases. The exact magnitude of the network demand during these Up and Down phases depends on the parallelization strategy and hyper-parameters. For instance, Figure 1(d) shows each training iteration has six Up-Down phases, labeled as ❶ to ❻. Section 3 describes CASSINI's approach to capture the duration and bandwidth of Up-Down phases.

2.2 Interleaving the Up and Down Phases

CASSINI's goal is to augment ML schedulers to consider the traffic demand of training jobs when making scheduling decisions. In particular, given the key takeaways in the previous

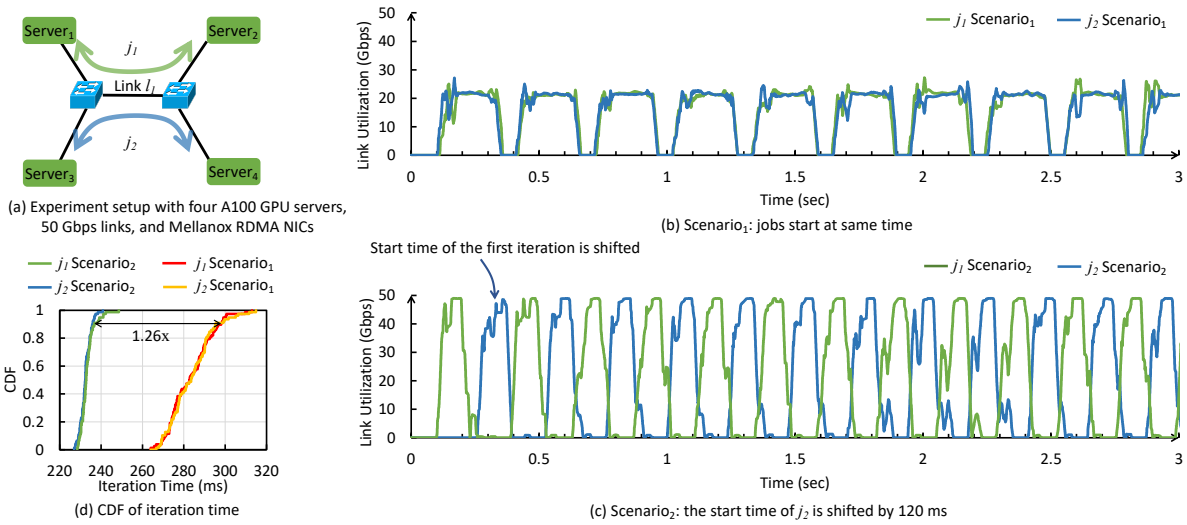


Figure 2: Impact of interleaving the Up-Down phases of two VGG19 jobs sharing link l_1.

section, we aim to interleave the bandwidth demand of Up and Down phases of different jobs to leverage the periodic network demand of distributed DNN training jobs.

To demonstrate the power of Up-Down network demand interleaving, we consider two data parallel training jobs, j_1 and j_2, as shown in Figure 2(a). Each job has one Up and one Down phase at every training iteration. We run each job for 1,000 iterations under two scenarios. In the first scenario, two VGG19 [62] jobs start simultaneously and share l_1 fairly. The communication uses the RDMA-based DCQCN congestion control algorithm [77]. Figure 2(b) shows that both jobs achieve roughly 22 Gbps bandwidth (i.e., half of l_1's capacity). In the second scenario, shown in Figure 2(c), we interleave the Down phase of j_1 with the Up phase of j_2 and vice versa, by shifting the start time of j_2 by 120 ms (Section 3 describes how we obtained this value). In this scenario, the jobs do not compete for bandwidth during their respective Up phases, giving both jobs the entire available bandwidth. Figure 2(d) plots the CDF of training iteration times for both scenarios demonstrating that scenario$_2$ accelerates the 90^{th} percentile tail iteration time of both jobs by 1.26×.

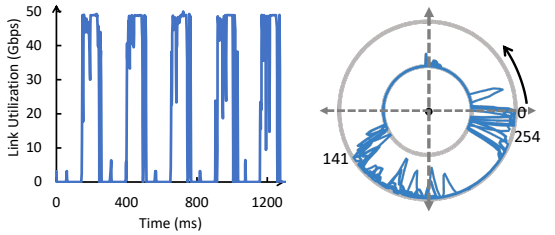
Perfectly interleaving the Up and Down phases of different jobs is not always possible. For instance, when BERT [20] and VGG19 [62] models share a link, no suitable time-shift can achieve perfect interleaving. But when WideResNet101 [72] and VGG16 [62] share a link, shifting VGG16 by 150 ms enables perfect interleaving. Instead of relying on perfectly matching Up and Down phases of jobs, we define a metric called *compatibility score* that captures the potential degree of interleaving across jobs sharing the network. Section 3 describes a novel technique to determine the compatibility score and the amount of required time-shift to achieve it.

3 Geometric Abstraction

Consider a time-series representation of the network demand for a job running in a dedicated cluster with no congestion. As shown in Section 2, different training jobs have different Up and Down patterns but the duration and bandwidth demand of the same job remain more or less the same across training iterations. The key idea of our abstraction is to *roll* time around a circle whose *perimeter* is equal to the iteration time of a job. Consequently, the Up-Down phases of all iterations will appear on approximately the same angles of the circle.

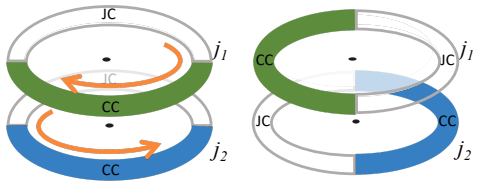
Figure 3(a) illustrates the time-series network demand of a data parallel VGG16 training job with a training iteration time of 255 ms. Figure 3(b) shows a circle with perimeter 255 units where the time-series data is plotted around it. The figure demonstrates that the Up and Down phases of different iterations cover the same angles of the circle. Our geometric abstraction captures this property, as shown in Figure 3(c). The perimeter of the circle is the iteration time, set to 255 units. The Down phase spans 141 units, represented by the uncolored arc with 200° angle, starting at 0°, on the x-axis. The Up phase represented by the colored arc occupies the remainder of the circle.

Rotate the circle to interleave Down and Up phases of different jobs. To determine the compatibility score of two (or more) jobs on a link, we place each job on its corresponding circle and overlay the circles on top of each other. Congestion occurs when the total bandwidth demand of a particular angle is higher than the capacity of the link, as shown in Figure 4(a). To find the best interleaving, we rotate the circles to a position where the summation of the bandwidth demands is less than the capacity of the link for all angles in the circle, as shown in Figure 4(b). If such a rotation is found, the jobs are fully compatible.



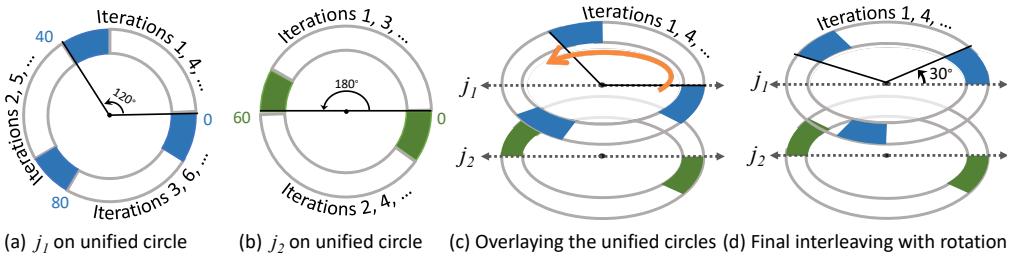
(a) Network demand (b) Network demand rolled around a circle (c) Geometric abstraction

Figure 3: CASSINI's geometric abstraction.



(a) CC phases are colliding (b) CC and JC phases are interleaved

Figure 4: Rotating the circles enables interleaving the network demand of j_1 and j_2.



(a) j_1 on unified circle (b) j_2 on unified circle (c) Overlaying the unified circles (d) Final interleaving with rotation

Figure 5: CASSINI's unified circles for jobs with different iteration times.

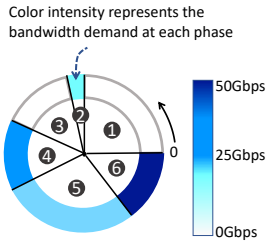


Figure 6: Geometric circle for the job in Fig. 1(d).

Capturing jobs with different iteration times. The above technique only works when the perimeters of the circles are the same. To generalize to the case where jobs have different iteration times, we place each job on a *unified circle* whose perimeter is equal to the Least Common Multiple (LCM) of the iteration time of all jobs competing on the link. For instance, consider two jobs j_1 and j_2 competing on a bottleneck link with iteration times 40 ms and 60 ms, respectively. To determine the compatibility score of the two jobs, we place them on a circle with a perimeter equal to $LCM(40,60) = 120$ units. Figure 5(a) shows j_1 on this unified circle. As the perimeter of the circle is $3\times$ j_1's iteration time, there are three periods of Up and Down phases in the figure. Similarly, Figure 5(b) shows j_2 on the unified circle. We then overlay the unified circles on top of each other (shown in Figure 5(c)) and rotate the circles to determine the compatibility score. Figure 5(d) shows that by rotating j_1 by $\Delta = 30°$ counter-clockwise, the sum of bandwidth demands on all angles of the unified circles is lower than the link capacity, giving these two jobs a compatibility score of 1 (i.e., fully compatible).

Capturing the bandwidth demand of model parallel training jobs. For clarity of presentation, the examples in this section contain data parallel training jobs with one Up and one Down phase during each iteration. However, CASSINI's geometric abstraction is generic and can capture more complex traffic patterns induced by various parallelization paradigms. Consider the communication pattern of the GPT-3 model with hybrid data/pipeline/tensor parallelism shown in Figure 1(d). Here, GPT-3's communication pattern consists of six Up-Down phases with different durations and bandwidth demands. The geometric circle of this job contains six colored arcs where the length and color intensity of each arc corresponds to the duration and bandwidth demand of each Up-Down phase of the model, as shown in Figure 6. Next, we formalize our geometric representation and show an optimization formulation that uses the geometric abstraction to find rotation angles to interleave the Up-Down phases of multiple jobs sharing a link, irrespective of the parallelization strategy.

Finding rotation angles. Once jobs are placed on their unified circles, CASSINI uses an optimization formulation, shown in Table 1, to find the best angle of rotation for jobs to maximize their compatibility.

Optimization input. The input is a set of ML jobs $J^l = \{j\}$ competing on a link l. We profile each job j to construct its unified circle, denoted by unified_circle$_j$. The perimeter of the unified circle is the LCM of the iteration times of all jobs $j \in J^l$. The data structure of unified_circle$_j$ contains a series of bandwidth demands $bw\_circle_j(\alpha)$, where $\alpha \in [0, 2\pi]$ identifies an arc of the circle that corresponds to an Up or Down phase in the communication pattern. The total capacity of link l is denoted by C^l.

Optimization objective and output. The optimization goal is to overlay the unified circles of each job and rotate them such that the excess bandwidth demand across all angles is minimized. We define the compatibility score as $score = 1 - \text{average}(Excess(demand_\alpha))$, where $Excess$ is the excess bandwidth demand of all jobs at a particular angle α (Equation 1). To make the score a unitless metric, we divide the average excess bandwidth by the link capacity C^l. Note

Input	$J^l = \{j\}$	Set of ML jobs $j \in J^l$ competing on link l.
	$\{\text{unified\_circle}_j\}$	Set of unified circles for $\forall j \in J$. Each circle is a data structure that contains the angles and bandwidth demand of Up or Down phases.
	$bw\_circle_j(\alpha)$	Bandwidth demand at angle α on unified_circle$_j$
	r_j	Number of iterations of j in its unified_circle$_j$.
	$A = \{\alpha\}$	Set of discrete angles $\alpha \in [0,2\pi]$. $\lvert A\rvert$ denotes the number of discrete angles.
	C^l	Total link capacity of link l.
Output	$demand_\alpha$	Total bandwidth demand at angle α when considering the demand of all jobs $j \in J$.
	Δ_j^l	Rotation angle of $j \in J$ on link l, in radians.
	$score$	Compatibility score of jobs sharing link l.

Auxiliary definitions:

$$Excess(demand_\alpha) = \begin{cases} demand_\alpha - C^l & if\, demand_\alpha > C^l \\ 0 & otherwise \end{cases} \quad (1)$$

Maximize: $$score = 1 - \frac{\sum_\alpha Excess(demand_\alpha)}{|A|C} \quad (2)$$

Subject to:

$$\forall \alpha : \sum_j bw\_circle_j(\alpha - \Delta_j^l) \leq demand_\alpha \quad (3)$$

$$\forall \Delta_j^l : 0 \leq \Delta_j^l \leq \frac{2\pi}{r_j} \quad (4)$$

Table 1: CASSINI's optimization formulation.

that when the excess bandwidth demand is zero, the compatibility score is 1 (i.e., 100% compatible). However, when there are many jobs with large excess bandwidth demands, it is possible for the score to become negative, indicating a highly incompatible combination. The optimization objective is to maximize this compatibility score, and the output of the optimization is a rotation angle Δ_j^l for each job.

Optimization constraints. Equation 3 computes the sum of the bandwidth demands across all the jobs sharing link l at a particular angle α on their unified circles, rotated by angle Δ_j^l. We bound this value by the output parameter $demand_\alpha$. Equation 4 bounds the rotation angle Δ_j^l between 0 and $\frac{2\pi}{r_j}$ because the unified_circle$_j$ contains r_j iterations of job j. Hence, setting an upper limit of $\frac{2\pi}{r_j}$ ensures that the rotation angle is in the first iteration and eliminates duplicate solutions.

4 Augmenting ML Schedulers with CASSINI

This section describes how CASSINI extends its link-level geometric abstraction to the entire cluster.

4.1 CASSINI Affinity Graph

Translating angular rotations to time-shifts. Consider a set of jobs $j \in J^l$ sharing link l. Using the formulation in Table 1, CASSINI computes a rotation angle Δ_j^l for $\forall j \in J^l$ such that the compatibility score is maximized. Each Δ_j^l corresponds to a time-shift t_j^l to delay the start time of j to maximize its compatibility with all other jobs in J^l. Given that the perimeter of the unified circle p^l, is the LCM of the iteration times of all jobs $j \in J^l$, CASSINI computes these time-shifts by multiplying the normalized rotation angle with p^l. Formally:

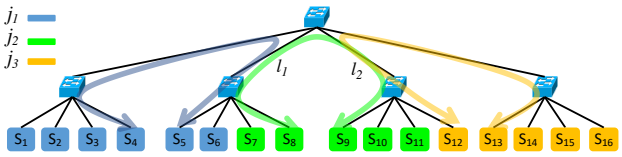


Figure 7: Example illustrating a cluster-scale compatibility challenge: CASSINI must ensure a unique time-shift for j_2.

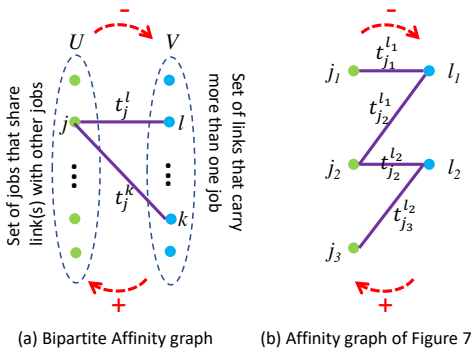


Figure 8: CASSINI's Affinity graph. Traversing left to right incurs a negative sign on the weight of edges and vice versa.

$$\forall j \in J^l, t_j^l = (\frac{\Delta_j^l}{2\pi} \times p^l) \mod iter\_time_j \quad (5)$$

Challenge: ensuring a unique time-shift for each job. In a large-scale cluster, jobs are likely to traverse multiple links, and they may compete with different jobs on different links. Consider the case depicted in Figure 7 where job j_1 competes with job j_2 on link l_1, and j_2 competes with job j_3 on link l_2. Theoretically, it is possible to migrate the jobs to pack workers of the same job under the same rack to avoid sharing the links altogether, but our experiments show that today's ML scheduling systems frequently end up with fragmented placements because of the dynamic nature of their scheduling decisions and job arrival patterns. In fact, no scheduler guarantees it can maintain perfect placement throughout time without continuously migrating jobs to defragment the cluster. For the case depicted in Figure 7, computing the time-shifts of j_2 using Equation 5 would result in two time-shift values $t_{j_2}^{l_1}$ and $t_{j_2}^{l_2}$. Given the interdependence between all servers participating in a training job, CASSINI must find a *unique* time-shift value for each job across links without compromising the compatibility on any link.

Simple approach. A potential approach to address the above challenge is to simply break the tie by choosing one of the t_j^l values at random. But this approach cancels out the benefits of compatibility because it does not respect the carefully computed time-shifts for different links.

Algorithm 1 Traversing the Affinity graph

```
1:  procedure BFSAffinityGraph
    ▷ Input Graph G = (U,V,E): Cassini's Affinity graph
    ▷ Output time_shifts_G: Time-shifts of jobs in G
2:      time_shifts_G = { }
3:      for all connected subgraphs H ∈ G, H = (U_H,V_H,E_H) do
4:          time_shifts_H = { }
            ▷ BFS traversal
5:          Mark all vertices u ∈ U_H as not-visited
6:          u = randomly_select_vertex(U_H)
7:          t_u = 0 and mark u as visited
            ▷ Only enqueue vertices from U (jobs)
8:          Q.enqueue(u)
9:          while Q is not empty do
10:             j = Q.dequeue()
                ▷ Find the corresponding links and jobs
11:             for all neighbors l of j do
12:                 for all neighbors k of l do
13:                     if k is not visited then
14:                         Q.enqueue(k) and mark k as visited
                            ▷ Find the edge from U to V
15:                         e_1 = E_H(j,l)
                            ▷ Find the edges from V to U
16:                         e_2 = E_H(l,k)
                            ▷ Compute the final time-shift
17:                         t_k = (t_j − w_{e_1} + w_{e_2}) % iter_time_k
18:                         time_shifts_H[k] = t_k
19:         time_shifts_G = time_shifts_G ∪ time_shifts_H
20:     return time_shifts_G
```

Complex approach. Another potential approach is to expand the footprint of our geometric abstraction from link-level to cluster-level. This approach requires expanding the optimization formulation in Table 1 to include all jobs that share their paths with any other jobs in the cluster and to encode a unique Δ_j in the constraints. This approach is not scalable because it requires expanding the perimeter of the unified circle to become the LCM of the iteration times of a large number of jobs in the cluster. Thus, finding a unique rotation angle for each job requires adding an exponential number of constraints to the optimization formulation which increases the complexity and overhead of the formulation dramatically.

Cassini's approach. Cassini introduces a bipartite Affinity graph $G = (U,V,E)$, where U and V are two sets of vertices, and E denotes the edge set between U and V, shown in Figure 8(a). Each vertex $u \in U$ represents a job that is sharing its path with other jobs somewhere in the network. Each vertex $v \in V$ represents a link that carries more than one job. An undirected edge $e = (j,l) \in E$ exists between a job $j \in U$ and a link $l \in V$ if j traverses l. The weight of edge $e = (j,l) \in E$ is the time-shift of job j on link l; i.e., $w_e = t_j^l$.

Traversing the Affinity graph. Cassini uses a graph traversal algorithm to find unique time-shifts t_j for all jobs $j \in J$ while maintaining compatibility on all links. To consolidate t_j^l values for each job j and link l into a unique t_j value,

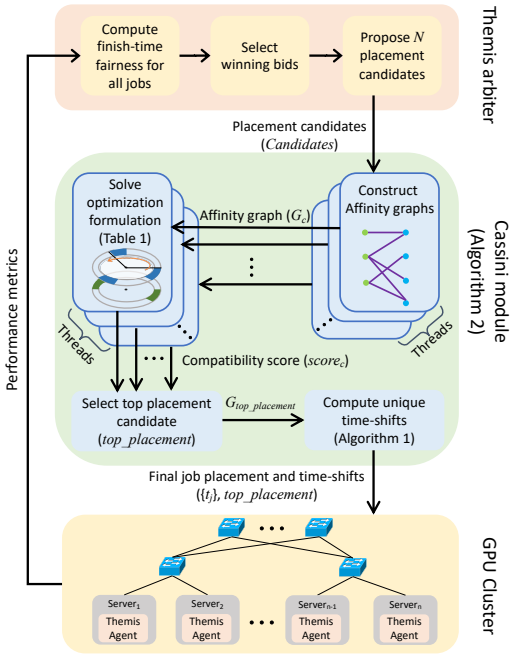


Figure 9: Using Cassini to augment Themis [40].

Cassini first randomly selects one of the jobs in the Affinity graph as the reference point with $t_j = 0$ and then traverses the graph to compute unique time-shifts for all others. Algorithm 1 describes the pseudocode of our graph traversal. In the general case, the Affinity graph is not necessarily a connected graph, hence, the algorithm traverses each connected subgraph separately (line 3). The traversal algorithm extends the Breadth First Search (BFS) algorithm in two ways. First, only vertices in U are added to the BFS queue (Q) because the time-shifts correspond to jobs, not links (lines 6- 14). Second, traversing from jobs ($j \in U$) to links ($l \in V$) incurs a negative sign on the t_j^l weight on edge $e = (j,l)$, whereas traversing the reverse direction incurs a positive sign (lines 15-18). As soon as the vertex corresponding to job j is visited, its unique time-shift is determined by the algorithm (line 18).

Theorem 1 (Correctness and Uniqueness Guarantee)**.** *Given a cluster with J jobs and a loop-free Affinity graph, $G = (U,V,E)$, Algorithm 1 guarantees both correct and unique time-shifts t_j for all jobs $j \in J$.*

Proof. The key insight behind this theorem is that our graph traversal maintains the same *relative* time-shift for all job pairs in the Affinity graph. The full proof uses induction and is provided in Appendix A, along with an example corresponding to the Affinity graph in Figure 8(b). □

4.2 Putting It All Together

This section uses Themis [40] as a running example of a scheduler augmented by Cassini.

Overview of Themis. Themis uses a fairness metric, called finish-time fairness, to achieve long-term fairness across the entire cluster by periodically updating the placement of jobs. To achieve fairness, workers in Themis lease resources and go through periodic auction epochs to help jobs that are farthest in terms of their fairness metric bid for more resources. Themis's central arbiter determines the global winning bids to maximize the aggregate improvement in the finish-time fair metrics across all bidding jobs. To capture network overheads, Themis uses a slowdown penalty based on whether the workers are under the same rack or across racks.

Augmenting Themis with CASSINI. Figure 9 shows how CASSINI augments Themis. First, CASSINI modifies Themis's arbiter to return a set of potential placement candidates instead of a single placement. Then, CASSINI selects the top placement candidate based on its compatibility metric and computes unique time-shifts for jobs that share the network. CASSINI transfers the time-shifts to Themis's agent running on servers. Finally, Themis's agent applies the time-shifts at the start of the epoch. Note that CASSINI respects the hyper-parameters, such as batch size or the number of workers, decided by Themis (or other schedulers that CASSINI is augmenting). Next, we describe each step in detail.

Step 1. Discover placement candidates. In this step, CASSINI decouples the process of finding the number of workers for each job to improve finish-time fairness from the exact worker placement in the cluster. To do so, instead of returning the precise job placements at the end of the auction phase, we configure Themis to return up to *N candidate placements*. These candidate placements all achieve the same finish-time fairness, but their worker placements are different. For instance, consider a case where jobs j_1 and j_2 each place a bid on two additional workers, and they both win, while job k_1 is losing one worker, and job k_2 is losing three. In this case, there are two ways to distribute workers: (i) k_1 and k_2 each give up one worker to j_1, and k_2 gives two workers to j_2; or (ii) k_1 and k_2 each give up one worker to j_2, and k_2 gives two workers to j_1. Both options are candidate placements. Moreover, selecting which workers in k_1 and k_2 should be reassigned creates another set of candidate placements. CASSINI collects these candidate placements and feeds them as input to the next step. This process requires changing only ≈300 lines of code in Themis.

Step 2. Find unique time-shifts. This step is listed in Algorithm 2 and includes CASSINI's key contributions. CASSINI first constructs an Affinity graph G_c for each placement candidate $c \in Candidates$ (lines 3-12). Following Theorem 1, to ensure correctness, we discard placement candidates with loop(s) in any of their Affinity subgraphs (line 15). Then, CASSINI constructs the unified circles for each job and solves the optimization formulation in Table 1 for all links in G_c to obtain the compatibility metric for each link in V_c (lines 17-22). Given that the placement candidates are independent of each other, our implementation uses multiple threads to parallelize this computation. Once the compatibility score of all candidate placements is determined, CASSINI sorts each placement candidate based on the average compatibility score of its member links to find the top placement candidate $top\_placement \in Candidates$ (lines 24-25).[1] Then, it executes Algorithm 1 on $top\_placement$'s Affinity graph $G_{top\_placement}$ to obtain unique time-shifts $\{t_j\}, \forall j \in V_{top\_placement}$ for jobs that share links with other jobs in this placement (line 26). Finally, $top\_placement$ and its corresponding time-shifts are transferred to Themis's agent running on the servers (line 27).

Step 3. Apply time-shifts. When a time-shift t_j is received by the Themis agent running job j, it delays the start of the

Algorithm 2 CASSINI Module's Pluggable Algorithm

```
1:  procedure CassiniModule
      ▷ Input Jobs: Array of active training jobs in the cluster
      ▷ Input Links: Array of all links in the cluster
      ▷ Input Candidates: Array of candidate placements for jobs
      ▷ Output top_placement, {t_j}: Top placement and time-shifts
2:    for c ∈ Candidates do  ▷ (Loop is executed with threads)
        ▷ Construct CASSINI's Affinity graph corresponding
        to this placement (§4.1)
3:      G_c = (U_c, V_c, E_c)
4:      for all j ∈ Jobs, l ∈ Links do
5:        if j shares links with other jobs then
6:          U_c = U_c ∪ j
7:          if l carries more than one job then
8:            V_c = V_c ∪ l
9:            if j is traversing l then
10:             e = new Edge between {(j, l)}
11:             E = E ∪ e
12:             w_e = 0
        ▷ Discard this candidate if Affinity graph has a loop
13:     if there is a loop in G_c then
14:       Candidates.remove(c)
15:       continue
16:     score_c = { }
17:     for l ∈ V_c do  ▷ (Executed with threads)
          ▷ List of jobs traversing link l
18:       J^l = {}
19:       for all neighbors j of l do
20:         J^l = J^l ∪ j
          ▷ Solve CASSINI optimization (Table 1)
21:       score_l = CassiniOptimization(J^l)
22:       score_c = score_c ∪ score_l
        ▷ Set the compatibility score of candidate c
23:     c.score = score_c
      ▷ Sort placements based on compatibility metric
24:   SortCandidates(Candidates, "Decreasing")
25:   top_placement = Candidate[0]
      ▷ Find unique time-shifts (Algorithm 1)
26:   {t_j} = BFSAffinityGraph(G_top_placement)
27:   return {t_j}, top_placement
```

[1] Instead of averaging, tail or other metrics may also be used.

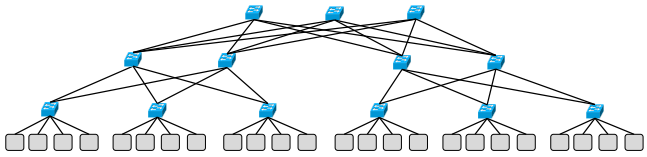


Figure 10: Logical topology of our testbed.

next immediate training iteration by t_j. However, even though the workers of the same job apply a unique time-shift, the time-shift values can drift due to noise, stragglers, and other unpredictable events. CASSINI updates the agent on each server to measure the drift and adjust the time-shifts. Our evaluations show that time-shift adjustments are rare (§5.7).

5 Evaluations

We evaluate CASSINI on a 24-server cluster and compare its performance to that of other state-of-the-art ML schedulers. First, we describe our evaluation methodology and setup (§5.1). Then, we compare CASSINI's performance gains with respect to the state-of-art ML schedulers for a mix of data and model parallel DNN training jobs (§5.2). Next, we evaluate the impact of data parallelism (§5.3), model parallelism (§5.4), partial compatibility (§5.5), and having multiple GPUs per server on CASSINI's performance (§5.6). Finally, we evaluate the frequency of time-shift adjustments and CASSINI's overhead (§5.7). CASSINI's source code is available at http://cassini.csail.mit.edu.

5.1 Methodology and Setup

Setup. We build a prototype to demonstrate the gains of CASSINI in real-world settings. Our prototype includes 24 ASUS ESC4000A-E10 servers, each with one A100 Nvidia GPU [8] (40 GB of HBM2 memory) and one 50 Gbps Mellanox ConnectX5 NIC. We use RoCEv2 for communication and enable DCB [5] and PFC on these interfaces to support a lossless fabric for RDMA. The servers run Ubuntu 18.04 LTS. We use PyTorch [36] version 1.8.0, CUDA version 11.1, and NCCL version 2.11.4 in our training framework.

Topology. We use a Tofino switch to construct the logical topology illustrated in Figure 10 with 13 logical switches. The Mellanox ConnectX5 NICs on each of the servers are connected to the Tofino switch. The Tofino switch emulates 13 logical switches and 48 bi-directional links for a 2:1 oversubscribed topology. We use flow table rules that match on <input port, destination MAC> to forward packets to the correct output port and physical loopback cables for switch-to-switch links. We use the default RDMA-based DCQCN congestion control algorithm [77]. ECN is enabled through WRED with min and max thresholds set to 1000 and 2000 cells. The PFC skid buffer threshold of each virtual switch is 4000 cells.

DNN workloads. We experiment with 13 popular DNN models: VGG11 [26], VGG16 [18], VGG19 [32], ResNet50 [27], WideResNet101 [72], BERT [20], RoBERTa [39], XLM [17], CamemBERT [43], GPT-1 [51], GPT-2 [52], GPT-3 [11], and DLRM [6]. All models have an equal probability of occurrence and the training duration time is randomly selected between 200 - 1,000 iterations. Table 3 (Appendix B) provides details about model configurations and batch sizes used in this paper.

Parallelization strategy. We use data parallelism to train the VGG, ResNet, and BERT family of models using Pytorch's DistributedDataParallel framework [38]. This framework distributes the dataset across GPUs and uses RingAllreduce to update the gradients during each training iteration. We train the DLRM and GPT family of models using a hybrid of data/model parallelism. Following prior work [66], we use Meta's opensource codebase for training DLRM [6] where the embedding tables are partitioned across GPUs, while the rest of the model is replicated on all GPUs. Finally, we use Microsoft's DeepSpeed tool [7] to partition the GPT models across GPUs using hybrid data/model parallelism.

Traces. Following prior work [40, 44, 50, 75], we use three sets of traces in our evaluations: (*i*) *Poisson trace*: we use a Poisson distribution for job arrivals where the job arrival time is determined by the load parameter defined as the average fraction of GPUs that are serving active jobs in the cluster. We vary the load between 80% and 100%; (*ii*) *dynamic trace*: where a set of DNN training jobs are present in the cluster, and a new set of jobs arrive; (*iii*) *snapshot trace*: we take several snapshots of the cluster where all jobs are present at the start of the experiment.

We implement the following schemes in our testbed.

• **Themis.** We use the default Themis [40] scheduler as one of our baselines. The bidding period (epoch) is set to 10 mins. Jobs participate in an auction where they send bid values for different GPU allocations. An assignment of GPU servers is valid until the period of lease time. When the lease time expires, the job gives up the server, and a new auction is conducted for all the released servers. When a job arrives, its initial number of requested workers is randomly selected between 1 to 12 GPUs. As the experiment progresses, the number of workers is automatically tuned based on Themis's finish-time-fairness metric.

• **Th+CASSINI.** Themis augmented with CASSINI as described in Section 4.2. In particular, this scheduler takes up to 10 placement candidates from Themis, constructs geometric circles and Affinity graphs for each placement to capture the cluster-level compatibility, solves our optimization formulation to find time-shifts for jobs that are competing on bandwidth, selects the top placement candidate based on compatibility ranks, and finally computes a unique time-shift for jobs. The unique time-shifts and final placement are given to the Themis agent running on GPUs. Unless otherwise stated, we use 5° as the angle discretization precision (Table 1) to compute the time-shifts.

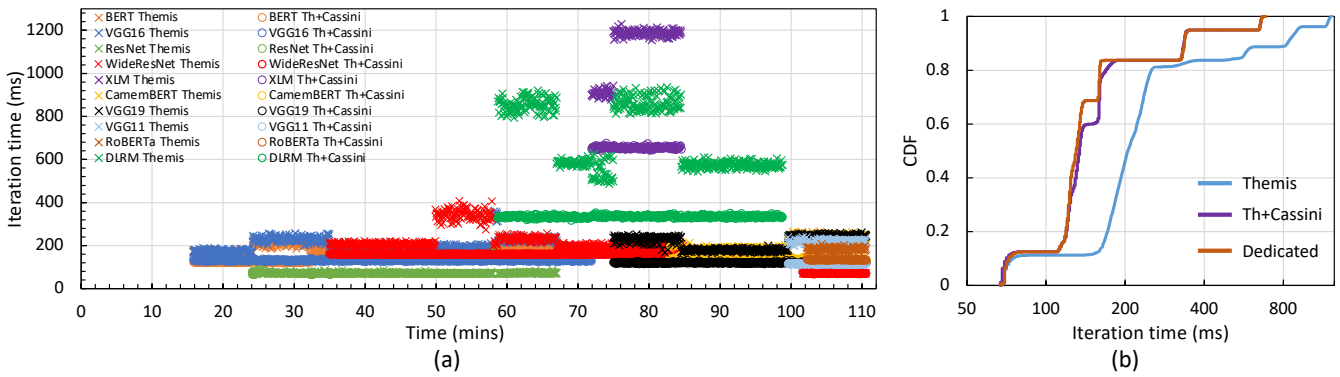


Figure 11: [Poisson trace] (a) Time series of DNN training jobs and their iteration times. (b) CDF of the iteration times.

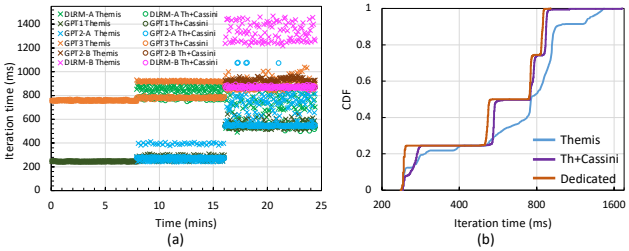


Figure 12: [Poisson trace] (a) Time series of model parallel jobs and their iteration times. (b) CDF of the iteration times.

• **Pollux.** We use Pollux as a second baseline [50]. Pollux considers the system throughput and statistical efficiency to maximize cluster-wide training performance. It periodically queries jobs and reassigns GPUs to maximize the overall goodput of the cluster. Pollux also models migration costs and avoids frequent job migrations.

• **Po+CASSINI.** We augment Pollux with CASSINI using an approach similar to that described in Section 4.2 except that Pollux uses overall goodput instead of finish-time-fairness to adjust hyper-parameters during scheduling epochs. Hence, the number of workers assigned to each job does not always agree with Themis. To make an apples-to-apples comparison, all CASSINI-related parameters in Po+CASSINI and Th+CASSINI are the same.

• **Ideal.** An ideal scheduler that runs each training job on a dedicated cluster. This scheduler incurs no congestion, as the entire cluster is dedicated to one job, and there is no need to take job compatibility into account.

• **Random.** A random placement scheduler that places workers for each job randomly. This scheduler has the highest network overhead, because it does not take locality or compatibility into account.

Profiling DNN models. Similar to Themis and Pollux, we profile each DNN using Pytorch and Infiniband port counters. Our profiling script executes a few iterations of each job to measure iteration times and collect link utilization patterns for various batch sizes and numbers of workers. Fine-grained link utilization data from the port counters enables CASSINI to build the geometric circles and the corresponding bandwidth demands for our optimization ($bw\_circle_j(\alpha)$ in Table 1).

5.2 Performance Gains

We evaluate CASSINI's performance gains using job arrivals and departures from our Poisson trace. Figure 11(a) plots the time series of events in the cluster for Themis and Th+CASSINI. In this experiment, we train a combination of DNN models. We use model parallelism for the DLRM [6] model because of its large model size, and we use data parallelism for all the other DNN models. Placement changes are triggered by job arrivals, job departures, and when the lease time of any of the servers expires. Given the dynamic nature of the trace, the servers are occupied gradually, and their lease times are not synchronized. For instance, at time $t = 72$ mins, a data parallel training job for the XLM [17] model arrives at the cluster, and Themis places it such that one of the links is shared with WideResNet101 [72] without the knowledge that XLM and WideResNet101 are not compatible jobs. In contrast, Th+CASSINI improves the iteration time of XLM by placing it with compatible jobs. Figure 11(b) plots the CDF of iteration times of all the data points in Figure 11(a) and shows that compared to Themis, Th+CASSINI improves the average and 99^{th} percentile tail iteration times by 1.6× and 1.8× respectively. We observe similar gains between Po+CASSINI and Pollux. The figure also shows that Th+CASSINI achieves similar performance as our Ideal benchmark.

To evaluate CASSINI's performance with model parallelism, we measure iteration times of various jobs trained using model parallelism, as shown in Figure 12(a). We use our Poisson trace for the job arrivals and departures. Note that this trace contains different training instances of the same DNN models where they differ in their hyper-parameters and number of workers (details in Appendix B). We use suffixes on their names to distinguish between the instances, for example, GPT2-A and GPT2-B are two different training jobs, as shown in the legend of Figure 12(a). GPT2-A has a batch

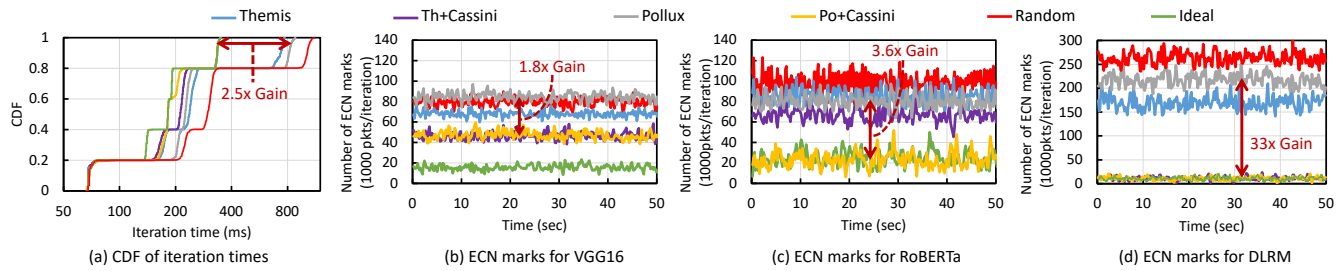


Figure 13: [Dynamic trace] CDF of training iteration times and the number of ECN marked packets per iteration.

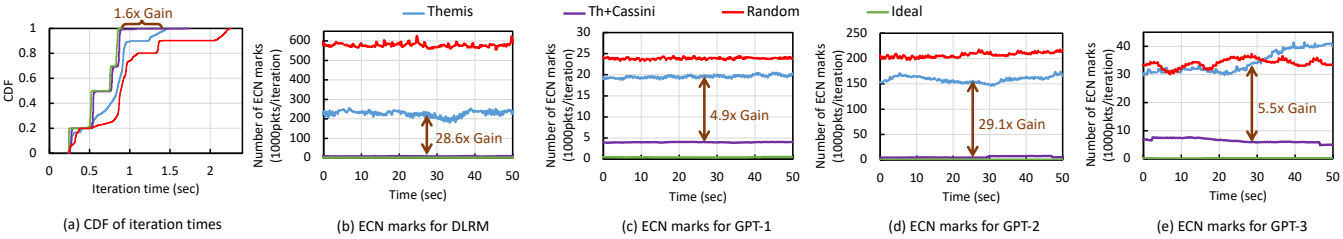


Figure 14: [Dynamic trace, model parallelism] CDF of training iteration times and the number of ECN marked packets.

size of 24 with a model hidden size of 1536 (as defined by Deepspeed's codebase [7]), while GPT2-B has a batch size of 70 with a model hidden size of 1184. For instance, at time $t = 8$ min, a model parallel GPT-2 [52] training job (labeled as GPT-2-A) arrives at the cluster and without considering the communication demands, Themis places this job such that it shares a link with another large GPT-3 [11] model in the cluster. GPT-2-A and GPT-3 models are not compatible, causing both training jobs to slow down. In contrast, Th+CASSINI improves GPT-2-A's iteration time by placing it with a compatible GPT-1 model. Figure 12(b) plots the CDF of iteration times of all the data points in Figure 12(a) and shows that compared to Themis, Th+CASSINI improves the average and 99^{th} percentile tail iteration times by 1.2× and 1.6× respectively.

5.3 CASSINI Reduces Congestion

We next demonstrate the effectiveness of CASSINI in reducing the congestion in the network. We use our dynamic trace to trigger the arrival of DLRM and ResNet50 to the cluster while the cluster is busy running other jobs. Given the contrast between the network demand between these two models, this experiment serves as a stress test to highlight the importance of compatible job placement on network congestion. In this case, both Pollux and Themis end up placing DLRM on servers that share network links with other non-compatible jobs which hurts the iteration times. In contrast, Th+CASSINI and Po+CASSINI flip the placements of DLM and ResNet50 to achieve compatibility, thereby improving the training iteration times, as depicted in Figure 13(a). Compared to Themis, Th+CASSINI improves the average and 99^{th} percentile tail iteration times by 1.5× and 2.2×, respectively. Similarly, compared to Pollux, Po+CASSINI improves the average and 99^{th} percentile tail iteration times by 1.6× and 2.5×, respectively.

The gains in iteration times are a direct consequence of CASSINI's ability to reduce network congestion. Figures 13(b) to (d) show the number of ECN marked packets per iteration for different models. The figure shows that Th+CASSINI and Po+CASSINI consistently maintain a lower number of ECN marks per iteration across the models. In particular, Figure 13(d) shows that, on average, DLRM is experiencing 27× and 33× more ECN marks in Themis and Pollux, compared to their CASSINI-augmented counterparts.

5.4 Impact of Model Parallelism

To ensure CASSINI's gains are not limited to data parallel jobs, we run a series of experiments in which all jobs in the trace use model parallelism. As shown in Section 2.1, model parallel jobs have several Up and Down phases in each iteration where the duration and bandwidth demand of each phase depends on the parallelization strategy and hyper-parameters. Similar to the data parallel case, we use CASSINI's geometric abstraction to capture the duration and bandwidth demand of Up and Down phases of a series of model parallel jobs. We then use CASSINI's optimization formulation and Affinity graph to compute the time-shifts for the jobs sharing the same network links. We use our dynamic trace to trigger the arrival of multiple GPT and DLRM models while the cluster is training other model parallel jobs.

Figure 14(a) shows the CDF of training iteration times. We find that similar to the data parallel case, Themis ends up placing non-compatible jobs, such as <GPT-3 and GPT-2> or <GPT-1 and DLRM>, on the same network link, whereas Th+CASSINI places compatible jobs, such as <GPT-1 and

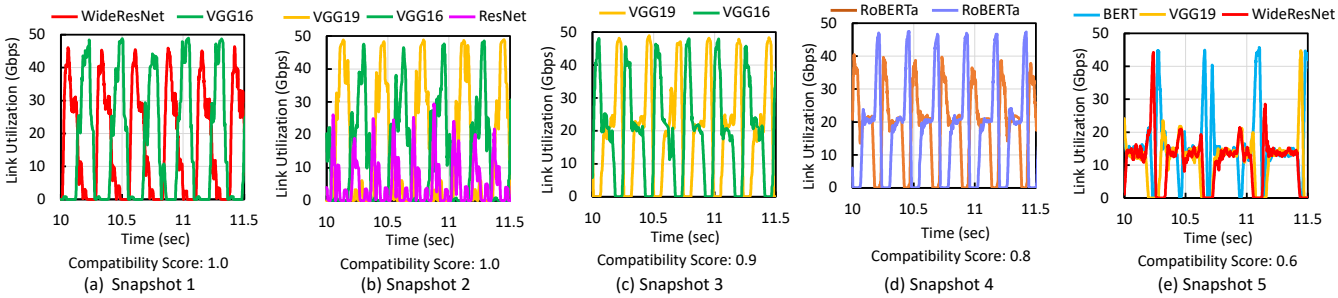


Figure 15: [Snapshot trace] Link utilization of compatible and partially compatible snapshots.

Snapshot ID	Competing jobs (batch size)	Th+CASSINI	Themis	Compatibility score	time-shift (ms)
1	WideResNet101 (800)	138 ms	205 ms	1.0	0 ms
	VGG16 (1400)	148 ms	199 ms		150 ms
2	VGG19 (1400)	168 ms	220 ms	1.0	0 ms
	VGG16 (1700)	163 ms	220 ms		158 ms
	RESNET50 (1600)	59 ms	55 ms		46 ms
3	VGG19 (1024)	166 ms	176 ms	0.9	0 ms
	VGG16 (1200)	168 ms	177 ms		100 ms
4	RoBERTa (12)	164 ms	210 ms	0.8	0 ms
	RoBERTa (12)	180 ms	208 ms		60 ms
5	BERT (8)	209 ms	213 ms	0.6	0 ms
	VGG19 (1400)	294 ms	292 ms		42 ms
	WideResNet101 (800)	265 ms	266 ms		191 ms

Table 2: [Snapshot trace] Compatibility score of DNN jobs.

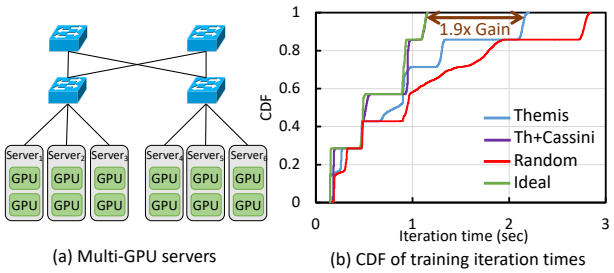


Figure 16: [Dynamic trace] Multi-GPU experiment.

GPT-2> or <GPT-3 and DLRM>, on the same network links. Consequently, Th+CASSINI improves the average and 99^{th} percentile tail iteration times by 1.2× and 1.6×, respectively. We observe similar gains between Pollux and Po+CASSINI.

Figures 14(b) to (e) depict the number of ECN marked packets per iteration for the models in this experiment. Depending on the status of congestion, different models experience different numbers of ECN marked packets. For instance, Figure 14(d) shows that compared to Themis, Th+CASSINI reduces the average number of ECN marked packets by 29×.

5.5 Impact of Partial Compatibility

An important consideration for practical deployment of CASSINI is to evaluate the impact of placing *partially* compatible jobs on the same link(s). Intuitively, the higher the compatibility score, the better interleaving is achieved. As the compatibility score reduces, the gains also diminish. To evaluate the impact of partial compatibility, we take five snapshots of the cluster, as shown in Table 2, and compute the compatibility scores and time-shift values from our optimization formulation (§3) for each snapshot. We then measure the average communication time of each model under Themis and Th+CASSINI. The table shows that when the compatibility score is 0.6, CASSINI's gain compared to Themis starts to diminish. Note that CASSINI avoids placing jobs with low compatibility score (e.g., snapshot 5) on the same link.

We demonstrate the reason behind diminishing returns by plotting the link utilization of each snapshot in Figure 15. When the compatibility score is high, the opportunity for interleaving is large, and jobs end up interleaving their network usage most of the time, as shown in Figures 15(a)–(d). However, as the compatibility score is reduced, jobs are forced to share the link most of the time, as shown in Figure 15(e). Additionally, Figure 15(b) demonstrates a desirable feature of our optimization formulation where compatibility does not require *strict* interleaving. In this snapshot, only VGG19 and VGG16 are interleaved, and ResNet's communications overlap with both jobs because its network demand is not significant and can co-exist with the other jobs.

5.6 Impact of Multiple GPUs per Server

Although having multiple GPUs per server enables allocating more GPUs within the same server to a job, today's large-scale training jobs require hundreds of workers [45, 66], making it impossible to avoid network congestion entirely by relying on multi-GPU servers. In such cases, CASSINI's gains are more pronounced for models that are distributed outside the boundary of a server.

We evaluate CASSINI's gains with multi-GPU servers by removing GPUs from some of our single-GPU servers and adding them to other servers to compose servers with two GPUs. We create a topology with six servers, each with two GPUs, as shown in Figure 16(a). We then use a mix of data parallel and model parallel jobs and generate a series of job arrivals using our dynamic trace.

Figure 16(b) demonstrates that compared to Themis, Th+CASSINI improves the average and 99^{th} percentile tail iteration times by 1.4× and 1.9×, respectively. These gains are

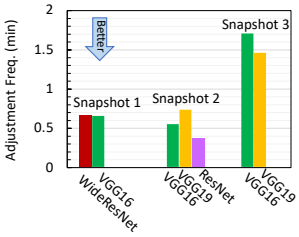


Figure 17: [Snapshot trace] The frequency of adjusting time-shifts for snapshots 1–3.

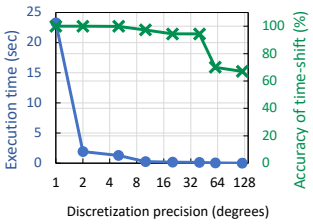


Figure 18: Impact of angle discretization on execution time and time-shift accuracy.

achieved because some jobs require more than two GPUs to train. For instance, at a particular instant in our dynamic trace, the XLM and ResNet50 models each require three GPUs to train. With the arrival of a network-intensive model DLRM requesting three more GPUs, Themis decides to place DLRM such that it shares a server with a non-compatible model (XLM), making both jobs experience congestion. In contrast, Th+CASSINI selects a placement where DLRM shares a link with a compatible model (ResNet50), thereby improving the training iteration times of both models.

5.7 Adjusting Time-Shifts and Overhead

To maintain CASSINI's interleaving, workers must respect the time-shift values given to them through the scheduler. Given that our servers are not running perfectly in sync, we evaluate the frequency of automatic time-shift adjustments by the Themis (or Pollux) agents running on the servers. Note that respecting the time-shift is only required for compatible jobs. All other jobs in the cluster can send packets at any time. A worker triggers an adjustment when the start of the communication phase deviates by more than five percent of the ideal iteration time. Figure 17 shows the average frequency of time-shift adjustments for snapshots 1,2, and 3. In all cases, the frequency is less than two adjustments per minute.

Finally, we evaluate the impact of angle discretization precision on CASSINI's optimization formulation (Table 1). Intuitively, the execution time of a coarse-grained discretization is fast but such formulation misses interleaving opportunities, thereby finding imprecise rotation angles. Given that CASSINI's time-shifts are driven from rotation angles, a coarse-grained formulation leads to inaccurate time-shifts. On the other hand, having fine-grained precision leads to more accurate time-shifts at the expense of a longer execution time. Figure 18 demonstrates this trend and shows that using a precision of 5° is the sweet spot for achieving 100% accuracy for time-shifts while maintaining a low execution overhead.

6 Discussion and Limitations

Sharing with legacy datacenter workloads. We assume the ML training traffic is not sharing the network with non-ML legacy datacenter workloads, such as websearch, indexing, cloud, and storage. We believe this is reasonable because modern training clusters consist of custom-designed servers, each with dedicated NICs for training traffic (GPU NICs) and additional NICs for storage and other traffic (CPU NICs) [4, 45]. The CPU NICs are often connected through a separate fabric to carry storage and other control plane traffic. Our abstraction and time-shift values only affect the GPU NICs.

GPU multi-tenancy. For simplicity, we assume GPUs are dedicated resources for each job, and different jobs are not sharing the same GPU – this is not far from how many production clusters run today to ensure predictable and high-throughput training performance. Thus, our geometric abstraction only considers the network links as shared resources and allows the Down (Just Compute) phases of different jobs to overlap. Recent proposals have demonstrated the feasibility of multi-tenancy on GPUs [9, 67, 69, 70]. We note that capturing GPU multi-tenancy is possible by adding more constraints in our optimization formulation, but we omit the details for brevity.

Scaling. Scaling the number of GPUs on each server enables service providers to pack jobs within fewer servers, thereby reducing the chances of network congestion. In recent years, the compute requirements of DNN models are growing exponentially [3]. Training models across multiple servers is inevitable with growing model and dataset sizes. For example, large models like AlphaGo [61] and AlphaZero [60] are trained using hundreds to thousands of GPUs and TPUs. We expect CASSINI's gains to remain consistent for clusters with multiple GPUs per server, but we leave further investigation to future work.

CASSINI advocates placing jobs such that jobs with higher compatibility scores share network links. However, as the number of jobs sharing a network link increases, it becomes harder to interleave the communication demands, and the compatibility score reduces. CASSINI tries to avoid scenarios where jobs with low compatibility scores share a network link. We leave the study of the effect of the number of jobs sharing a network link on the compatibility scores for future work.

7 Related Work

Our work builds on several lines of related research.

Compute scheduling approaches. A large number of systems and techniques have focused on improving the performance of large-scale distributed ML workloads [13, 16, 19, 23, 24, 36, 42, 54, 63, 68, 71, 73]. Relevant to this paper, several papers aim to reduce communication overhead using smart scheduling techniques; e.g., Gandiva [69], Themis [40], Pollux [50], Tiresias [25], Shockwave [76], and Optimus [48]. These schedulers try to minimize network sharing by placing workers of the same job as close as possible to each other. However, these approaches do not consider interleaving the communication patterns of different training jobs when placing them on servers. CASSINI's contribution is complementary to these approaches by considering both the compute resources and the communication demands of different jobs

during scheduling. Moreover, CASSINI is designed as a pluggable module to augment these schedulers.

Multi-resource sharing. Recently, Muri [75] proposed a scheduling technique to interleave critical resources (e.g., GPU, CPU, network, storage) of DNN training jobs. Muri packs jobs that are being executed *on the same set of resources* into a group and interleaves their resource requirements using a Blossom-based scheduler. However, Muri's approach to resource interleaving only applies to jobs that share the same set of GPUs, CPUs, memory, and network resources.[2] Hence, Muri can interleave compute and communication phases of a set of jobs only if the jobs *are sharing the same set of GPUs*. In contrast, CASSINI is able to interleave compute and communication phases of different jobs, irrespective of which GPUs they occupy. For instance, Muri's algorithm is not applicable to interleave the resources of j_1 and j_2 in Figure 2(a), because j_1 is distributed between server$_1$ and server$_2$ while j_2 is distributed between server$_3$ and server$_4$; i.e., these two jobs do not belong to the same resource group in Muri's algorithm. Muri would have been able to interleave the resources if both j_1 and j_2 were distributed between all four servers. However, for many of the large models we use in our experiments, GPU-sharing is not possible because of the memory requirements of the model. Moreover, even with GPU sharing, in a large-scale cluster, cross-group network congestion is common. CASSINI is able to interleave the Up and Down phases of different jobs, without requiring them to share the same set of resources. Similarly, Synergy [44] has proposed a multi-resource interleaving scheduling approach by inferring the sensitivity of DNN jobs to GPU, CPU, and memory resources using optimistic profiling. Synergy improves the overall cluster efficiency by performing resource-sensitive allocations instead of a GPU-proportional allocation. However, Synergy's approach does not consider the network bandwidth as a resource and is unable to interleave the communication phases with other resources. In contrast, CASSINI's focus is on interleaving the network demand with the GPU resources. CASSINI is designed to augment both Muri and Synergy schedulers. Some previous studies have concentrated on the theoretical analysis of periodic tasks [22, 34]. However, these approaches exploit characteristics distinct from those inherent to distributed DNN training jobs.

Communication-aware scheduling. A variety of approaches have been developed to accelerate communication among ML training workers of the same job to reduce network overhead [2, 14, 23, 33, 45, 55, 65, 74] and to enable more efficient pipelining strategies [28, 47]. ByteScheduler [49] and Syndicate [41] accelerate ML training by scheduling and optimizing the order of communication operations between different GPU servers used by a training job. ByteScheduler overlaps compute and communication operations *within a training job*, while Syndicate provides a solution for planning and scheduling communication operations for large DNN training. Similarly, TACCL [56], BytePS [30], and CLOPT [74] improve the communication collective algorithms of large DNN models. BytePS seeks to find a balance between the Parameter Server [37] and Ring-AllReduce algorithms for synchronizing the gradients. TACCL proposes a communication collective algorithm for training large models with data and model parallelism. CLOPT co-optimizes network topology and communication schedules for ML training. These approaches optimize communication *within a training job*, however, they do not consider congestion and network sharing *across training jobs*. In contrast, CASSINI's approach is orthogonal to these techniques because CASSINI focuses on sharing the network resources *across different training jobs*.

Difference with prior workshop paper. A prior workshop paper [53] introduced the idea of using a geometric abstraction to achieve job compatibility at a single-link level. We extend this workshop paper in a few important ways. First, the workshop paper considers compute/communication interleaving at a high level and does not provide a concrete scheduling technique to achieve it. Specifically, it relies on an unfair congestion control protocol to achieve interleaving, but CASSINI does not require any changes to or assumptions about the congestion control protocol. Second, the workshop paper ignores the impact of cluster-level interleaving. Third, the workshop paper only considers the data parallelism paradigm, and its geometric abstraction does not generalize to model parallelism techniques. Finally, our optimization formulation, the Affinity graph abstraction, the design and implementation of the CASSINI module, and our formal arguments around correctness (Theorem 1) are all new contributions.

8 Conclusion

CASSINI is a simple but effective approach that can integrate with existing cluster schedulers to allow them to accommodate multiple ML jobs' network needs. We introduce a novel metric, called compatibility score, to rank different GPU placements when jobs compete on network links. Our evaluations show that CASSINI improves the average and tail completion time of jobs by up to 1.6× and 2.5×, respectively. Moreover, we show that CASSINI reduces the number of ECN marked packets by up to 33×.

Acknowledgements. We thank NSDI's anonymous reviewers and our shepherd, Xin Jin, for their valuable feedback. Thanks to Gautam Kumar, Frank Wang, Benoit Pit–Claudel, Venkat Arun, and Kapil Vaidya for helpful suggestions and discussions. The MIT-affiliated authors were supported in part by ACE and CUbiC, two of the seven centers in JUMP 2.0, a Semiconductor Research Corporation (SRC) program sponsored by DARPA, as well as NSF SHF-2107244, NSF ASCENT-2023468, NSF CAREER-2144766, NSF PPoSS-2217099, NSF CNS-2211382, and Sloan fellowship FG-2022-18504. Akella was supported by NSF grants CNS-2214015 and CNS-2207317, and by a gift from Cisco Research.

[2] Muri [75] states this limitation: "The algorithm avoids cross-group packing to minimize the packing overhead."

References

[1] MLPerf: A broad ML benchmark suite. https://mlperf.org/.

[2] Baidu, 2017. https://github.com/baidu-research/baidu-allreduce.

[3] AI and Compute, 2020. https://openai.com/blog/ai-and-compute/.

[4] Nvidia DGX SuperPOD, 2020. https://www.nvidia.com/en-us/data-center/dgx-superpod/.

[5] Data Center Bridging eXchange (DCBX), 2021. https://man7.org/linux/man-pages/man8/dcb-dcbx.8.html.

[6] Deep Learning Recommendation Model for Personalization and Recommendation Systems, 2021. https://github.com/facebookresearch/dlrm.

[7] DeepSpeed version of NVIDIA's Megatron-LM, 2021. https://github.com/microsoft/Megatron-DeepSpeed.

[8] NVIDIA A100 Tensor Core GPU, 2021. https://www.nvidia.com/en-us/data-center/a100/.

[9] R. Ausavarungnirun, V. Miller, J. Landgraf, S. Ghose, J. Gandhi, A. Jog, C. J. Rossbach, and O. Mutlu. Mask: Redesigning the gpu memory hierarchy to support multi-application concurrency. *SIGPLAN Not.*, 53(2):503–518, mar 2018.

[10] T. Ben-Nun and T. Hoefler. Demystifying parallel and distributed deep learning: An in-depth concurrency analysis. *ACM Comput. Surv.*, 52(4), aug 2019.

[11] T. B. Brown, B. Mann, N. Ryder, M. Subbiah, J. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, S. Agarwal, A. Herbert-Voss, G. Krueger, T. Henighan, R. Child, A. Ramesh, D. M. Ziegler, J. Wu, C. Winter, C. Hesse, M. Chen, E. Sigler, M. Litwin, S. Gray, B. Chess, J. Clark, C. Berner, S. McCandlish, A. Radford, I. Sutskever, and D. Amodei. Language models are few-shot learners. *CoRR*, abs/2005.14165, 2020.

[12] M. A. Chang, A. Panda, D. Bottini, L. Jian, P. Kumar, and S. Shenker. Network evolution for dnns. *SysML*, 2018.

[13] T. Chilimbi, Y. Suzue, J. Apacible, and K. Kalyanaraman. Project adam: Building an efficient and scalable deep learning training system. In *OSDI'14*, pages 571–582, 2014.

[14] M. Cho, U. Finkler, D. Kung, and H. Hunter. Blueconnect: Decomposing all-reduce for deep learning on heterogeneous network hierarchy. SysML Conference, 2019.

[15] E. Chung, J. Fowers, K. Ovtcharov, M. Papamichael, A. Caulfield, T. Massengil, M. Liu, D. Lo, S. Alkalay, and M. Haselman. Accelerating persistent neural networks at datacenter scale. In *Hot Chips*, volume 29, 2017.

[16] A. Coates, B. Huval, T. Wang, D. Wu, B. Catanzaro, and N. Andrew. Deep learning with cots hpc systems. In *International conference on machine learning*, pages 1337–1345, 2013.

[17] A. CONNEAU and G. Lample. Cross-lingual language model pretraining. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 32. Curran Associates, Inc., 2019.

[18] A. K. Dash. VGG-16 Architecture. https://iq.opengenus.org/vgg16/.

[19] J. Dean, G. Corrado, R. Monga, K. Chen, M. Devin, M. Mao, A. Senior, P. Tucker, K. Yang, Q. V. Le, et al. Large scale distributed deep networks. In *Advances in neural information processing systems*, pages 1223–1231, 2012.

[20] J. Devlin, M. Chang, K. Lee, and K. Toutanova. BERT: pre-training of deep bidirectional transformers for language understanding. *CoRR*, abs/1810.04805, 2018.

[21] A. Gholami, A. Azad, K. Keutzer, and A. Buluç. Integrated model and data parallelism in training neural networks. *CoRR*, abs/1712.04432, 2017.

[22] J. Goossens. Scheduling of offset free systems. *Real-Time Systems*, 2003.

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

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

[25] J. Gu, M. Chowdhury, K. G. Shin, Y. Zhu, M. Jeon, J. Qian, H. Liu, and C. Guo. Tiresias: A GPU cluster manager for distributed deep learning. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 485–500, Boston, MA, Feb. 2019. USENIX Association.

[26] S. Gupta. VGG-11 Architecture. https://iq.opengenus.org/vgg-11/.

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

[28] Y. Huang, Y. Cheng, D. Chen, H. Lee, J. Ngiam, Q. V. Le, and Z. Chen. Gpipe: Efficient training of giant neural networks using pipeline parallelism. *NeurIPS*, 2019.

[29] Z. Jia, S. Lin, C. R. Qi, and A. Aiken. Exploring hidden dimensions in accelerating convolutional neural networks. In J. Dy and A. Krause, editors, *Proceedings of the 35th International Conference on Machine Learning*, volume 80 of *Proceedings of Machine Learning Research*, pages 2274–2283, Stockholmsmässan, Stockholm Sweden, 10–15 Jul 2018. PMLR.

[30] Y. Jiang, Y. Zhu, C. Lan, B. Yi, Y. Cui, and C. Guo. A unified architecture for accelerating distributed DNN training in heterogeneous gpu/cpu clusters. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 463–479. USENIX Association, Nov. 2020.

[31] C. Karakus, R. Huilgol, F. Wu, A. Subramanian, C. Daniel, D. Cavdar, T. Xu, H. Chen, A. Rahnama, and L. Quintela. Amazon sagemaker model parallelism: A general and flexible framework for large model training, 2021.

[32] A. Kaushik. VGG-19 Architecture. https://iq.opengenus.org/vgg19-architecture/.

[33] M. Khani, M. Ghobadi, M. Alizadeh, Z. Zhu, M. Glick, K. Bergman, A. Vahdat, B. Klenk, and E. Ebrahimi. Sip-ml: High-bandwidth optical network interconnects for machine learning training. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*, SIGCOMM '21, pages 657–675, New York, NY, USA, 2021. Association for Computing Machinery.

[34] M. Ladeira, E. Grolleau, F. Bonneval, G. Hattenberger, Y. Ouhammou, and Y. Hérouard. Scheduling Offset-Free Systems Under FIFO Priority Protocol. In M. Maggio, editor, *34th Euromicro Conference on Real-Time Systems (ECRTS 2022)*, volume 231 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 11:1–11:19, Dagstuhl, Germany, 2022. Schloss Dagstuhl – Leibniz-Zentrum für Informatik.

[35] S. Lee, J. K. Kim, X. Zheng, Q. Ho, G. A. Gibson, and E. P. Xing. On model parallelization and scheduling strategies for distributed machine learning. In Z. Ghahramani, M. Welling, C. Cortes, N. Lawrence, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems*, volume 27, pages 2834–2842. Curran Associates, Inc., 2014.

[36] A. Lerer, L. Wu, J. Shen, T. Lacroix, L. Wehrstedt, A. Bose, and A. Peysakhovich. Pytorch-biggraph: A large-scale graph embedding system. *CoRR*, abs/1903.12287, 2019.

[37] M. Li, D. G. Andersen, J. W. Park, A. J. Smola, A. Ahmed, V. Josifovski, J. Long, E. J. Shekita, and B.-Y. Su. Scaling distributed machine learning with the parameter server. OSDI'14, pages 583–598. USENIX Association, 2014.

[38] S. Li, Y. Zhao, R. Varma, O. Salpekar, P. Noordhuis, T. Li, A. Paszke, J. Smith, B. Vaughan, P. Damania, and S. Chintala. Pytorch distributed: Experiences on accelerating data parallel training. *Proc. VLDB Endow.*, 13(12):3005–3018, aug 2020.

[39] Y. Liu, M. Ott, N. Goyal, J. Du, M. Joshi, D. Chen, O. Levy, M. Lewis, L. Zettlemoyer, and V. Stoyanov. Roberta: A robustly optimized bert pretraining approach, 2019.

[40] K. Mahajan, A. Balasubramanian, A. Singhvi, S. Venkataraman, A. Akella, A. Phanishayee, and S. Chawla. Themis: Fair and efficient GPU cluster scheduling. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pages 289–304, Santa Clara, CA, Feb. 2020. USENIX Association.

[41] K. Mahajan, C.-H. Chu, S. Sridharan, and A. Akella. Better together: Jointly optimizing ML collective scheduling and execution planning using SYNDICATE. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pages 809–824, Boston, MA, Apr. 2023. USENIX Association.

[42] L. Mai, C. Hong, and P. Costa. Optimizing network performance in distributed machine learning. In *7th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 15)*, Santa Clara, CA, 2015. USENIX Association.

[43] L. Martin, B. Müller, P. J. O. Suárez, Y. Dupont, L. Romary, É. V. de la Clergerie, D. Seddah, and B. Sagot. Camembert: a tasty french language model. *CoRR*, abs/1911.03894, 2019.

[44] J. Mohan, A. Phanishayee, J. J. Kulkarni, and V. Chidambaram. Looking beyond gpus for dnn scheduling on multi-tenant clusters. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI 2022)*, July 2022.

[45] D. Mudigere, Y. Hao, J. Huang, Z. Jia, A. Tulloch, S. Sridharan, X. Liu, M. Ozdal, J. Nie, J. Park, L. Luo, J. A. Yang, L. Gao, D. Ivchenko, A. Basant, Y. Hu, J. Yang, E. K. Ardestani, X. Wang, R. Komuravelli, C.-H. Chu, S. Yilmaz, H. Li, J. Qian, Z. Feng, Y. Ma, J. Yang, E. Wen, H. Li, L. Yang, C. Sun, W. Zhao, D. Melts, K. Dhulipala, K. Kishore, T. Graf, A. Eisenman, K. K. Matam, A. Gangidi, G. J. Chen, M. Krishnan, A. Nayak, K. Nair, B. Muthiah, M. khorashadi, P. Bhattacharya, P. Lapukhov, M. Naumov, L. Qiao, M. Smelyanskiy, B. Jia, and V. Rao. Software-hardware co-design for fast and scalable training of deep learning recommendation models, 2021.

[46] S. Narasimhan. NVIDIA Clocks World's Fastest BERT Training Time and Largest Transformer Based Model, Paving Path For Advanced Conversational AI, Aug. 2019. https://devblogs.nvidia.com/training-bert-with-gpus/.

[47] D. Narayanan, A. Harlap, A. Phanishayee, V. Seshadri, N. R. Devanur, G. R. Ganger, P. B. Gibbons, and M. Zaharia. Pipedream: Generalized pipeline parallelism for dnn training. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, SOSP'19, pages 1–15, New York, NY, USA, 2019. Association for Computing Machinery.

[48] Y. Peng, Y. Bao, Y. Chen, C. Wu, and C. Guo. Optimus: An efficient dynamic resource scheduler for deep learning clusters. In *Proceedings of the Thirteenth EuroSys Conference*, EuroSys '18, New York, NY, USA, 2018. Association for Computing Machinery.

[49] Y. Peng, Y. Zhu, Y. Chen, Y. Bao, B. Yi, C. Lan, C. Wu, and C. Guo. A generic communication scheduler for distributed dnn training acceleration. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, SOSP '19, page 16–29, New York, NY, USA, 2019. Association for Computing Machinery.

[50] A. Qiao, S. K. Choe, S. J. Subramanya, W. Neiswanger, Q. Ho, H. Zhang, G. R. Ganger, and E. P. Xing. Pollux: Co-adaptive cluster scheduling for goodput-optimized deep learning. In *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*, pages 1–18. USENIX Association, July 2021.

[51] A. Radford, K. Narasimhan, T. Salimans, and I. Sutskever. Improving language understanding by generative pre-training. https://s3-us-west-2.amazonaws.com/openai-assets/research-covers/language-unsupervised/language_understanding_paper.pdf.

[52] A. Radford, J. Wu, R. Child, D. Luan, D. Amodei, and I. Sutskever. Language models are unsupervised multitask learners. 2018. https://d4mucfpksywv.cloudfront.net/better-language-models/language-models.pdf.

[53] S. Rajasekaran, M. Ghobadi, G. Kumar, and A. Akella. Congestion Control in Machine Learning Clusters. In *Proceedings of the 21st ACM Workshop on Hot Topics in Networks*, HotNets '22, page 235–242, 2022.

[54] A. Sapio, M. Canini, C.-Y. Ho, J. Nelson, P. Kalnis, C. Kim, A. Krishnamurthy, M. Moshref, D. Ports, and P. Richtarik. Scaling distributed machine learning with In-Network aggregation.

In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*, pages 785–808. USENIX Association, Apr. 2021.

[55] A. Sergeev and M. D. Balso. Horovod: fast and easy distributed deep learning in tensorflow. *CoRR*, abs/1802.05799, 2018.

[56] A. Shah, V. Chidambaram, M. Cowan, S. Maleki, M. Musuvathi, T. Mytkowicz, J. Nelson, O. Saarikivi, and R. Singh. TACCL: Guiding collective algorithm synthesis using communication sketches. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pages 593–612, Boston, MA, Apr. 2023. USENIX Association.

[57] C. J. Shallue, J. Lee, J. M. Antognini, J. Sohl-Dickstein, R. Frostig, and G. E. Dahl. Measuring the effects of data parallelism on neural network training. *CoRR*, abs/1811.03600, 2018.

[58] N. Shazeer, Y. Cheng, N. Parmar, D. Tran, A. Vaswani, P. Koanantakool, P. Hawkins, H. Lee, M. Hong, C. Young, R. Sepassi, and B. Hechtman. Mesh-tensorflow: Deep learning for supercomputers. In S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 31. Curran Associates, Inc., 2018.

[59] M. Shoeybi, M. Patwary, R. Puri, P. LeGresley, J. Casper, and B. Catanzaro. Megatron-lm: Training multi-billion parameter language models using model parallelism, 2020.

[60] D. Silver, T. Hubert, J. Schrittwieser, I. Antonoglou, M. Lai, A. Guez, M. Lanctot, L. Sifre, D. Kumaran, T. Graepel, T. Lillicrap, K. Simonyan, and D. Hassabis. Mastering chess and shogi by self-play with a general reinforcement learning algorithm, 2017.

[61] D. Silver, J. Schrittwieser, K. Simonyan, I. Antonoglou, A. Huang, A. Guez, T. Hubert, L. Baker, M. Lai, A. Bolton, Y. Chen, T. Lillicrap, F. Hui, L. Sifre, G. van den Driessche, T. Graepel, and D. Hassabis. Mastering the game of go without human knowledge. *Nature*, 2017.

[62] K. Simonyan and A. Zisserman. Very deep convolutional networks for large-scale image recognition, 2015.

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

[64] B. Wang, Q. Xu, Z. Bian, and Y. You. Tesseract: Parallelize the tensor parallelism efficiently. In *Proceedings of the 51st International Conference on Parallel Processing*. ACM, aug 2022.

[65] G. Wang, S. Venkataraman, A. Phanishayee, J. Thelin, N. Devanur, and I. Stoica. Blink: Fast and generic collectives for distributed ml. In *Conference on Machine Learning and Systems (MLSys 2020)*, March 2020.

[66] W. Wang, M. Khazraee, Z. Zhong, M. Ghobadi, Z. Jia, D. Mudigere, Y. Zhang, and A. Kewitsch. TopoOpt: Co-optimizing network topology and parallelization strategy for distributed training jobs. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pages 739–767, Boston, MA, Apr. 2023. USENIX Association.

[67] Q. Weng, W. Xiao, Y. Yu, W. Wang, C. Wang, J. He, Y. Li, L. Zhang, W. Lin, and Y. Ding. MLaaS in the wild: Workload analysis and scheduling in Large-Scale heterogeneous GPU clusters. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, pages 945–960, Renton, WA, Apr. 2022. USENIX Association.

[68] Z. Xia, Y. Zhou, F. Y. Yan, and J. Jiang. Genet: Automatic curriculum generation for learning adaptation in networking. In *Proceedings of the ACM SIGCOMM 2022 Conference*, SIGCOMM '22, page 397–413, New York, NY, USA, 2022. Association for Computing Machinery.

[69] W. Xiao, R. Bhardwaj, R. Ramjee, M. Sivathanu, N. Kwatra, Z. Han, P. Patel, X. Peng, H. Zhao, Q. Zhang, F. Yang, and L. Zhou. Gandiva: Introspective cluster scheduling for deep learning. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 595–610, Carlsbad, CA, Oct. 2018. USENIX Association.

[70] W. Xiao, S. Ren, Y. Li, Y. Zhang, P. Hou, Z. Li, Y. Feng, W. Lin, and Y. Jia. AntMan: Dynamic scaling on GPU clusters for deep learning. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 533–548. USENIX Association, Nov. 2020.

[71] M. Yang, A. Baban, V. Kugel, J. Libby, S. Mackie, S. S. R. Kananda, C.-H. Wu, and M. Ghobadi. Using trio: Juniper networks' programmable chipset - for emerging in-network applications. In *Proceedings of the ACM SIGCOMM 2022 Conference*, SIGCOMM '22, page 633–648, New York, NY, USA, 2022. Association for Computing Machinery.

[72] S. Zagoruyko and N. Komodakis. Wide residual networks, 2016.

[73] H. Zhao, Z. Han, Z. Yang, Q. Zhang, F. Yang, L. Zhou, M. Yang, F. C. Lau, Y. Wang, Y. Xiong, and B. Wang. HiveD: Sharing a GPU cluster for deep learning with guarantees. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 515–532. USENIX Association, Nov. 2020.

[74] L. Zhao, S. Pal, T. Chugh, W. Wang, P. Basu, J. Khoury, and A. Krishnamurthy. Optimal direct-connect topologies for collective communications, 2022.

[75] Y. Zhao, Y. Liu, Y. Peng, Y. Zhu, X. Liu, and X. Jin. Multi-resource interleaving for deep learning training. In *Proceedings of the ACM SIGCOMM 2022 Conference*, SIGCOMM '22, page 428–440, New York, NY, USA, 2022. Association for Computing Machinery.

[76] P. Zheng, R. Pan, T. Khan, S. Venkataraman, and A. Akella. Shockwave: Fair and efficient cluster scheduling for dynamic adaptation in machine learning. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pages 703–723, Boston, MA, Apr. 2023. USENIX Association.

[77] Y. Zhu, H. Eran, D. Firestone, C. Guo, M. Lipshteyn, Y. Liron, J. Padhye, S. Raindel, M. H. Yahia, and M. Zhang. Congestion control for large-scale rdma deployments. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, SIGCOMM '15, page 523–536, New York, NY, USA, 2015. Association for Computing Machinery.

A Proof of Theorem 1

This section provides the proof of Theorem 1 (Correctness and Uniqueness Guarantee) in Section 4. To prove uniqueness we need to show that Algorithm 1 assigns a time shift value exactly once to every job $j \in J$ in the cluster with Affinity graph $G = (U, V, E)$. To prove the correctness, we need to show that:

$$\forall l, \forall (j_n, j_m) \in \{(j_i, j_k) | (j_i, l) \in E \text{ and } (j_k, l) \in E\} : \\ (t_{j_n} - t_{j_m}) \% p^l = (t^l_{j_n} - t^l_{j_m}) \% p^l \quad (6)$$

where p^l is the perimeter of the geometric abstraction for link l. In other words, to guarantee correctness, we prove that for every pair of jobs sharing a link, the difference in the time-shift values assigned by the algorithm is equal to the relative time-shift given by CASSINI's optimization formulation for that link.

We first use induction to prove that both the above statements are true for any connected and loop-free Affinity graph $G = (U_1, V_1, E_1)$, and later we extend this to a general Affinity graph with many connected sub-graphs.

Base case: First, we show that both statements hold for a graph G with only one link l. Algorithm 1 first selects one of the jobs j_1 connected to the link l and sets $t_{j_1} = 0$. Using its BFS traversal algorithm for all the other jobs j_i connected to l, Algorithm 1 sets the time shift as:

$$t_{j_i} = -t^l_{j_1} + t^l_{j_i}$$

As the algorithm uses BFS and visits each job exactly once, the time-shift value is computed and assigned exactly once for each job. This ensures that for a given job, there is a unique time-shift value computed by the algorithm.

To show correctness, we need to prove equation 6 for all job pairs connected to the link l. Say j_n and j_m are two jobs connected to the link l, then:

$$(t_{j_m} - t_{j_n}) \% p^l = (-t^l_{j_1} + t^l_{j_m} - (-t^l_{j_1} + t^l_{j_n})) \% p^l \\ = (t^l_{j_m} - t^l_{j_n}) \% p^l$$

This shows that the time shift assignments are correct for the base case.

Assumption Step: Let us assume that the two statements hold for every connected and loop-free Affinity graph having n links.

Induction step: We use the above assumption to prove that the two statements hold for a connected and loop-free Affinity graph having $n+1$ links. Let $G = (U_s, V_s, E_s)$ be the connected sub-graph with n links. Now, we create an affinity graph with $n+1$ links, by adding a new link l_n which is already connected to some set of jobs $J = \{j\}$. In order to get a connected and loop-free Affinity graph with $n+1$ links, l_n has to be connected to exactly one job $j_i \in U_s$. It has to be exactly one because having an edge with more than one job from the sub-graph G will form a loop, and not being connected with any of the jobs from the sub-graph G will make the Affinity graph disconnected. Let j_i be the job from subgraph G that is connected to l_n. Since j_i is the only path to reach the link l_n and the jobs J connected to the link, our algorithm 1 will reach link l_n through job j_i. Then, from the algorithm, the time assignments for the jobs in J are given by:

$$\forall j_k \in J, t_{j_k} = t_{j_i} - t^{l_n}_{j_i} + t^{l_n}_{j_k}$$

The uniqueness is guaranteed since BFS visits each job only once. From the assumption step, the correctness constraints for all the links in the subgraph G are assumed to be valid, so we have to only prove equation 6 for the jobs connected to l_n.

$$\forall (j_m, j_n) \in J, (t_{j_m} - t_{j_n}) \% p^l = (t_{j_i} - t^l_{j_i} + t^l_{j_m} - (t_{j_i} - t^l_{j_i} + t^l_{j_n})) \% p^l \\ = (t^l_{j_m} - t^l_{j_n}) \% p^l$$

This shows that both statements hold true for any Affinity graph with $n+1$ links. This concludes the induction proof. Hence, Algorithm 1 holds true for all connected and loop-free Affinity graphs.

Now, we extend to an Affinity graph of a cluster with multiple connected sub-graphs. Since our algorithm solves each connected sub-graph one by one and assigns a single time-shift value for each job in the sub-graph, uniqueness is guaranteed. For correctness, since there is no edge connecting jobs and links from different disjoint sub-graphs there are no constraints across disjoint graphs that need to be checked for correctness. Hence, this concludes the overall proof.

Example. As an example, traversing the affinity graph in Figure 8(b) results in the following unique time-shifts for j_1, j_2, and j_3:

$$t_{j_1} = 0 \ (\textit{reference point}) \quad (7)$$

$$t_{j_2} = (-t^{l_1}_{j_1} + t^{l_1}_{j_2}) \quad \text{mod} \quad \textit{iter\_time}_{j_2} \quad (8)$$

Affinity graph path: $j_1 \rightarrow l_1 \rightarrow j_2$

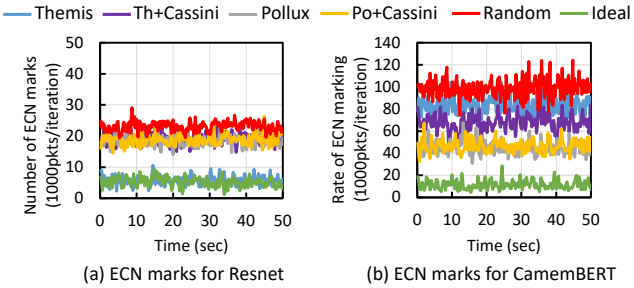
$$t_{j_3} = (-t^{l_1}_{j_1} + t^{l_1}_{j_2} - t^{l_2}_{j_2} + t^{l_2}_{j_3}) \quad \text{mod} \quad \textit{iter\_time}_{j_3} \quad (9)$$

Affinity graph path: $j_1 \rightarrow l_1 \rightarrow j_2 \rightarrow l_2 \rightarrow j_3$

For the correctness of the algorithm, the graph should be loop-free. In CASSINI's design, we eliminate placement configurations that have loops. Themis allocates servers using an auction procedure, which involves multiple jobs in the cluster participating in the auction. This allows multiple possible placement configurations for the jobs participating in the auction. Hence, it is easy to find many loop-free placement configurations among them. Similarly, Pollux reallocates resources periodically, involving multiple jobs and creating many possible placement configurations.

DNN	Memory requirement (MB)	Batch size/GPU	Parallelization strategy	Type
VGG11 [26]	507	512-1800	Data Parallel	Vision
VGG16 [18]	528	512-1800	Data Parallel	Vision
VGG19 [32]	549	512-1800	Data Parallel	Vision
WideResNet101 [72]	243	256-1200	Data Parallel	Vision
ResNet50 [27]	98	256-1800	Data Parallel	Vision
BERT [20]	450	8-32	Data Parallel	Language
RoBERTa [39]	800	8-32	Data Parallel	Language
CamemBERT [43]	266	8-32	Data Parallel	Language
XLM [17]	1116	4-32	Data Parallel	Language
GPT1 [51]	650 - 9000	32-80	Model Parallel	Language
GPT2 [52]	1623- 27000	32-80	Model Parallel	Language
GPT3 [11]	1952-155000	16-48	Model Parallel	Language
DLRM [6]	890 - 1962	16-1024	Model Parallel	Recomm.

Table 3: DNN models used in our experiments.



(a) ECN marks for Resnet (b) ECN marks for CamemBERT

Figure 19: Number of ECN marked packets per iteration

B DNN Models

As mentioned in Section 5.1, we run our experiment with 13 popular DNN models: VGG11 [26], VGG16 [18], VGG19 [32], ResNet50 [27], WideResNet101 [72], BERT [20], RoBERTa [39], XLM [17], CamemBERT [43], GPT1 [51], GPT2 [52], GPT3 [11], and DLRM [6]. Table 3 summarizes the parameters of each model and batch sizes. Note that the batch sizes are provided as a range because the number of workers and hyper-parameters change during scheduling epochs. In particular, in different experiments, we select the batch size according to the hyper-parameters used in prior work [1, 40, 50, 54, 57]. The memory requirement of each model reflects the amount of memory each model occupies in the GPU memory. We adjust the model sizes for different models depending on the parallelization strategy.

C Number of ECN Marked Packets

Figure 19 plots the number of ECN marked packets per iteration for the models ResNet and CamemBERT. These measurements are from the experiment of Section 5.3. The ResNet model has relatively lower ECN marks in general than other models because ResNet has a smaller model size and requires less network bandwidth for its AllReduce phase.