FORESIGHT: Joint Time and Space Scheduling for Efficient Distributed ML Training

Farid Zandi[†] Manya Ghobadi[‡] Yashar Ganjali[†]

[†]University of Toronto, [‡]Massachusetts Institute of Technology

{faridzandi,yganjali}@cs.toronto.edu, ghobadi@csail.mit.edu

***Abstract*—The rapid growth of Machine Learning (ML) workloads has led to increased reliance on large-scale accelerator clusters, where distributed training jobs demand high-performance network communication. However, the independent execution of ML jobs on shared cluster resources results in network contention, degrading training performance. Existing solutions either focus on optimizing communication operations for isolated jobs or address network scheduling in the time and space dimensions separately, leading to suboptimal outcomes.**

In this paper, we introduce FORESIGHT, a system that jointly optimizes communication scheduling across both time (when to communicate) and space (where to route traffic) dimensions. By taking advantage of the predictable and repetitive nature of ML training workloads, we can forecast future network demands and better coordinate communication to reduce congestion. Our approach iteratively refines scheduling decisions based on routing feedback, making the optimization problem tractable, while achieving a contention-free schedule.

Our extensive evaluations demonstrate that FORESIGHT improves network efficiency, causing up to 46% improvement in ML job iteration times, without requiring modifications to existing network hardware or application frameworks. Our findings emphasize the importance of network-aware scheduling and provide a scalable solution for optimizing distributed ML training in shared cluster environments.

I. INTRODUCTION

Setting the Context. Machine Learning (ML) workloads are rapidly evolving, with their size and complexity significantly increasing. As these workloads scale, distributed training jobs increasingly rely on large clusters of machines, demanding intensive communication with strict performance requirements. To efficiently support this growing volume of concurrent ML workloads, accelerator clusters have become commonplace, aiming to simplify resource management and reduce costs. However, variability in job requirements can result in fragmented resource allocations over time. As users frequently compete for the most readily available and cost-effective options, contention increases, leading to overlapping communication events that slow down the involved jobs. Despite recent proposals to mitigate communication bottlenecks through overlapping communication and computation [1], or reducing the size of exchanged data [2]–[4], network performance continues to be a critical bottleneck in distributed ML training workloads [5], [6].

ISBN 978-3-903176-72-0 ©2025 IFIP

State of the Art. The independently submitted ML jobs typically utilize the shared network without coordination with each other, as they lack the mechanisms to do so. The resource management frameworks that focus on ML workloads are not primarily concerned with the network resources, but focus on efficient GPU allocation strategies, such as effectively packing jobs onto available accelerators, reducing resource contention, and ensuring fairness [7]–[9]. They tend to treat network communication as a black-box operation that is not included in the overall scheduling decisions. Previous research aimed at improving communication performance for ML workloads have largely focused on isolated workloads and operations, such as optimizing collective communication events by optimizing for the current topology or scheduling communication phases. These works generally overlook interference between concurrent workloads [10], [11].

Recently, a growing line of research [5], [12], [13] have acknowledged the limitations of isolated optimizations and proposed network-aware, cluster-wide solutions that consider concurrent job executions. These solutions are grounded in two key observations: Observation #1: Fair sharing of network resources is far from optimal in terms of minimizing average flow and job completion times [14], [15]. Effective scheduling of the communication events along the *time dimension*, such as delaying or prioritizing communications strategically [12], [16], can limit the scenarios of network sharing and optimize for completion times. Observation #2: The common link-level load-balancing schemes in large networks result in frequent suboptimal routing choices, as exemplified by the Equal Cost Multi-Path (ECMP) hash collision problem [17], limiting total available network capacity. Recent works proposed improved routing techniques, selecting optimal paths for concurrent flows. These works schedule across the *space dimension* explicitly, maximizing network capacity usage to enhance overall performance [5], [13].

Our Position. We observe that network-aware solutions typically tackle the interconnected dimensions of time scheduling (when to communicate) and space schedul-

ing (where to route network flows) independently. We argue that isolating these two dimensions restricts the full potential of the network scheduling techniques. In fact, decisions about optimal timing schedule depend on routing decisions (which flows share the same path), while efficient routing schedule relies on timing decisions (which flows are transmitting at the same time). Therefore, addressing communication scheduling and routing in a unified manner by jointly optimizing across both time and space dimensions is necessary for achieving optimal network utilization and job performance. We elaborate more on this in sections II and III.

Driven by this, we present FORESIGHT, a system that explicitly integrates scheduling for communication operations at both time and space dimensions simultaneously. We leverage the communication characteristics typical to ML training workloads, such as repetitive optimization steps, collective communication events, and on-off communication patterns, to develop our solution. We argue that the highly predictable behavior allows us to anticipate future network demands and make informed decisions on how the network resources should be utilized.

Our Solution. Optimizing the network scheduling while jointly considering both time and space dimensions is computationally hard [17]. As the job count and their sizes increase, the complexity of scheduling decisions grows exponentially, making exhaustive search infeasible. Interdependence between timing and routing choices further complicate the problem, as adjustments in one dimension affect the other. From a workload perspective, any delay or slowdown in a task alters future execution times for every dependent operation, based on the unique internal structure of every job.

FORESIGHT captures the communication patterns of the jobs, allowing for an initial scheduling strategy in the time dimension. With this schedule in place, we make routing decisions (space dimension), selecting network paths that can fit the demand for the flows. We show this may not produce an optimal schedule, and iteratively adjust timing decisions based on routing feedback to improve scheduling across both time and space. This approach, while making the problem tractable, results in a contention-free schedule for the network resources. We outline the rationale and the details of our algorithm in Sec. III.

This scheduling strategy improves network efficiency and ML job performance without requiring modifications to existing network hardware or applications. Through extensive evaluations, we show that FORESIGHT can drastically reduce network congestion, and improves the job iteration times by up to 46% in our experiments. We study the performance gains in depth in Sec. IV, and conclude the paper in Sec. V.

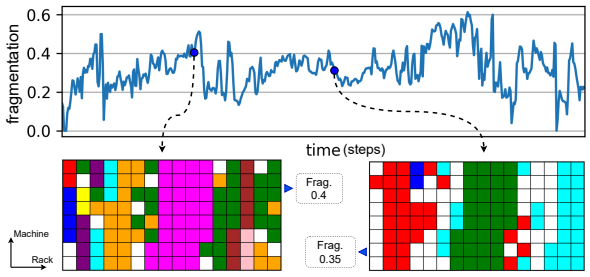

Fig. 1. Simulating job arrivals in a shared cluster. The resources can get fragmented despite attempts to allocate contiguous machines. Color-coded job placement are shown in bottom for 2 sample points.

II. BACKGROUND AND MOTIVATION

Machine Learning Jobs. Large-scale ML training jobs, most notably on Deep Neural Network (DNN) models, necessitate distributed training across numerous machines due to their increasing size, and replication of data is essential to complete training within a reasonable time-frame. To manage this complexity, various approaches have been developed to distribute the training process across multiple machines, such as data, model, and pipeline parallelism [18], resulting in a collection of intertwined computation and communication tasks.

These training processes are characterized by successive optimization steps, with each iteration typically involving the same sequence of computation and communication tasks. The latter usually involves transmitting the parameters, gradients, or activations across a wide range of machines, using Collective Communication (CC) operations such as `all-reduce`, `all-gather`, or `all-to-all`, commonly managed by a library, such as NCCL. This repetitive structure allows for the prediction of future workload behavior. By profiling a single iteration of a job and identifying its network footprint, it becomes possible to create a clear picture of its future operations. This approach can be extended to the set of concurrently running jobs to understand their potential interactions and conflicts, and where interventions are necessary to reduce resource contention.

Resource Fragmentation. Proper allocation of accelerators to jobs significantly impacts communication contention. Ideally, contention-free placement would cleanly separate network resources across jobs, but such placements are hard to achieve in practice. Job resource requirements vary widely, from small to extremely large, and job durations are often dictated by accuracy targets, which are difficult to predict for both job owners and cluster managers. Combined with independent job arrival times, this leads to sub-optimal placement decisions and significant resource fragmentation over time.

To illustrate these challenges, we simulate a cluster of 128 machines grouped in 16 racks. Jobs arrive and remain for durations determined by Pareto distributions. Upon arrival, each job looks for the largest contiguous

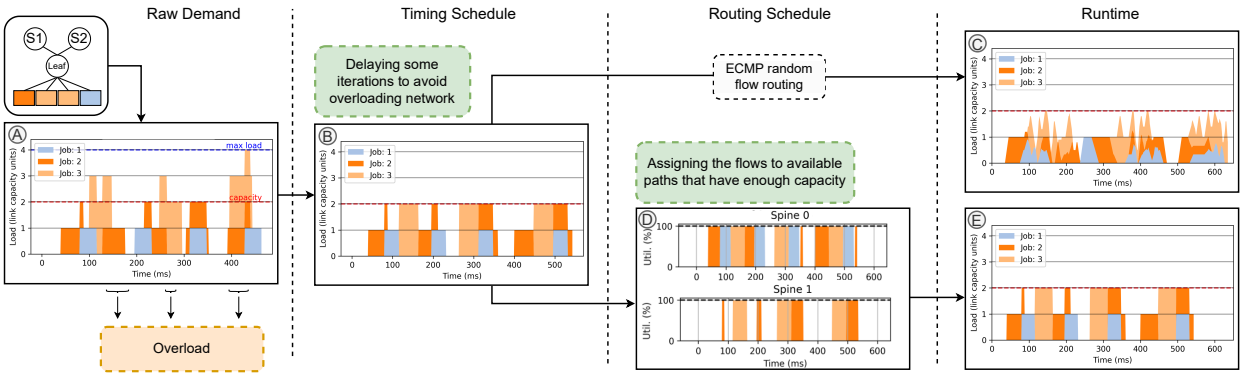

Fig. 2. Overview of the scheduling and routing process, showing how job demand is managed through timing and routing adjustments. Effective scheduling balances network utilization, improving overall runtime performance.

set of machines that fits its machine requirement; if a large-enough contiguous block is not found, more available machines are assigned to the job, albeit in a non-contiguous manner. Assuming ring-based communications among assigned machines, we define the fragmentation level of the machine assignment for the entire cluster as the *ratio of the number of inter-rack flows to total flows*. Fig. 1 shows how the available resources can get fragmented over time, causing fragmentation levels to heavily fluctuate. This often leads to scenarios where a majority of the total communication volume is not contained within racks, thus increasing the potential for contention among flows.

Snapshots of resource allocation in Fig. 1 show scattered resource allocations to the jobs. The available resources (shown in white) are the result of the previous jobs leaving the cluster, leaving "holes" on their departure. When news jobs request for resources, the cluster managers would be eager to allocate as much of the available resources to maximize the utilization of their machines. This perpetuates the issue and keeps the fragmentation high through time.

Interconnecting Networks. Fragmented resources lead to large volumes of inter-rack traffic. To relay this traffic, large-scale cluster interconnects are designed with multiple tiers and numerous paths between racks to provide high bandwidth and ensure reliability [19]. Effective load-balancing (LB) among these paths is therefore crucial for maximizing network usage and delivering maximum performance. ECMP, as an effective and scalable LB approach, has become the standard method used to randomly assign flows to available network paths [6]. While simple to deploy, ECMP suffers from hash collisions, leading to imbalanced load distribution and underutilization of available network resources.

To address the shortcomings of ECMP, other LB schemes have been proposed. Some approaches focus on finer-grained choices using fixed-size cells [20] or flowlets [21], [22]. These solutions often require modifications to network switches, making them challenging to deploy. Other strategies combine load balancing with transport-layer mechanisms, but they remain reactive to network congestion [23], [24]. Packet spraying and its variations [25], aim to eliminate load imbalance but may not be compatible with Remote Direct Memory Access (RDMA) transport protocols commonly used in high-performance clusters due to the out-of-order delivery of the packets [26].

Network Scheduling. Making optimal routing decisions in an online manner based on the locally available information yields sub-optimal results. Works like [17], [27] argue that with a centralized view of network demand, optimal decisions can be made that minimize network contention and improve application-level performance. For example, [27] specifically schedules every single packet in the network to eliminate queues. However, such operations could be computationally expensive and might face scalability issues. Other works draw on application-level information, such as flow sizes or transmissions deadlines [14], [28] and improve metrics like average completion times for the flows.

Based on the understanding that workload-specific characteristics at a global level enable better scheduling, systems like Crux, CASSINI, and MCCS have been designed to address the challenges of distributed Deep Neural Network (DNN) training. Crux [5] tackles inter-job communication contention by introducing the concept of GPU intensity, prioritizing jobs with higher GPU intensity to maximize GPU utilization. CASSINI [12] focuses on the periodic communication patterns of ML workloads, employing a geometric abstraction to find opportunities for interleaving the communication phases of different jobs through time-shifting, thereby minimizing network congestion. Finally, MCCS [13] takes a service-based approach to collective communication in multi-tenant clouds, allowing the cloud provider to centrally route collective communication operations based on network topology and utilization, optimizing performance for tenants.

Time-Space Scheduling. Time or space scheduling in

isolation, as done in the aforementioned works [5], [12], [13] do not yield the optimal results. Fig. 2-A shows an example where three training jobs share a 4-machine rack with half as much capacity to the upper tier. As these jobs periodically transmit their gradients to other machines, their communications occasionally overlap, and their aggregate demand exceeds the available capacity. Attempting to schedule these flows across different paths for optimal transmission would be ineffective, as there are simply not enough paths to accommodate their peak demand. Approaches that rely only on time scheduling, might introduce delays or prioritization, as in Fig. 2-B, ensuring that peak demands of jobs do not coincide. However, if they still depend on the inefficient ECMP routing in runtime, collisions might happen and transmission times can extend beyond what was expected. As a result, the job progress might diverge from the original schedule, leading to even further unintended collisions (Fig. 2-C).

To successfully schedule these workloads across the available resources, one might leverage the fact that demand no longer exceeds capacity after the time scheduling and apply a successful routing strategy that distributes traffic across the two available paths, as seen in (Fig. 2-D). This approach ensures that the runtime behavior remains on track with the schedule, allowing the jobs to complete in 550 ms instead of approximately 620 ms in the ECMP scenario (Fig. 2-E). While this example outlines the importance of time-space scheduling, in Sec. III we study where this approach doesn't yield the optimal results and describe how we attempt to fix the issue.

III. Design

Problem Statement. We define the problem as multiple iterative training jobs running concurrently across an interconnected cluster of machines. The strategy used to allocate machines to jobs lies outside the scope of our work, though we assume that the placement could be far from optimal. We assume that a single iteration of each job has been profiled at the flow level in isolation from the other jobs, showing the sending rate through time for each flow. However, we do not assume knowledge of dependencies between the flows. Our goal is to create a contention-free network schedule for a given period of time, such that the average iteration time across all jobs is minimized. Specifically, for each iteration, we determine 1) when it should start, *i.e.*, if any extra delay should be added before the iteration, and 2) the network path each flows should take. The length of the schedule is at most the Largest Common Multiplier (LCM) of the iteration lengths, but smaller numbers could be chosen if the LCM is too large.

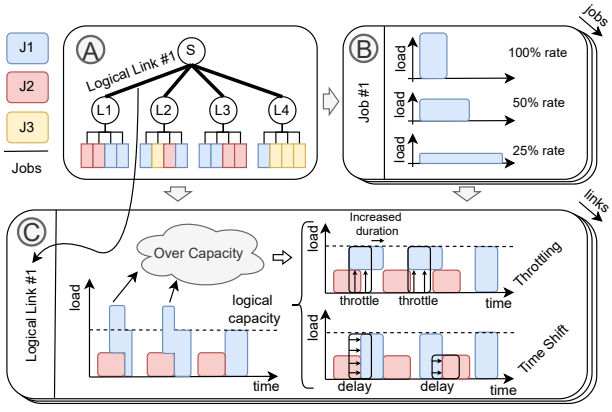

Fig. 3. Managing network congestion by throttling and time-shifting in a leaf-spine topology. When jobs exceed link capacity, throttling reduces flow rates, extending duration, while time-shifting delays execution to balance resource usage.

Simplifying the Network. Solving timing and routing simultaneously is challenging because each influences the other directly: routing decisions rely on knowing which flows occur simultaneously (dictated by timing), while timing decisions depend on knowing which flows share the same network paths (determined by routing). To overcome this cyclical dependency, we approach the problem in stages. Initially, we simplify the routing aspect by abstracting the network topology. For example, in a leaf-spine topology, we consolidate all spines into a logical one with the aggregated capacity. This simplification allows us to come up with a rough estimate of the timing decisions, ensuring the demands don't exceed the network capacity at a logical level. Then, based on these timing decisions, we can assign flows to network paths accordingly, such that the link-level capacities are not exceeded.

Beyond Simple Delays. while making the timing decisions, we represent the network as a set of logical links. In a leaf-spine topology, each leaf switch has upward and downward logical links connecting it to the logical spine, while directly connecting to multiple machines within its rack. Each job with machines in a rack sends network traffic over the logical link connecting that rack to the rest of the cluster, unless the job is fully confined to a single rack. An example scenario can be seen in Fig. 3-A.

We aim to schedule network traffic so that no logical link exceeds its capacity at any point. Previous work that focuses on timing schedules [12], suggested that applying a single initial delay to each job is sufficient to prevent overlapping traffic. However, this might not be true in many cases, especially when the communication-to-computation ratios are large, and job iteration times are not similar. Furthermore, shifting one job's iteration affects traffic across multiple logical links simultaneously, leading to scenarios that singular delays are not enough. To address this, we argue that delays should be applied more frequently, potentially before every iteration, to effectively schedule the iterations.

FORESIGHT schedules iterations one at a time, as illustrated in Fig. 3-C. At each step, we select the job that has received the least service so far, ensuring fairness across all jobs. We then calculate the minimum delay required for the current iteration to fit within the capacity constraints of all involved logical links, repeating this process until all iterations have been scheduled. Although this method guarantees a schedule without capacity violations, delaying iterations isn't always the best choice, particularly when partial network capacity remains unused. In such scenarios, we explore throttling as an alternative, which limits the transmission rates of flows at the hosts. This spreads an iteration's network load over a longer duration, while lowering the peak network demand.

We therefore profile each job at different throttling rates, as shown in Fig. 3-B. During scheduling, we select the rate that minimizes each iteration's completion time. Note that throttling isn't necessarily the same as a uniformly stretching the workload in time, as internal job dependencies influence whether specific communication events lie on the critical path, impacting dependent tasks differently.

Assigning Flow Paths. Once the timing schedule has been established based on logical network capacity, the next step is to assign each flow to a specific network path. In a two-tier leaf-spine topology, this involves selecting a spine switch to route traffic between the source and destination leaf switches. Because each routing decision affects multiple links (upward and downward links), a greedy assignment of flows to spines typically does not yield optimal results. Instead, we model the leaf switches as nodes in a bipartite multigraph, where each flow corresponds to an edge connecting the source leaf on the left side to the destination leaf on the right. We then apply edge coloring to this graph, ensuring no two edges connected to the same node share the same color.

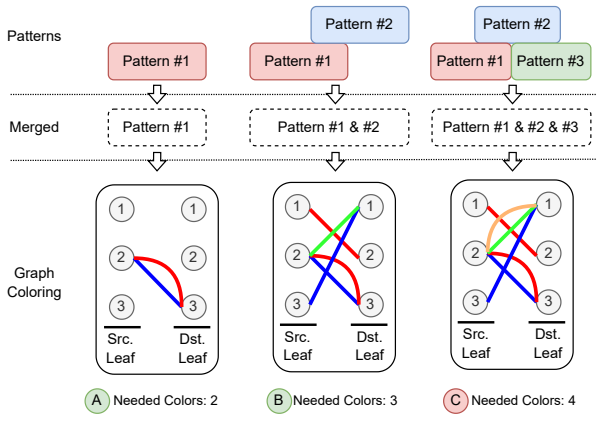

Fig. 4. Increasing complexity in graph coloring due to pattern merging. As more patterns overlap, additional colors are required to prevent conflicts in routing.

Each color assigned to an edge identifies the chosen spine switch for the corresponding flow, as been shown in [27].

The flows routed together are those transmitting simultaneously. Due to the repetitive and collective nature of these communication events, we observe recurring patterns. For instance, in a ring-all-reduce operation, each node transmits data chunks to the next machine in the ring over multiple rounds, repeating this process for each layer in every iteration. When patterns of different jobs overlap, their routing decisions must be made together. To achieve this, we identify the patterns and merge them when overlaps occur. This merging can progress iteratively, as shown in Fig. 4, growing into larger and larger merged regions. Different combinations of merged patterns define the distinct coloring problems we must solve. Efficient coloring algorithms exist to solve the edge coloring with $O(E \log V)$ complexity, where E is the number of flows and V is the number of leaf switches [29].

Algorithm 1: Time-Space Scheduling

```
1  function SolveTiming(jobs, bad_ranges):
2      while jobs remains to be scheduled do
3          (job, iter_id) = Least Attained Service job
4
5          if iteration span ∈ bad_ranges then
6              if first encounter then
7                  # inflate communication times
8              else
9                  # reduce available capacity
10
11         foreach rate ∈ profiled rates do
12             delay_rate = least added delay such that
                            demand fits on all used links
13             finish_rate = start + length_rate + delay_rate
14
15         selected_rate = arg min_rate(finish_rate)
16         selected_delay = delay_selected_rate
17
18         timing_job, iter_id = (selected_rate, selected_delay)
19     return timing
20
21 function SolveRouting(jobs, timing):
22     patterns = GetPatterns()
23     merged_ranges = MergeRanges(patterns)
24     bad_ranges = empty
25
26     foreach range in merged_ranges do
27         combined_flows = flows of overlapping patterns
28         graph = CreateGraph(combined_flows)
29         coloring = ColorGraph(graph, num_spines)
30         routing_flow = coloring(flow)
31
32         if needed colors > spine count then
33             bad_ranges.append(merged_range)
34
35     return routing, bad_ranges
36
37 function Schedule(jobs):
38     bad_ranges = empty
39     repeat
40         timing = SolveTiming(jobs, bad_ranges)
41         routing, bad_ranges = SolveRouting(jobs,
             timing)
42     until bad_ranges is empty
43
44     return timing, routing
```

Fixing the Routing Issues. Expanding merged patterns can lead to scenarios where initially non-overlapping patterns are combined due to shared timing with other patterns, as seen for patterns #1 and #3 in Fig. 4-C. This issue primarily arises because flows in pattern #2 cannot change their paths mid-transfer. As more patterns overlap, additional colors are needed for a proper edge-coloring. If the number of required colors exceeds available spines, some flows inevitably collide. To address this, we first identify problematic ranges where edge coloring fails. These problematic ranges are then returned to the timing solver, which introduces additional delays to reduce complexity in these ranges by: a) inserting artificial delays between jobs to break overlap chains among patterns, and b) reducing logical capacity, effectively treating the network as having fewer spines to minimize pattern overlaps, if the problem persists.

Alg. 1 shows how this process is iteratively applied to fix all routing issues. At every step, timing decisions are made based on the bad ranges identified by the routing. The routing decisions are then made based on the new timings, identifying new problematic ranges. The back-and-forth between the two components continues until routing can be successfully done for all flows.

Handling Flow Granularity. The edge-coloring output assigns the whole link to a flow, regardless of whether the flow is going to use the entire capacity. This leads to inefficient routing attempts, especially when throttled job profiles are used. Additionally, the rigid allocation of entire flows to links can create difficulties in path assignment. For example, if 3 flows each consume $\frac{2}{3}$ of a link's capacity and must be distributed across two links, one link will become overloaded while the other remains underutilized due to the indivisibility of flows. To address this issue, we break flows into subflows to provide greater routing flexibility when needed. The subflows can then be distributed across different paths, making the routing problem easier to solve. To solve edge-coloring with subflows, each flow might be presented by more than a single edge depending on its rate. Each spine will then also be represented by more than one color. Once the coloring is complete, subflows are consolidated back into larger flows if they use the same spine.

In Sec. IV, we show how these components build on top of another, to create a contention-free schedule for the flows, as we study the performance gains of our solution based on the mentioned design decisions.

IV. Evaluations

Simulation Environment. To evaluate Foresight, we developed a network simulator in C++ [30] in ∼7K lines of code, designed to execute concurrent workloads on a shared cluster. Each workload is defined by a Directed Acyclic Graph (DAG) of communication and computation tasks and their dependencies. The simulator operates at a flow level, but progresses in discrete timesteps. At every step, the simulator calculates the sending rate of active flows (the rate share on the bottleneck link), continuing until the flows have completed their transmission. Meanwhile, the end-hosts perform the computation tasks that unblocks the subsequent transmissions.

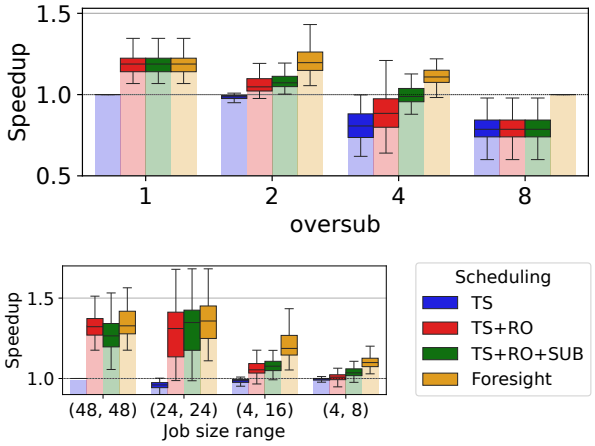

Fig. 5. Our solution outperforms the other baselines in improving the average iteration time of the jobs.

We simulate leaf-spine topologies, with varying rack size and switch count, as specified in each experiment. We run various jobs for a number of intervals that fits within a given period. Each job is allocated to a number of machines and simulates a data-parallel training process, which involves all layers performing a forward pass followed by a backward pass. Once the backward pass for any layer finishes, the machines perform a ring-all-reduce operation to synchronize the corresponding gradients. When all layers are synchronized, the training moves on to the next iteration.

Experiment Setting. The baseline configuration in our experiments employs ECMP, such that each cross-leaf flow selects a random spine switch, and flows competing for the same link, share bandwidth according to a max-min fairness model. All jobs begin execution in parallel in the baseline configuration. In the simulation, we monitor the length of each training iteration in each job. The primary performance metric is the overall average iteration time across all jobs, with lower values indicating higher utilization of the network and accelerator resources, hence faster completion of the jobs.

In the following sections we explore 1) the effectiveness of different scheduling approaches in improving the performance of the training jobs, 2) the impact of environment parameters on the gains that our schedule can bring about, and 3) in what ways are the jobs affected by the scheduling decisions.

1) Impact of Time-Space Scheduling. In our first experiment, we study how our solution behaves in

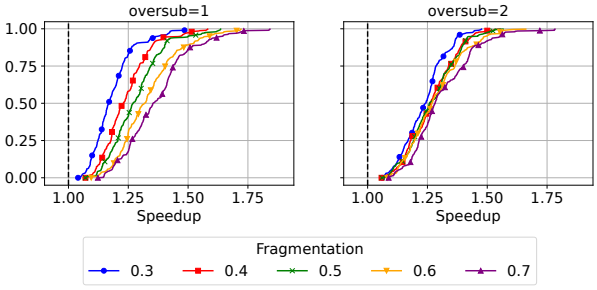

Fig. 6. CDF of the performance gain of FORESIGHT across different job placements.

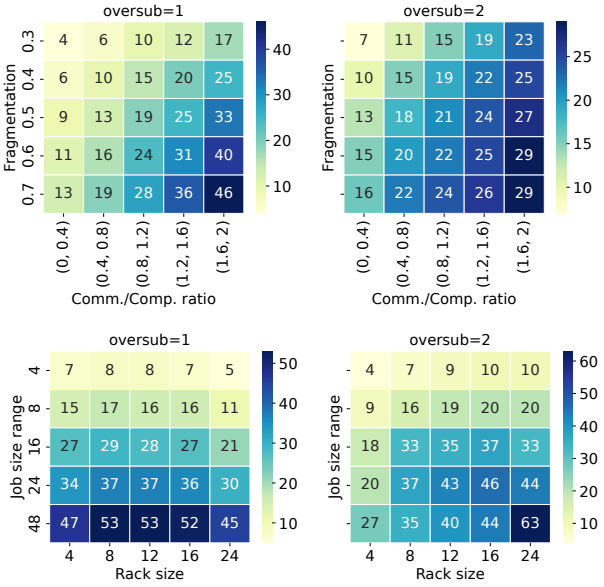

Fig. 7. Gains of our solution across varying fragmentation levels, communication intensity, job sizes, and rack sizes. Higher fragmentation, increased communication, larger jobs, and appropriately sized racks contribute to higher gains.

different situations and how it manages to outperform the other baselines:

- Timing Schedule (TS): Introduces delays between iterations to avoid overloading network capacity.
- Routing (RO): Assigns spine switch to each flow based on TS, calculated with edge-coloring.
- Subflows (SUB): Splits flows into lower-rate subflows across multiple spines to improving routing.
- FORESIGHT: Iteratively detects and corrects routing failures by changing the timing decisions.

We simulate 48 machines, grouped in 8-machine racks. The size of each job is chosen between 4 and 16, and the placement has an fragmentation level of 0.5. Number of spines changes from 1 to 8, to show different oversubscription ratios. Fig. 5 shows the effectiveness of our solution varies based on the available network resources.

When there is no oversubscription, the timing schedule (TS) has no impact, as the network capacity is always sufficient to accommodate workloads without introducing delays. However, as multiple paths exist, routing flows along different paths (RO) significantly improves the progress of the jobs. While naive routing can be effective in low-congestion scenarios, it fails to capture potential improvements in more complex cases. The addition of subflows improves routing flexibility, but without a mechanism to actively fix the routing issues, it still fails to achieve much gains. FORESIGHT, identifies and corrects unsuccessful routing-timing combinations, leading to substantial performance gains. At moderate oversubscription levels (1 and 2), we observe an average improvement of 19% and 21% in iteration times, respectively. Under extreme oversubscription (8), we observe that creating a valid schedule is impossible, and we fall back to the baseline network operation. Note that timing scheduling on its own is usually counterproductive, as it fails to consider the uncertainties that arise from runtime-based routing decisions.

The impact of our approach is also influenced by job size and workload diversity. In cases where a single job spans multiple racks, better routing decisions improve job performance by 34% on average. In such large-scale jobs, the collective communication pipeline is affected by many cross-rack flows, and a slowdown in any of those transmissions can potentially slow down all subsequent operations. When a mix of two jobs is running, the timing schedule also helps with reducing contention, leading to speedup improvements of up to 36% on average. In workloads dominated by many small jobs, a significant portion of the communication remains within rack boundaries, making these jobs less affected by routing conflicts. Even in these cases, our approach provides consistent, but more modest, improvements by ensuring efficient utilization of available network resources. Overall, our solution shows strong adaptability across different network conditions and job compositions, consistently enhancing iteration times and minimizing contention.

2) Impact of Environment Parameters. Job placement significantly impacts the effectiveness of our solution, as workloads that naturally avoid conflicts leave little room for scheduling improvements. To quantify this, we defined fragmentation in Sec. II, as the ratio of inter-rack flows to all flows. We analyze multiple placement scenarios with varying fragmentation levels while keeping the number of machines and network topology constant.

Fig. 6 shows the CDF of speedups for 400 different placements. We observe that low fragmentation results in less cross-rack communication, limiting potential gains, whereas high fragmentation increases routing conflicts, creating more opportunities for scheduling to improve performance. It is also worth noting that our

TABLE I
COMPARISON OF JOB PERFORMANCE ACROSS SCHEDULING APPROACHES. OPTIMIZING TIMING AND ROUTING REDUCES SLOWDOWNS AND IMPROVES FAIRNESS (TIMES ARE IN MS)

	Job ID	1	2	3	4	5
(isolated)	iter. time	468	1064	786	686	890
ECMP	Avg iter.	697	1310	1014	950	951
	slowdown	1.49	1.23	1.29	1.38	1.07
Ours	Avg. iter.	492	1086	809	707	902
	Avg. delay	22.1	21.6	21.4	19.1	11.31
	slowdown	1.05	1.02	1.03	1.03	1.01
Perfect	Avg. iter.	473	1071	791	689	895
	slowdown	1.01	1.01	1.01	1.0	1.0

solution is robust enough to never cause negative impact on the execution metrics of the jobs.

Another key factor in determining the potential conflicts within the cluster is the ratio of communication to computation time, as larger and more frequent communication events place a greater load on the network, emphasizing its influence on the overall progress rate of jobs. Recent studies have reported that up to 85% of training time in models such as VGG can be spent on communication [18]. The results in Fig. 7 show the gains of our solution as we iterate across different fragmentation levels and communication-to-computation ratios. As fragmentation increases, we observe a clear upward trend in the gains, indicating that more scattered job placements lead to improved gains, reaching up to 46% under the highest intensity. Similarly, higher communication intensity positively correlates with greater gains.

We further study how job and rack sizes influence the gains of our solution under different oversubscription levels in Fig. 7. Across both heatmaps, we observe that larger job sizes generally lead to higher gains, suggesting that our approach is particularly beneficial for workloads with greater resource demands. Additionally, increasing the rack size tends to yield higher gains, as larger racks with at the similar oversubscription ratio means larger number of routing options, pushing up the gains of space scheduling.

3) Effect on Jobs. Table I shows an example comparison of iteration times across different scheduling approaches for five jobs running on 48-machine. The second rows shows the iteration time for the jobs if running in isolation and experiencing no contention with the other jobs. The baseline approach results in significant and unfair slowdowns due to inefficient decisions in network resource utilization. The slowdown values range from 1.07 to 1.49, with some jobs being disproportionately affected. This imbalance occurs as contention in the network affects jobs differently depending on their placement and demand patterns, leading to such a wide range of negative impacts.

In contrast, our solution, which optimizes both timing and routing, significantly reduces iteration times compared with the baseline approach as we virtually eliminate network congestion. The iterations times are only minimally increased compared with isolated execution, primarily due to the minor timing adjustments in the form of added delays, ranging from 11 to 22 ms of average delay per iteration. The negative impact on the jobs is fairly similar, compared with the wide range seen in the baseline method, as we employ the least-service-attained heuristic in the time scheduling. We use this example to indicate that a coordinated time-space scheduling is a practical approach for mitigating congestion, improving performance, and ensuring fair resource allocation across jobs, with performance close to that of a perfect routing and rate-sharing scheme.

V. DISCUSSION AND CONCLUSION

Deployment Challenges. To deploy FORESIGHT in a training cluster, it is essential to support adding delays to training iterations and throttling the flow rates. This is managed by an agent running on the servers, added by the cluster managers, to apply time shifts as discussed in [12]. The same agent can also enforce flow throttling to regulate network usage, as well as the routing decisions specified within the space schedule. Prior research [5] suggested that this can be achieved by strategically selecting transport-layer ports to influence ECMP decisions. Furthermore, source-routing techniques can be employed by the agents to drive routing without requiring modifications to switch architecture. Finally, the complexities of breaking flows into smaller subflows has been studied in prior works [31].

Offline Scheduling. Machine learning workloads are generally stable and repetitive, which makes offline scheduling a viable option for them in many cases. While offline scheduling might be simpler to approach, it can struggle to adapt to unexpected changes, background processes, or device failures. On the other hand, online scheduling offers greater flexibility in handling such uncertainties but often requires more complex system changes, or run the risk of causing latencies on the critical path of execution. Leveraging the predictability of machine learning workloads presents an opportunity to strike a balance between these two approaches.

Flow-level Simulation. Our flow-level simulator might not capture some of the nuances of a network's operation, such as queue occupancy levels, PFC signals, or the immediate impact of congestion notifications on transmission rates. We argue that since our proposed schedule creates a contention-free environment for the flows, we are minimally affected by these. On the other hand, the methods we compare against are more prone to having their performance negatively affected by these details, which only enhances our comparative advantage.

Conclusions. In this paper, we presented FORESIGHT, a unified scheduling system that jointly optimizes com-

munication timing and routing for distributed ML workloads. By leveraging the predictable structure of ML training jobs, we minimize network contention and improve resource utilization. Our evaluations show that FORESIGHT reduces network congestion and cuts down iteration times by up to 46%. This approach offers a practical and scalable solution for improving ML training efficiency in shared cluster environments.

REFERENCES

[1] 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*, 2019, pp. 16–29.

[2] J. Wangni, J. Wang, J. Liu, and T. Zhang, "Gradient sparsification for communication-efficient distributed optimization," *Advances in Neural Information Processing Systems*, vol. 31, 2018.

[3] D. Alistarh, D. Grubic, J. Li, R. Tomioka, and M. Vojnovic, "Qsgd: Communication-efficient sgd via gradient quantization and encoding," *Advances in neural information processing systems*, vol. 30, 2017.

[4] Y. Lin, S. Han, H. Mao, Y. Wang, and W. J. Dally, "Deep gradient compression: Reducing the communication bandwidth for distributed training," *arXiv preprint arXiv:1712.01887*, 2017.

[5] J. Cao, Y. Guan, K. Qian, J. Gao, W. Xiao, J. Dong, B. Fu, D. Cai, and E. Zhai, "Crux: Gpu-efficient communication scheduling for deep learning training," in *Proceedings of the ACM SIGCOMM 2024 Conference*, 2024, pp. 1–15.

[6] A. Gangidi, R. Miao, S. Zheng, S. J. Bondu, G. Goes, H. Morsy, R. Puri, M. Riftadi, A. J. Shetty, J. Yang *et al.*, "Rdma over ethernet for distributed training at meta scale," in *Proceedings of the ACM SIGCOMM 2024 Conference*, 2024, pp. 57–70.

[7] 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)*, 2020, pp. 289–304.

[8] W. Xiao, R. Bhardwaj, R. Ramjee, M. Sivathanu, N. Kwatra, Z. Han, P. Patel, X. Peng, H. Zhao, Q. Zhang *et al.*, "Gandiva: Introspective cluster scheduling for deep learning," in *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, 2018, pp. 595–610.

[9] 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)*, 2021.

[10] G. Wang, S. Venkataraman, A. Phanishayee, N. Devanur, J. Thelin, and I. Stoica, "Blink: Fast and generic collectives for distributed ml," *Proceedings of Machine Learning and Systems*, vol. 2, pp. 172–186, 2020.

[11] 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)*, 2023, pp. 593–612.

[12] S. Rajasekaran, M. Ghobadi, and A. Akella, "CASSINI:Network-Aware job scheduling in machine learning clusters," in *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*, 2024, pp. 1403–1420.

[13] Y. Wu, Y. Xu, J. Chen, Z. Wang, Y. Zhang, M. Lentz, and D. Zhuo, "Mccs: A service-based approach to collective communication for multi-tenant cloud," in *Proceedings of the ACM SIGCOMM 2024 Conference*, 2024, pp. 679–690.

[14] C.-Y. Hong, M. Caesar, and P. B. Godfrey, "Finishing flows quickly with preemptive scheduling," *ACM SIGCOMM Computer Communication Review*, vol. 42, no. 4, pp. 127–138, 2012.

[15] 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*, 2022, pp. 235–242.

[16] S. Rajasekaran, S. Narang, A. A. Zabreyko, and M. Ghobadi, "Mltcp: A distributed technique to approximate centralized flow scheduling for machine learning," in *Proceedings of the 23rd ACM Workshop on Hot Topics in Networks*, 2024, pp. 167–176.

[17] M. Al-Fares, S. Radhakrishnan, B. Raghavan, N. Huang, A. Vahdat *et al.*, "Hedera: dynamic flow scheduling for data center networks." in *Nsdi*, vol. 10, no. 8. San Jose, USA, 2010, pp. 89–92.

[18] A. Harlap, D. Narayanan, A. Phanishayee, V. Seshadri, N. Devanur, G. Ganger, and P. Gibbons, "Pipedream: Fast and efficient pipeline parallel dnn training," *arXiv preprint arXiv:1806.03377*, 2018.

[19] M. Al-Fares, A. Loukissas, and A. Vahdat, "A scalable, commodity data center network architecture," *ACM SIGCOMM computer communication review*, vol. 38, no. 4, pp. 63–74, 2008.

[20] K. He, E. Rozner, K. Agarwal, W. Felter, J. Carter, and A. Akella, "Presto: Edge-based load balancing for fast datacenter networks," *ACM SIGCOMM Computer Communication Review*, vol. 45, no. 4, pp. 465–478, 2015.

[21] M. Alizadeh, T. Edsall, S. Dharmapurikar, R. Vaidyanathan, K. Chu, A. Fingerhut, V. T. Lam, F. Matus, R. Pan, N. Yadav *et al.*, "Conga: Distributed congestion-aware load balancing for datacenters," in *Proceedings of the 2014 ACM conference on SIGCOMM*, 2014, pp. 503–514.

[22] E. Vanini, R. Pan, M. Alizadeh, P. Taheri, and T. Edsall, "Let it flow: Resilient asymmetric load balancing with flowlet switching," in *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, 2017, pp. 407–420.

[23] M. A. Qureshi, Y. Cheng, Q. Yin, Q. Fu, G. Kumar, M. Moshref, J. Yan, V. Jacobson, D. Wetherall, and A. Kabbani, "Plb: congestion signals are simple and effective for network load balancing," in *Proceedings of the ACM SIGCOMM 2022 Conference*, 2022, pp. 207–218.

[24] T. Bonato, A. Kabbani, D. De Sensi, R. Pan, Y. Le, C. Raiciu, M. Handley, T. Schneider, N. Blach, A. Ghalayini *et al.*, "Smarttreps: Sender-based marked rapidly-adapting trimmed & timed transport with recycled entropies," *arXiv e-prints*, pp. arXiv–2404, 2024.

[25] S. Ghorbani, Z. Yang, P. B. Godfrey, Y. Ganjali, and A. Firoozshahian, "Drill: Micro load balancing for low-latency data center networks," in *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, 2017, pp. 225–238.

[26] C. H. Song, X. Z. Khooi, R. Joshi, I. Choi, J. Li, and M. C. Chan, "Network load balancing with in-network reordering support for rdma," in *Proceedings of the ACM SIGCOMM 2023 Conference*, 2023, pp. 816–831.

[27] J. Perry, A. Ousterhout, H. Balakrishnan, D. Shah, and H. Fugal, "Fastpass: A centralized" zero-queue" datacenter network," in *Proceedings of the 2014 ACM conference on SIGCOMM*, 2014, pp. 307–318.

[28] M. Alizadeh, S. Yang, M. Sharif, S. Katti, N. McKeown, B. Prabhakar, and S. Shenker, "pfabric: Minimal near-optimal datacenter transport," *ACM SIGCOMM Computer Communication Review*, vol. 43, no. 4, pp. 435–446, 2013.

[29] R. Cole and J. Hopcroft, "On edge coloring bipartite graphs," *SIAM Journal on Computing*, vol. 11, no. 3, pp. 540–546, 1982.

[30] F. Zandi, "Psim. network simulator for dag-based protocols on custom networks." https://github.com/FaridZandi/psim/, 2025.

[31] Y. Lu, G. Chen, B. Li, K. Tan, Y. Xiong, P. Cheng, J. Zhang, E. Chen, and T. Moscibroda, "{Multi-Path} transport for {RDMA} in datacenters," in *15th USENIX symposium on networked systems design and implementation (NSDI 18)*, 2018, pp. 357–371.