MINA: Fine-grained In-network Aggregation Resource Scheduling for Machine Learning Service

Shichen Dong[1], Zhixiong Niu[2*], Mingchao Zhang[1], Zhiying Xu[1], Chuntao Hu[1], Pengzhi Zhu[3], Qingchun Song[3], Lei Qu[2], Peng Cheng[2], Cam-Tu Nguyen[1], Shaoling Sun[4], Xiaohu Xu[4], Yongqiang Xiong[2], Wei Wang[1], Xiaoliang Wang[1*]

[1]Nanjing University, [2]Microsoft Research, [3]NVIDIA, [4]China Mobile

IEEE INFOCOM 2025 - IEEE Conference on Computer Communications | 979-8-3315-4305-1/25/$31.00 ©2025 IEEE | DOI: 10.1109/INFOCOM55648.2025.11044657

***Abstract*—In-network aggregation (INA) offloads gradient aggregation onto switches, and thus effectively reduces the aggregation latency and the volume of traffic. However, INA resources are limited due to the high cost of on-chip memory, which imposes distinct challenges to the effective scheduling of these resources in multi-job MLaaS scenarios. In this paper, we explore the scheduling of INA resources in spatial and temporal dimensions, specifically focusing on its impact on the average job completion time (JCT) and the efficiency of INA resources. We propose MINA, an innovative co-design of algorithm and system that intelligently assigns INA resources to each job and effectively schedules these resources among multiple jobs. Our experiments show that MINA attains an INA efficiency score of 0.9998, implying that almost all jobs run nearly as efficiently as they would with exclusive INA acceleration.**

***Index Terms*—In-network Aggregation, Machine Learning as a Service, Data Center Networking**

I. INTRODUCTION

The training of the increasingly large Machine Learning (ML) models necessitates distributed computing approaches, often spanning multi-rack clusters [1], [2]. The concept of Machine Learning as a Service (MLaaS) [3] has gained traction, primarily for its cost-efficiency in handling large-scale ML model training. However, a substantial challenge in these distributed training environments is the escalated communication overhead. This issue frequently emerges as the predominant factor in prolonging training time [4], [5], [6], [2], underscoring the need for efficient communication strategies.

The technology of in-network aggregation (INA) has thus been applied to speed up distributed ML training [7], [8], which realizes collective operations in switches to reduce latency and bandwidth consumption. Several prototypes (*e.g.*, SwitchML [7] and ATP [8]) have verified that INA can effectively accelerate communication-intensive ML training jobs such as VGG16 [9] using programmable switches over Ethernet. Over InfiniBand networks, Scalable Hierarchical Aggregation and Reduction Protocol (SHARP) [10], [11] has been used to speed up ML data centers through commodity switches that have been widely deployed in large cloud providers [12] and TOP500 data centers [13]. All these INA implementations achieve large performance enhancements. For example, SHARP improves the all-reduce bandwidth by a factor of 4.8 at 256MB message size on 64 hosts [11].

Unfortunately, while INA provides significant performance boosts to ML jobs, it also imposes substantial constraints and challenges on the allocation and scheduling of INA resources. For instance, Quantum HDR switches with SHARP technology only support one active INA job per switch at the same time [14] in real production environments. The next generation Quantum-2 NDR switches support one active INA job per port. The existing job placement and resource scheduling solutions in data centers fail to consider the unique characteristics of INA resources [15], [16]. This renders INA acceleration inapplicable in MLaaS scenarios.

In this paper, we begin by analyzing the distinctive temporal and spatial characteristics of INA-enabled ML jobs. This analysis leads to two key insights: 1) the same amount of JCT improvements can be achieved by selectively applying INA acceleration for only parts of the collective operations of a job; 2) the probability of INA resource conflicts correlates closely with the amount of host fragments, implying that conflict mitigation is achievable through reducing these fragments. Building on these insights, we introduce MINA, a system for fine-grained INA resource scheduling in MLaaS scenarios, aiming to optimize the utilization of INA resources and enhance the average job completion time (JCT). To achieve this, MINA needs to answer the following two questions.

First, *how to efficiently allocate the limited INA resources across multiple jobs and schedule them to enhance the utilization of INA resources and thereby improve the average JCT?* MINA systematically tackles this issue by breaking it down into three steps: job placement, aggregation tree building, and INA resource sharing, each step being resolved by a dedicated algorithm tailored to its specific targets and requirements. The job placement algorithm determines the optimal hosts for each job to be allocated by minimizing host fragments. The tree building algorithm constructs the aggregation trees and resolves the INA resource conflicts to maximize the number of jobs that can use INA acceleration. The INA resource sharing algorithm regulates the scheduling of INA resources among multiple jobs through time-division multiplexing.

Second, *how to design an MLaaS system that supports the above algorithms with minimal overhead, while still maintaining compatibility with existing ML applications?* Furthermore, time-division multiplexing sharing of INA resources necessitates sophisticated coordination and efficient synchronization

*Zhixiong Niu (zhniu@microsoft.com) and Xiaoliang Wang (waxili@nju.edu.cn) are corresponding authors.

The project is partially supported by *National Natural Science Foundation of China* under Grant Numbers 62172204, 62325205, 62072228, 62272213, *Collaborative Innovation Center of Novel Software Technology and Industrialization*, and *Nanjing University-China Mobile Communications Group Co.,Ltd. Joint Institute*.

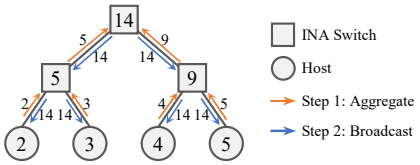

Fig. 1. Process of in-network aggregation

across hosts, especially in high-speed networks. MINA addresses this challenge through the co-design of an allocation controller, a sharing controller, and a network-layer runtime library. The allocation controller facilitates the seamless migration of aggregation trees without interfering with the ongoing execution of jobs. The sharing controller and the runtime library support efficient host coordination using a carefully designed distributed consensus protocol.

We prototype MINA by utilizing SHARP as the INA implementation and deploy the system on an InfiniBand-based cluster. We evaluate MINA's scalability via large-scale simulations using real traces. Evaluation results demonstrate that MINA achieves an INA efficiency score of up to 0.9998, indicating that almost all jobs are as fast as if they were using INA acceleration exclusively, even in scenarios where INA resources are limited. Moreover, MINA exhibits a high degree of robustness to different cluster configurations and imposes only minimal overheads. Our contributions are as follows:

- We propose a fine-grained model that captures the characteristics of communication patterns in distributed ML training. We analyze and reveal the importance of reducing host fragments in improving average JCT and training throughput.
- We make INA acceleration applicable to multi-job scenarios in MLaaS clouds by 1) carefully allocating hosts to jobs and building aggregation trees for each job, and 2) enabling time-division multiplexing sharing of INA resources among multiple jobs.
- We design and implement a prototype system, MINA, for supporting INA resource allocation, scheduling, and sharing in practice. Evaluations show MINA effectively improves the average JCT and INA resource utilization in various cluster configurations. The evaluations are reproducible, and the code has been open-sourced at github.com/ClubieDong/MinaSimulator.

II. FINE-GRAINED INA RESOURCE SCHEDULING

A. Current Practices in MLaaS

Figure 1 illustrates a typical hierarchical aggregation process of INA. Switches serving the same job form an aggregation tree. The gradients from children nodes are aggregated on their parent switches and then forwarded to higher level switches. The root switch of the aggregation tree will accumulate the aggregation result and broadcast the result back to all workers.

INA has been demonstrated to significantly accelerate distributed ML training jobs [8], [7], thereby enhancing the training request throughput and speeding up MLaaS data centers. However, INA resources are constrained owing to the high cost of on-chip memory of programmable switches, which poses substantial challenges in the efficient allocation and utilization of the limited INA resources [10], [11]. The prevailing job allocation and scheduling policies within MLaaS data centers do not account for the distinctive characteristics of INA resources and ML jobs, leading to the suboptimal performance.

a) Unaware of aggregation tree topology: Prevailing job managers in data centers like Slurm [16] focus on satisfying the required amount of resources (*e.g.*, CPU, memory, network bandwidth) for each job. They treat each server as an individual without the awareness of network topology. However, each INA-enabled job occupies not only host resources but also INA resources within switches on its aggregation tree. Careless job placement can easily lead to INA resource conflicts, making INA unavailable for most jobs.

b) Unaware of ML job communication patterns: Currently, resource allocation paradigms within data centers (*e.g.*, Tiresias [15]) allocate resources to jobs exclusively throughout their entire execution duration. This strategy, while suitable for traditional scientific computations, proves suboptimal for distributed ML training, especially those with INA acceleration. Without a careful investigation of INA communication patterns of ML jobs, current allocation strategies left INA resources idle and wasted during periods when there is no inter-host communication.

To address these issues, we introduce two key insights in the next subsections that support our system, MINA.

B. How to Reduce INA Resource Conflicts?

In MLaaS scenarios, training requests specify the requisite number of hosts for distributed ML training, and it is impossible to foresee future requests that have not yet arrived. Upon the arrival of a request, the controller must immediately determine the hosts for the job to run. To improve the request throughput, most previous works [17] model this as a bin packing problem, yet neglect to account for the proximity of hosts within a job. In this work, we argue that taking into account *the spatial characteristics of distributed ML training* when allocating jobs can effectively improve the request throughput of INA-enabled clusters.

a) Tree conflict: Each INA-enabled job necessitates the construction of an aggregation tree, as depicted in Figure 1. The switches in the aggregation tree are required to allocate memory for that job, which means that INA-enabled jobs not only occupy the hosts but also the switch memory within the associated aggregation tree. Given the limited on-chip memory available in switches, aggregation trees constructed for different jobs may conflict with one another, make it infeasible for these jobs to be simultaneously accelerated by INA. We call this phenomenon *tree conflict*.

b) Host proximity: If all hosts are under a single top-of-rack (TOR) switch, allocation of INA resources is confined to that specific TOR switch. Conversely, if hosts are distributed across multiple racks or even multiple rows, the aggregation tree becomes higher and necessitates more INA resources, leading to higher probability of tree conflicts. Therefore, it is desirable for hosts to maintain topological proximity to minimize tree conflicts.

However, ensuring such proximity could be challenging in MLaaS scenarios. This is because the requisite number of hosts of each training request ranges from one single host

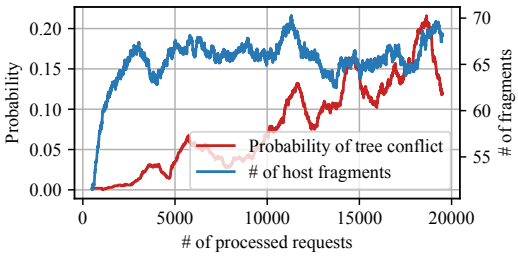

Fig. 2. Relationship between tree conflicts and host fragments

to several hundreds, and the execution time of each training job also varies. As hosts experience turnover—owing to the completion of running jobs and the arrival of new ones—the fragmentation within the cluster intensifies, analogous to disk or memory fragmentation, rendering the maintenance of node proximity increasingly difficult. Figure 2 illustrates the trends of the probability of tree conflict and the number of host fragments as the cluster processes more and more requests. The probability of tree conflict exhibits an ascending trajectory and has a strong positive correlation with the number of host fragments.

Given the correlation between tree conflicts and host fragments, we argue that reducing host fragments when allocating jobs can effectively minimize the probability of tree conflicts, thereby allowing more jobs to use INA acceleration.

C. How to Achieve Better INA Resource Scheduling?

INA resources are limited due to the high cost of the on-chip memory of programmable switches. Prior works [8], [18] address this issue by dividing the switch memory among multiple jobs. In this way, each job has its own dedicated portion of switch memory from the beginning to the end of the job. However, if there are too many jobs, the amount of memory allocated to each job may be less than the amount required to achieve the optimal bandwidth, resulting in an increased JCT [8], [18].

In this paper, we propose a new dimension for applying INA in multi-job scenarios: *the time-division multiplexing sharing of INA resources*. To maximize the benefit of INA resources by sharing them across multiple jobs, we need to achieve an in-depth understanding of how specifically INA acceleration contributes to better JCT in distributed ML training.

Most prior works which apply INA to distributed ML training [7], [8] model the training process as alternating computation and communication stages: in computation stages, workers perform forward and backward propagation to compute gradients; in communication stages, workers aggregate gradients and update model parameters; the computation and communication stages may overlap due to computation-communication pipelining.

However, we found this model inaccurate to predict the improvement in JCT when applying INA acceleration. To investigate the underlying cause, we analyze GPU and network traces recorded during the data-parallel training process of OPT-350M with and without INA acceleration, as illustrated in Figure 3. The trace in the upper portion of the figure depicts

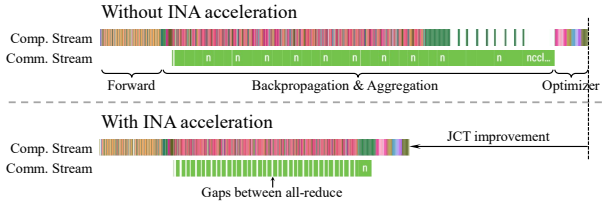

Fig. 3. Traces of data-parallel training with and without INA acceleration

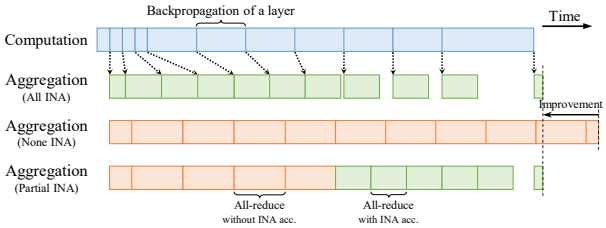

Fig. 4. INA acceleration effectiveness

the overlapping computation and communication stage without INA acceleration. However, when applying INA acceleration, the duration of the communication stage did not reduce in proportion to the INA acceleration ratio. On the contrary, gaps appeared within the communication phase, which made the improvement of JCT much less than expected, as shown in the lower portion of the figure. This phenomenon also appears in the training process of other models, or using different GPU and network configurations.

The reason for these communication gaps is that ML frameworks, such as PyTorch, divide the communication stage into multiple smaller tasks of all-reduce operations. An all-reduce operation is invoked to perform the gradient aggregation right after the GPU finishes the computation of the gradient of a given layer. When computation is faster than network aggregation, all-reduce operations will be queued in the network backend, which leads to back-to-back network aggregations. However, with the help of INA acceleration, the network queue is drained fast, leading to gaps between all-reduce operations.

Figure 4 illustrates the relationship between the computations of gradients and their corresponding aggregations. The arrows between the first and the second row indicate the dependencies between computation and communication operations: the aggregation can only start after the computation of the corresponding gradients finishes. First, we can see that if we use INA acceleration for all all-reduce operations, we can reduce JCT in comparison with the aggregation without using INA acceleration, as shown in the second and third rows in the figure. One interesting observation is that *we can achieve the same amount of JCT improvement by using INA acceleration for only parts of the all-reduce operations*, as shown in the second and fourth rows in the figure. It happens because JCT is determined by the end time of the last all-reduce operation. Accelerating the former all-reduce operations that are bottlenecked by computation does not truly speed up the communication stage, instead, it enlarges the gaps in the communication. With this insight, we argue that the existing coarse-grained scheduling methods that do not take communication patterns of ML jobs into consideration lead to

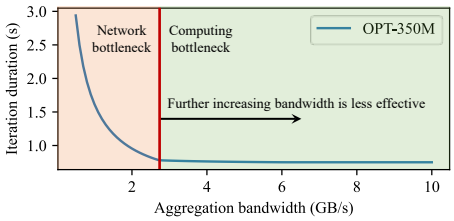

Fig. 5. Relationship between algorithm bandwidth and step duration

inefficient utilization of INA resources.

To understand this insight from another perspective, we plot the relationship between the algorithm bandwidth and the duration of one training step of OPT-350M in Figure 5. There is a clear turning point. Below the turning point, network is the bottleneck, increasing bandwidth can effectively reduce the duration of training steps. Surpassing the turning point, the limiting factor transitions to the computational speed, where further increasing in bandwidth yields diminishing returns on the reduction of JCT. In most cases, INA acceleration facilitates a bandwidth that exceeds this turning point. MINA capitalizes on the surplus bandwidth by time-dividing multiplexing sharing of INA resources across multiple jobs. It regulates the allocated bandwidth for each job to approximate the turning point, thereby optimizing the utilization efficiency of the limited INA resources.

III. Algorithm Design

A. Overview

Our proposed INA resource scheduling algorithm solves the problem in three steps: 1) upon the arrival of a new job, the job placement algorithm determines the optimal hosts for it to be allocated; 2) after the job is allocated, the tree building algorithm reconstructs the aggregation trees of all the running jobs in the cluster, including the newly arrived job; 3) during the training process, the INA resource sharing algorithm regulates the utilization of INA resources among all the running jobs to avoid INA resource conflicts. The first and second steps of the algorithm are applied upon the arrival of each job, they allocate INA resources in the space dimension. The third step of the algorithm is applied before each all-reduce operation of each job, it schedules INA resources in the time dimension.

It is worth noting that in the first step, the algorithm only allocates hosts for the newly arrived job without reallocating the running jobs. However, in the second step, the algorithm rebuilds the aggregation trees for all the running jobs, including the new one. This is because migrating running jobs inures much higher cost than switching the aggregation trees of running jobs. An evaluation of the overhead of tree migration can be found in Section V-D.

B. Job Placement Algorithm

The job placement algorithm is responsible for assigning hosts for each arriving job. According to the insight elaborated in Section II-B, we can effectively reducing the probability of tree conflicts and improve the average JCT by minimizing the number of host fragments.

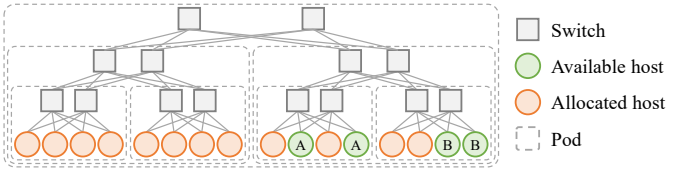

Fig. 6. Example of host fragments

Algorithm 1: Place job

Input: Allocate n hosts in the given *pod*
Output: Optimal *hosts* and *score*

Function `Allocate`(*pod*, n)**:**
  if $n = 0$ **then**
    return $\varnothing$, α·*pod*
  if $n =$ *# of available hosts in pod* **then**
    return *available hosts*, *pod*
  if $n >$ *# of available hosts in pod* **then**
    return $\varnothing$, $+\infty$
  $m \leftarrow$ # of sub pods in *pods*
  `// Hosts`
  $H \leftarrow$ array$[1..m][0..n]$ with value $\varnothing$
  `// Score`
  $S \leftarrow$ array$[1..m][0..n]$ with value $+\infty$
  for $j \leftarrow 0$ **to** n **do**
    $H[1,j]$, $S[1,j] \leftarrow$ `Allocate`$(subpod_1, j)$
  for $i \leftarrow 2$ **to** m **do**
    for $j \leftarrow 0$ **to** n **do**
      for $k \leftarrow 0$ **to** j **do**
        h, $s \leftarrow$ `Allocate`$(subpod_i, j)$
        if $S[i-1,j] + s < S[i,j]$ **then**
          $H[i,j] \leftarrow H[i-1,j] \cup h$
          $S[i,j] \leftarrow S[i-1,j] + s$
  return $H[m,n]$, $S[m,n]$

a) Definition of the number of host fragments: Unlike memory or disk fragments where resources are linearly organized, hosts in the cluster are typically organized in a tree topology, as shown in Figure 6. We call each subtree of the topology a *pod* and treat each host (*i.e.*, leaf node) as a pod containing only that host. Each node in the same pod is isomorphic and their order does not matter. The same applies to each child pod of the same parent pod. To illustrate, consider the allocation of a job that requires two hosts. Allocating this job to the hosts labeled as Ⓐ in the figure should be considered equivalent to allocating to the ones labeled as Ⓑ. Therefore, both Ⓐ and Ⓑ should be counted as two fragments, even though the latter pair is adjacent.

Given this property, we define the number of available host fragments of a given pod in a recursive approach as follows: if all of the hosts within the given pod are available, there is one single fragment; if none of them are available, there is zero fragment; otherwise, the number of fragments is computed as

the sum of the number of fragments inside each child pod. Similarly, we can define the number of fragments of hosts assigned to a given job by returning 1 if all hosts in the pod are allocated to that job and 0 if none. According to this definition, the example in Figure 6 contains four fragments of available hosts and five fragments of allocated hosts.

b) Target: We set the target of the algorithm as a weighted combination of two metrics: $A+\alpha \cdot B$, where A is the number of fragments of hosts allocated to the new job, B is the number of fragments of the remaining available hosts, and α is a hyperparameter used to control whether the algorithm should focus more on optimizing the current job, trying to optimize future jobs. We refer to $A + \alpha \cdot B$ as the *fragmentation score*, and use 0.5 as the default value for α.

c) Inputs and outputs: The algorithm is provided with the availability of all hosts in the cluster and the number of hosts requested by the arriving job. It outputs the optimal hosts for the job that minimizes the fragmentation score.

d) Solution: Our algorithm efficiently finds the optimal solution by using recursive dynamic programming. We use the function `Allocate` to compute the optimal allocation plan that allocates n hosts in the given pod. Invoking `Allocate` on the root pod, with n being the requested number of hosts of the arriving job, yields the final solution. The function returns a fragmentation score of infinity if the given pod contains fewer than n available hosts. We use `F[m,n]` to denote the optimal fragmentation score of allocating n hosts in the m^{th} sub pod, and use `S[m,n]` to denote the optimal fragmentation score of allocating a total of n hosts across the first m sub pods. `F[m,n]` can be obtained by invoking `Allocate(`$subpod_m$`,n)`, and `S[m,n]` can be calculated based on the following state-transition equation:

$$\texttt{S[m,n]} = \begin{cases} \texttt{F[m,n]} & \text{if } m = 1, \\ \min_{i=0}^{n} (\texttt{S[m-1,i]} + \texttt{F[m,n-i]}) & \text{if } m \geq 2. \end{cases}$$

When $m = 1$, `S[m,n]` is equal to `F[m,n]` by definition; when $m \geq 2$, `S[m,n]` is calculated by considering all possible distributions of the n hosts among the first m sub pods. For each possible number i of hosts allocated to the first $m-1$ sub pods, we calculate the score `S[m-1,i]` and combine it with the score `F[m,n-i]` for allocating the remaining $n-i$ hosts to the m^{th} sub pod. The minimum score across all possible distributions is then chosen as the optimal score for `S[m,n]`.

Algorithm 1 delineates a dynamic programming procedure to calculate `S[m,n]` as the result of `Allocate`, and also handles edge cases. Two implementation tricks can be applied to optimize this procedure. First, calculating the result of `Allocate(`pod`,`n`)` also yields the result of `Allocate(`pod`,`i`)` for all i from 0 to $n-1$, which can be cached to prevent redundant calculations. Second, calculating `S[m,n]` only requires the information from the previous row (*i.e.*, `S[m-1,:]`), therefore, we can reduce memory usage by only storing the last row of S and H. With these optimizations applied, the job placement algorithm has the time complexity of $O(KNM^2)$ and the space complexity of $O(KM)$, where

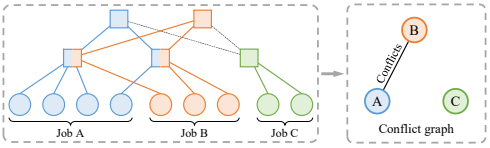

Fig. 7. Construction of conflict graph

K is the height of the tree topology, N is the total number of hosts in the cluster, and M is the required number of hosts of the newly arrived job.

C. Tree Building Algorithm

The tree building algorithm is responsible for constructing the aggregation trees of all the running jobs in the cluster. It aims to maximize the number of jobs allowed to use INA acceleration concurrently by reducing INA resource conflicts. The algorithm takes as input the topology of the cluster and the hosts allocated to each running job, and outputs an aggregation tree for each job. The algorithm also divides the jobs into several *sharing groups*. Jobs within the same sharing group can share INA resources by time-division multiplexing. Jobs that do not belong to any sharing group will not be allowed to use INA acceleration.

First, the algorithm needs to find the largest set of jobs whose aggregation trees do not conflict with each other. We model this as an *Integer Linear Programming* (ILP) problem. Let n denote the number of running jobs, k denote the number of possible aggregation trees of each job, $\mathcal{T}_{i,j}$ denote the j^{th} possible aggregation tree of the i^{th} job, $x_{i,j} \in \{0,1\}$ denote whether the tree $\mathcal{T}_{i,j}$ is chosen, C denote the maximum number of INA-enabled jobs that are supported using the same programmable switch at the same time without affecting aggregation bandwidth, and $\mathcal{H}_s = \{(i,j) \mid \text{switch } s \in \mathcal{T}_{i,j}\}$ denote the set of aggregation trees that contain the switch s. The problem can be formulated as follows:

$$\max \quad \sum_{i=1}^{n} \sum_{j=1}^{k} x_{i,j}$$

$$\text{s.t.} \quad \sum_{j=1}^{k} x_{i,j} \leq 1, \quad \forall i : 1 \leq i \leq n, \tag{1}$$

$$\sum_{(i,j) \in \mathcal{H}_s} x_{i,j} \leq C, \quad \forall \text{switch } s, \tag{2}$$

$$x_{i,j} \in \{0,1\}, \quad \forall i : 1 \leq i \leq n, \ \forall j : 1 \leq j \leq k, \tag{3}$$

where constraint (1) ensures that only one aggregation tree is chosen for each job and constraint (2) ensures that the INA capacity of each switch is never exceeded.

Most InfiniBand-based data centers today are equipped with HDR or NDR switches that support SHARP [10] as the INA protocol (introduced in Section II-A). The older generation HDR switches only support one active INA job per switch, while the newer generation NDR switches support one active INA job per port. Given this constraints, we can specialize the above formulation and model it as a *Maximum Independent Set* (MIS) problem. As exemplified in Figure 7, we construct the conflict graph where each vertex represents a possible aggregation tree of a job, and there is an edge between two vertices if and only if the corresponding aggregation trees

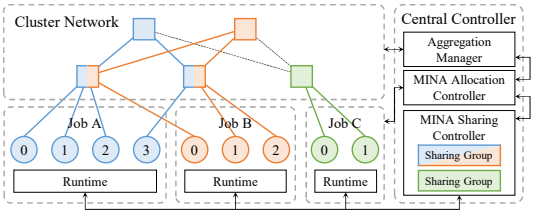

Fig. 8. MINA architecture

conflicts with each other, *i.e.*, not vertex-disjoint in HDR scenarios or not edge-disjoint in NDR scenarios.

Both ILP problem and MIS problem are NP-hard. Therefore, we employed approximation algorithms [19], [20] to solve them. To further enhance efficiency, we limit the number of possible aggregation trees per job by randomly selecting k candidates, instead of all. Section V-C presents evaluations that demonstrates the impact of k on the algorithm's efficiency and effectiveness.

After the above step, we have found the largest set of aggregation trees that have no INA resource conflicts among them. These trees can be partitioned into several non-overlapping *sharing groups*. In MIS problem, each vertex in the maximum independent set constitutes a sharing group. However, a subset of jobs remains, for which no corresponding aggregation trees are members of any sharing group.

To facilitate time-division multiplexing of INA resources, we opportunistically add these remaining aggregation trees to sharing groups. This induces tree conflicts, but will be handled by the sharing controller. Formally, all aggregation trees can be divided into three categories: 1) those have been allocated to a sharing group, 2) those conflicting with only one sharing group, and 3) those conflicting with multiple sharing groups. We then incorporate the trees from the second category into the respective sharing group with which they conflict, while still ensuring that each job has at most one aggregation tree.

D. INA Resource Sharing Algorithm

The INA resource sharing algorithm is responsible for regulating the scheduling of INA resources among jobs within a sharing group to avoid INA resource conflicts. It targets to improve the average JCT by selectively accelerating *key* all-reduce operations that have a large impact on JCT. The algorithm is provided with the running status of each job in the sharing group (tracked by the sharing controller), and the communication pattern of each job (collected by MINA's runtime). The algorithm is invoked on all hosts of a job before each all-reduce operation, determining whether the next all-reduce operation should use INA acceleration. Details of the consensus mechanism and the methodology for trace collection are elucidated in Section IV-B, while in this section we focus on the algorithm itself.

The observation in Figure 4 shows the same JCT improvement by using INA acceleration for only parts of the all-reduce operations in one training step. If there is only one job, we can simply partition the all-reduce operations into two portions where the latter uses INA acceleration while the former does not, as shown in the fourth row of Figure 4. When there are multiple jobs, this simple method no longer works, the acceleration portion of each job may conflicts and there may be wasted periods of INA resources when no job is in its acceleration portion. To address the issue, we use a heuristic approach by scoring each all-reduce operation according to the improvement in JCT per unit of INA acceleration time, and prioritizing those all-reduce operations that have a higher score to use INA resources.

IV. System Design

In this section, we delineate the system design of MINA that supports the algorithms introduced in Section III. The design of our system is predicated on two core principles: 1) to ensure minimal overhead to achieve optimal performance; 2) to support all kinds of ML models and distributed ML training methods without modifying application-layer frameworks (*e.g.*, PyTorch [21], TensorFlow [22]), thereby enabling seamless integration of our system.

Figure 8 illustrates the architecture of MINA, which consists of three primary components: *the allocation controller*, responsible for job placement and aggregation tree construction, *the sharing controller*, responsible for time-division multiplexing sharing of INA resources across multiple jobs, and *the runtime library* used by each job, responsible for collecting communication patterns of each job and interacting with the sharing controller. The two controllers are deployed on a dedicated server which acts as a central controller of the cluster. They are co-located with the aggregation manager (*e.g.*, SHARP AM [23] in InfiniBand-based clusters), which is responsible for configuring INA environments on switches and allocating switch memory for each INA job.

A. Allocation Controller

The allocation controller serves as the entry point of the system. Upon receiving MLaaS training requests, it assigns hosts to the job using the algorithm described in Section III-B and constructs the aggregation trees using the algorithm described in Section III-C.

a) Aggregation tree migration: When a new job arrives, the tree building algorithm may decide to rebuild the aggregation trees for some running jobs to make room for the newly arrived job. Therefore, the allocation controller also supports aggregation tree migration, which enables shifting aggregation trees of running jobs on-the-fly without affecting their execution. The INA resources associated with aggregation trees should be released first before assigned to other jobs, as failure to do so could lead to a deadlock scenario.

With this constraint, we design the process of aggregation tree migration as the following two steps. First, all the jobs that need to shift aggregation trees are instructed to release their INA resources after the completion of the current INA-enabled all-reduce operation (since it is not feasible to interrupt an ongoing all-reduce operation), or release the resources immediately if not in use. Then, the jobs keep trying to acquire the new aggregation tree, which will succeed once all jobs that held the resources have released them. This migration process

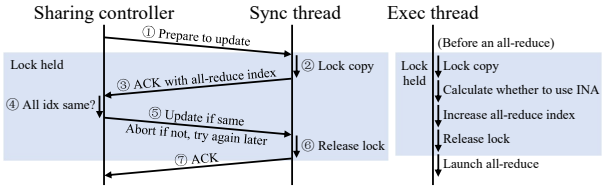

Fig. 9. Synchronization protocol

is designed to be non-blocking, meaning that subsequent all-reduce operations keeps being executed without using INA acceleration, until the new aggregation tree is successfully acquired. The overhead of this migration process is evaluated in Section V-B.

B. Sharing Controller and Runtime

The main responsibility of the sharing controller and the runtime library is to facilitate efficient coordination among hosts. They should ensure that hosts within the same job reach a consensus on whether to use INA acceleration for each all-reduce operation. Moreover, they should also safeguard against simultaneous usage of the same INA resources by multiple jobs, thereby preventing conflicts.

a) Host coordination: A naive solution is for each host to query the sharing controller before each all-reduce operation about whether to use INA acceleration. Nonetheless, this method incurs an overhead of one additional round-trip time (RTT) per all-reduce operation, incurring prohibitive costs.

Therefore, we seek a solution free of additional network communications before each all-reduce operation. To achieve this, we let the INA resource sharing algorithm run locally on each host, instead of on the sharing controller. And, we utilize an improved version of the two-phase commit (2PC) protocol to ensure consistency, as illustrated in Figure 9. There are two threads running on each host, the synchronization thread, dedicated to maintaining an updated local copy of the sharing group's status, and the execution thread, tasked with executing the INA resource sharing algorithm and launching all-reduce operations. To avoid data races, a mutex lock is employed to guard the copy on each host.

The protocol proceeds as follows: when the sharing group's status changes, the controller sends "prepare to update" messages to synchronization threads of all hosts in the sharing group (step ①). Upon receiving the message, each host tries to acquire the lock (step ②). After the lock is successfully acquired, it sends back an ACK message, including the index of the current all-reduce operation (step ③). The controller verifies whether the indices received from all hosts of the same job are identical (step ④). If so, it sends the information used to update the status; otherwise, it aborts the process and schedules a retry after an interval (step ⑤). Finally, each host updates its status, releases the lock (step ⑥), and sends back an ACK message (step ⑦). The process on the execution thread is straightforward: before launching all-reduce operations, it calculates whether to use INA acceleration using the INA resource sharing algorithm with the lock acquired, and also increments the index of all-reduce operations.

The correctness of this synchronization process can be attributed to two reasons. First, the mutex locks of all hosts within the job are always held during the update wherein the system is in an inconsistent state (step ② to step ⑥), preventing the next all-reduce operation from being launched. Second, the sharing controller checks the consistency of all-reduce indices, which avoids the scenario where some hosts have launched the all-reduce operation while others have not.

b) Communication pattern collection: The sharing controller and the runtime library are also responsible for collecting communication patterns of each job, which are required by the INA resource sharing algorithm. These patterns should also contain the information about dependencies between computation and communication operations, as illustrated using arrows in Figure 4. Such dependencies are critical to ascertain the potential improvement in JCT that may be realized by applying INA acceleration on certain all-reduce operations.

However, the collection of communication patterns is limited to utilizing network-layer information by design, without using any application-layer data, such as the ML model architectures. This approach obviates the necessity for modifying application-layer frameworks, thereby ensuring that our system can be seamlessly integrated and remain compatible with any job that is dynamically linked with our modified network-layer library.

Considering that the durations of computation operations are constant and not influenced by the efficiency of communication, we model the computation-to-communication dependency as the earliest starting time of each communication operation, and the communication-to-computation dependency as the latest end time of the last communication operation without affecting JCT. We propose that the existing asynchronous interface of network libraries are sufficient to capture these dependencies. Specifically, we use the time when each communication operation is scheduled (by calling `MPI_Iallreduce` or `ncclAllReduce`) as the former dependency, and the time when the application-layer framework starts waiting for results of communication operations (by calling `MPI_Waitall` or `cudaStreamSynchronize`) as the latter dependency. These data are collected at the beginning of each job after some warm-up iterations.

C. Combination with NCCL and SHARP

While MINA is designed to be generic to the implementations of network libraries and INA, we would like to highlight our design choices when integrating MINA with NCCL and SHARP, the predominant choice in top-tier data centers [13].

A significant challenge arises when combining MINA's sharing controller with NCCL. NCCL assumes that no other jobs are running simultaneously on the same server, the available link bandwidth and INA availability are therefore considered consistent throughout the job. Under such assumption, NCCL determines how to execute all-reduce operations (*e.g.*, enabling INA, deciding message chunk size, GPU thread count, etc.) immediately upon all-reduce requests arrive, which consists of the selection, configuration, and queuing of an all-reduce kernel into CUDA streams. However, in MLaaS scenarios

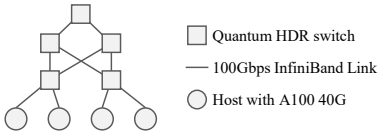

Fig. 10. Testbed topology

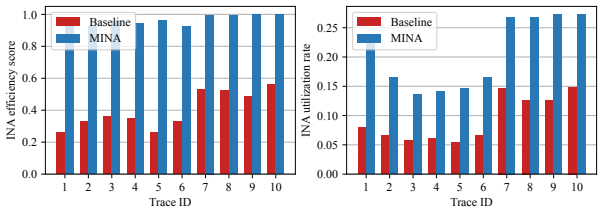

Fig. 11. Overall performance of MINA

where INA resources are shared among jobs, INA availability may change during the execution of previous all-reduce kernels. MINA's sharing controller necessitates the functionality to determine the all-reduce algorithm (whether to use INA) just before execution, rather than upon request arrival.

To address this, MINA does not immediately queue all-reduce kernels into CUDA streams upon request arrival. Instead, it maintains these requests in its own queue and sequentially places them into CUDA streams. MINA guarantees that only one kernel is present in the CUDA stream at any given time, with new kernels enqueued once the previous one completes. This can be achieved through host callback functions (*i.e.*, `cudaLaunchHostFunc`). By adopting this approach, MINA attains real-time control over all-reduce kernel configurations, albeit with an insignificant increase in kernel launch latency, as demonstrated in Section V-D.

V. EVALUATION

A. Experiment Settings

a) Implementation: The prototype includes the allocation controller, sharing controller, and runtime library, as described in Section IV. Our implementation leverages PyTorch [21] as the application-layer framework and a modified version of NCCL [24] and its SHARP plugin [25] as the network-layer library. We deploy the system on an InfiniBand-based cluster and utilize SHARP [10] as the INA implementation.

b) Simulator: To evaluate MINA's performance on large-scale data centers, we implement an event-driven communication-operation-level simulator that simulates the arrival and completion of each job, as well as every computation and communication operation of all running jobs using the model proposed in Section II-C. This simulator has around 3,500 lines of C++ code.

c) Testbed and simulation configurations: The topology of our real testbed is depicted in Figure 10. It comprises five Quantum HDR switches [14] with SHARP enabled and four hosts, each equipped with two A100-40G GPUs, interconnected via 100 Gbps InfiniBand links. We perform large-scale simulations in a three-layer fat-tree network topology with a degree of 16, which consists of 1024 hosts and 320 switches, interconnected via 200 Gbps links. The simulated switches emulate the features of Quantum-2 NDR switches [26] that support one active INA job per port, representing the state-of-the-art network facilities in top-tier data centers. We use 10 traces of ML training job requests in real MLaaS scenarios from Microsoft and use these traces to randomly generate 2,000 requests as inputs to our system for each evaluation.

TABLE I
RESULTS OF ABLATION STUDY

JP	TB	RS	INA Eff. Score	INA Util. Rate
–	–	–	0.263	0.080
✓	–	–	0.483 (+0.220)	0.129 (+0.049)
✓	✓	–	0.900 (+0.417)	0.223 (+0.094)
✓	✓	✓	**0.982** (+0.082)	**0.224** (+0.001)
–	✓	✓	0.947 (-0.035)	0.199 (-0.025)
✓	–	✓	0.979 (-0.003)	0.123 (-0.101)
✓	✓	–	0.900 (-0.082)	0.223 (-0.001)

d) Workloads: We select seven prevalent ML models in the fields of NLP and CV with different sizes and complexities: BERT-base, BERT-large, OPT-125M, OPT-350M, OPT-1.3B, ViT-base, and ViT-large.

e) Baselines: To the best of our knowledge, MINA is the first to apply INA in MLaaS scenarios and schedule INA resources in both spatial and temporal dimensions. Existing works are either INA-unaware [27], [28], [29] or inapplicable to MLaaS scenarios [7], [8], [30]. Therefore, we compare MINA with vanilla SHARP and establish a set of baseline algorithms reflective of current practices within MLaaS data centers. For job placement, the baseline algorithm assigns the first n available hosts to the newly arrived job, akin to the default strategy employed by Slurm [16]. For tree construction, the baseline approach opportunistically builds a tree for the new job if it has no conflicts with existing trees. For INA resource sharing, the baseline algorithm lets jobs greedily use INA for each all-reduce operation whenever available.

f) Metrics: The efficacy of MINA is evaluated through two performance metrics: the *INA efficiency score* and the *INA utilization rate*. The INA efficiency score is computed as the average JCT of all jobs, normalized to a scale where 0.0 represents the baseline JCT without any INA acceleration, and 1.0 represents the optimal JCT achievable with INA acceleration for every all-reduce operation. The score is directly proportional to the performance, with higher values indicating better performance. It is expressed mathematically as:

$$\text{INA Eff. Score} = \frac{\text{avg. JCT}_{\text{none-INA}} - \text{avg. JCT}_{\text{evaluated}}}{\text{avg. JCT}_{\text{none-INA}} - \text{avg. JCT}_{\text{all-INA}}}.$$

Meanwhile, the INA utilization rate is defined as the proportion of the job's running time during which INA acceleration is applied, averaged over all jobs.

g) Fidelity of large-scale simulation: Our simulator is built upon SHARP's implementation and insights from experiment results on our real testbed. In the job placement and tree building steps, our simulator faithfully follows SHARP's implementation, which is proven on our small-scale testbed and will not change on large-scale topology.

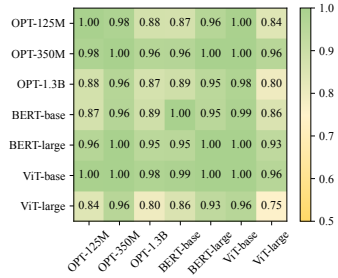

Fig. 12. Sharing performance

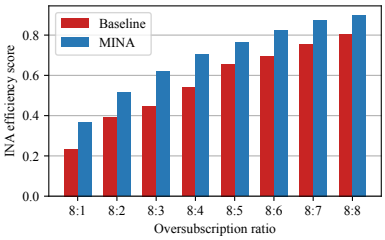

Fig. 13. Performance of job placement algorithm with different oversubscription ratios

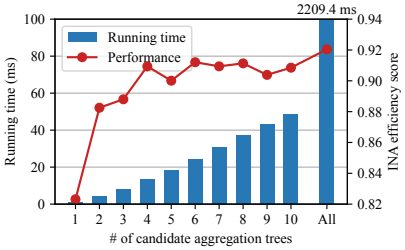

Fig. 14. Performance and overhead of tree building algorithm

B. Overall Performance

Figure 11 presents the results of large-scale simulations, showcasing the INA efficiency score and INA utilization rate across different traces. Specifically, MINA achieves an INA efficiency score as high as 0.9998, with an average score of 0.968, indicating a significant improvement over the baseline average of only 0.399. Such an efficiency score nearing 1.0 suggests that almost all the running jobs are as fast as if they were using INA acceleration exclusively, even in the scenario where INA resources are limited. MINA also increases the INA utilization rate by a factor of to up to 2.81, averaging 2.32 times the baseline.

a) Ablation study: Table I shows the results of the ablation study using trace #1, wherein we use JP, TB, and RS to represent the job placement, tree building, and INA resource sharing algorithms, respectively. The upper part of the table substantiates the effective contribution of each of the three proposed algorithm modules to the INA efficiency score. When we enable these three modules in order, they yields incremental enhancements of 0.220, 0.417, and 0.082 to the INA efficiency score, and reach a final score of 0.982. The lower part of the table illustrates that disabling any single module does not significantly compromise the INA efficiency score, thereby underscoring the robustness of our system.

C. Evaluation of Algorithm Modules

a) Job placement algorithm: We evaluate the performance of the job placement algorithm in fat-tree topologies with different oversubscription ratios. The oversubscription ratio is defined as the ratio between the number of downward and upward links of switches in the cluster. Here we have disabled the INA resource sharing algorithm (*i.e.*, use the same configuration as the last row of Table I) to highlight the impact of the job placement algorithm. As shown in Figure 13, the INA efficiency score of our algorithm exceeds the baseline by as much as 0.173, with an average improvement of 0.131.

b) Tree building algorithm: Figure 14 shows the performance and the running time of the tree building algorithm using different numbers of candidate aggregation trees (denoted as k). We disabled the INA resource sharing algorithm to highlight the impact of the tree building algorithm. Results show that the algorithm's running time grows rapidly with the increase of k, while the INA efficiency score approaches and remains close to the optimal value when k surpasses the value of 4. Therefore, we choose the default k to be 5, providing a judicious balance between efficiency and performance.

c) INA resource sharing algorithm: To evaluate the performance of our proposed INA resource sharing algorithm, we test the time-division multiplexing sharing scheme across all combinations of the workloads listed in Section V-A and run 1,000 seconds for each combination. Figure 12 presents the INA efficiency score for each combination. The average score reaches 0.939, with 33 of the 49 combinations surpassing 0.95. Without our proposed time-division multiplexing sharing of INA resources, the baseline score would only be 0.5 by definition. This result suggests that our INA resource sharing algorithm achieves almost the same JCT improvements as applying INA acceleration exclusively for each job.

D. Overhead Measurement

a) Overhead of aggregation tree migration: We conducted 500 times of aggregation tree migration using all four hosts of our testbed. The results show that the average migration time is 559 ms, of which 125 ms is for releasing the current occupied INA resources on switches and 294 ms is for reallocating INA resources of the new aggregation tree. Furthermore, the frequency of aggregation tree migration is low and our algorithm keeps the majority of tress unchanged. Simulations have demonstrated that an average job is subjected to 5.02 aggregation tree migrations throughout its execution, which equates to a total INA downtime of only 2.81 s.

b) Overhead of INA resource sharing controller: As introduced in Section IV-B, the host coordination itself (*i.e.*, the 2PC protocol) incurs no additional overhead to the training process because it is performed on the separate synchronization thread. To measure the additional overhead brought by the proposed callback-based CUDA kernel launching approach, we launch 1 million empty kernels using this approach. Results show that it takes 43.8 μs per launch, almost 20 times higher than the overhead of a conventional kernel launch (2.23 μs). Nevertheless, this overhead remains insignificant when compared to the typical duration of an all-reduce kernel (ranging from tens to hundreds of milliseconds).

VI. Conclusion

In this paper, we show that considering the spatial and temporal characteristics of ML jobs when scheduling INA resources can effectively improve the average JCT and INA resource utilization. We design and prototype our system, MINA, which efficiently allocates INA resources and schedules them across multiple jobs. Our evaluations indicate that MINA is effective, efficient, and robust on various configurations.

REFERENCES

[1] H. Yeo, Y. Jung, J. Kim, J. Shin, and D. Han, "Neural adaptive content-aware internet video delivery," in *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2018, pp. 645–661.

[2] S. Narasimhan, "Nvidia clocks world's fastest bert training time and largest transformer based model, paving path for advanced conversational ai," *NVidia Dev. Blog. URL https://devblogs. nvidia. com/training-bert-with-gpus/(accessed 8.21. 19)*, 2019.

[3] M. Ribeiro, K. Grolinger, and M. A. Capretz, "Mlaas: Machine learning as a service," in *IEEE 14th International Conference on Machine Learning and Applications (ICMLA)*. IEEE, 2015.

[4] R. Krashinsky, O. Giroux, S. Jones, N. Stam, and S. Ramaswamy, "NVIDIA ampere architecture in-depth," *NVIDIA blog: https://devblogs. nvidia. com/nvidia-ampere-architecture-in-depth*, 2020.

[5] S. Knowles, "Graphcore," in *2021 IEEE Hot Chips 33 Symposium (HCS)*. IEEE, 2021, pp. 1–25.

[6] L. Gwennap, "Habana offers gaudi for ai training," *Microprocessor Report, Tech. Rep., jun*, 2019.

[7] A. Sapio, M. Canini, C.-Y. Ho, J. Nelson, P. Kalnis, C. Kim, A. Krishnamurthy, M. Moshref, D. R. Ports, and P. Richtárik, "Scaling distributed machine learning with in-network aggregation," in *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2021.

[8] C. Lao, Y. Le, K. Mahajan, Y. Chen, W. Wu, A. Akella, and M. Swift, "ATP: In-network aggregation for multi-tenant learning," in *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2021.

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

[10] R. L. Graham, D. Bureddy, P. Lui, H. Rosenstock, G. Shainer, G. Bloch, D. Goldenerg, M. Dubman, S. Kotchubievsky, V. Koushnir, L. Levi, A. Margolin, T. Ronen, A. Shpiner, O. Wertheim, and E. Zahavi, "Scalable hierarchical aggregation protocol (sharp): A hardware architecture for efficient data reduction," in *2016 First International Workshop on Communication Optimizations in HPC (COMHPC)*, 2016.

[11] R. L. Graham, L. Levi, D. Burredy, G. Bloch, G. Shainer, D. Cho, G. Elias, D. Klein, J. Ladd, O. Maor *et al.*, "Scalable hierarchical aggregation and reduction protocol (SHARP) streaming-aggregation hardware design and evaluation," in *International Conference on High Performance Computing*, 2020.

[12] "ND A100 v4-series," https://learn.microsoft.com/en-us/azure/virtual-machines/nda100-v4-series, 2023.

[13] "Top500 list - june 2023," https://www.top500.org/lists/top500/list/2023/06/, 2023.

[14] "NVIDIA quantum HDR 200gb/s InfiniBand switch," https://www.nvidia.cn/networking/infiniband/qm8700, 2020.

[15] 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)*. Boston, MA: USENIX Association, Feb. 2019, pp. 485–500. [Online]. Available: https://www.usenix.org/conference/nsdi19/presentation/gu

[16] A. B. Yoo, M. A. Jette, and M. Grondona, "Slurm: Simple linux utility for resource management," in *Workshop on job scheduling strategies for parallel processing*. Springer, 2003, pp. 44–60.

[17] S. Kumaraswamy and M. K. Nair, "Bin packing algorithms for virtual machine placement in cloud computing: a review," *International Journal of Electrical and Computer Engineering*, vol. 9, no. 1, p. 512, 2019.

[18] Y. He, W. Wu, Y. Le, M. Liu, and C. Lao, "A generic service to provide in-network aggregation for key-value streams," in *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, 2023, pp. 33–47.

[19] L. Chang, W. Li, and W. Zhang, "Computing a near-maximum independent set in linear time by reducing-peeling," in *Proceedings of the 2017 ACM International Conference on Management of Data*, 2017, pp. 1181–1196.

[20] M. Xiao and H. Nagamochi, "Exact algorithms for maximum independent set," *Information and Computation*, vol. 255, pp. 126–146, 2017.

[21] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga *et al.*, "Pytorch: An imperative style, high-performance deep learning library," *Advances in neural information processing systems*, vol. 32, 2019.

[22] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard *et al.*, "TensorFlow: a system for Large-Scale machine learning," in *12th USENIX symposium on operating systems design and implementation (OSDI 16)*, 2016, pp. 265–283.

[23] "Sharp aggregation manager," https://docs.nvidia.com/networking/display/ufmsdnappcliv4142/sharp+aggregation+manager, 2023.

[24] "NVIDIA collective communication library (NCCL)," https://developer.nvidia.com/nccl, 2017.

[25] "RDMA and SHArP plugins for NCCL library," https://github.com/Mellanox/nccl-rdma-sharp-plugins, 2020.

[26] "Nvidia quantum-2 infiniband platform," https://www.nvidia.cn/networking/quantum2/, 2022.

[27] D. Narayanan, K. Santhanam, F. Kazhamiaka, A. Phanishayee, and M. Zaharia, "Heterogeneity-Aware cluster scheduling policies for deep learning workloads," in *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, 2020, pp. 481–498.

[28] 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.

[29] 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)*, 2020, pp. 533–548.

[30] Y. He, W. Wu, Y. Le, M. Liu, and C. Lao, "A generic service to provide in-network aggregation for key-value streams," in *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, 2023, pp. 33–47.