# Training Job Placement in Clusters with Statistical In-Network Aggregation


Bohan Zhao
Tsinghua University
zbh20@mails.tsinghua.edu.cn

Wei Xu
Tsinghua University
weixu@tsinghua.edu.cn

Shuo Liu
Huawei Technologies Co., Ltd.
liushuo15@huawei.com

Yang Tian
Huawei Technologies Co., Ltd.
tianyang21@huawei.com

Qiaoling Wang
Huawei Technologies Co., Ltd.
wangqiaoling@huawei.com

Wenfei Wu*
Peking University
wenfeiwu@pku.edu.cn



## Abstract

In-Network Aggregation (INA) offloads the gradient aggregation in distributed training (DT) onto programmable switches, where the switch memory could be allocated to jobs in either synchronous or statistical multiplexing mode. Statistical INA has advantages in switch memory utilization, control-plane simplicity, and management safety, but it faces the problem of cross-layer resource efficiency in job placement. This paper presents a job placement system NetPack for clusters with statistical INA, which aims to maximize the utilization of both computation and network resources. NetPack periodically batches and places jobs into the cluster. When placing a job, NetPack runs a steady state estimation algorithm to acquire the available resources in the cluster, heuristically values each server according to its available resources (GPU and bandwidth), and runs a dynamic programming algorithm to efficiently search for servers with the highest value for the job. Our prototype of NetPack and the experiments demonstrate that NetPack outperforms prior job placement methods by 45% in terms of average job completion time on production traces.



***CCS Concepts:*** • **Networks** → **Programmable networks**.

***Keywords:*** In-Network Aggregation, Distributed Training, Placement

**ACM Reference Format:**
Bohan Zhao, Wei Xu, Shuo Liu, Yang Tian, Qiaoling Wang, and Wenfei Wu. 2024. Training Job Placement in Clusters with Statistical In-Network Aggregation. In *29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1 (ASPLOS '24), April 27-May 1, 2024, La Jolla, CA, USA.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3617232.3624863



*Wenfei Wu is the corresponding author, and is with the School of Computer Science at Peking University.





*ASPLOS '24, April 27-May 1, 2024, La Jolla, CA, USA*





ACM ISBN 979-8-4007-0372-0/24/04.
https://doi.org/10.1145/3617232.3624863


## 1 Introduction

Recently, a family of *In-Network Aggregation (INA)* solutions is proposed to accelerate distributed modeling training (DT) [13, 22, 27, 37, 41]. These solutions achieve the performance gain by offloading the gradient *AllReduce* operation onto programmable switches. INA solutions introduce a new resource — *switch memory* [1] — for data stream aggregation, whose allocation to a job critically impacts job efficiency. Thus, when a job manager places multiple jobs in a shared cluster, it needs a *cross-layer* planning of both computation resource (GPU) and network resource (link bandwidth and switch memory). A good placement plan is supposed to balance jobs among the cluster servers and use multi-dimensional resources evenly.

We classify existing INA solutions into two categories according to whether the switch memory is isolated or shared among multiple concurrent jobs; the INA solutions with isolated memory among jobs are called *synchronous INA* [13, 27, 37, 41], and the ones with dynamically shared memory are called *statistical INA* [10, 22]. This paper focuses on statistical INA solutions and analyzes its advantages in switch memory utilization, control-plane simplicity, and management safety (Section 2.2).

Unfortunately, existing job placement solutions [22, 37] are not designed for clusters with statistical INA, and they could result in imbalanced and insufficient resource-sharing plans, which further degrades the individual and overall job efficiency. On the one hand, there are more critical resources — computation and network resources in INA, and few existing solutions consider all of them in the problem formulation, which leads to suboptimal results. On the other hand, computation and network resources are managed in different ways by nature, and existing solutions cannot trivially add another kind of resource into their provision model nor enforce them.

[1] Switch memory in this paper means the on-chip register memory, which is durable to store cross-packet states.

In detail, a DT job manager usually makes *centralized* allocation and enforcement of computation resources (GPU) to jobs; but statistical INA applies *decentralized* allocation of network resources to jobs (jobs contend for network resources and converge to a steady state with max-min fair share), which does not fit into the computation resource management model.

This paper presents a system named NetPack for job placement with statistical INA. NetPack targets maximizing the overall job efficiency by scheduling cross-layer and multi-dimensional resources to jobs. NetPack batches and places user jobs periodically into the cluster. When placing a job, NetPack heuristically values each server according to the remaining resources (GPU and bandwidth) on the server and the impact on existing jobs and selects the set of servers with the highest value for the job (Section 3.2). NetPack overcomes the following two challenges.

First, the network resource allocation is decentralized; thus, the controller does not have a real-time view of the resource usage. Meanwhile, since switch memory allocation in statistically INA is transient (tens of microsecond, Section 2.2), the controller-based measurement becomes expensive and inaccurate. Without critical resource availability, the job manager can hardly accurately value each server. NetPack "estimates" the resource availability using a water-filling algorithm. Different from existing water-filling algorithms [5] that estimate resource sharing of multiple types independently, NetPack customizes the algorithm to jointly estimate two-resource sharing of INA — link bandwidth and switch memory, considering their mutual dependencies (Section 4).

Second, the cross-layer and multi-dimensional resources make the search space for optimal placement exponential. Naïve methods such as Mixed Integer Programming (MIP) cannot give satisfactory solutions in an acceptable time. We observe that the value of a placement plan can be recursively computed by placing a subset of workers in the subgraph of the cluster. Thus, we propose a bottom-up dynamic programming algorithm, which searches for the highest-value plan in polynomial time (Section 5).

We prototype NetPack on a 5-server testbed and simulate it on large-scale clusters. We evaluate NetPack with production and artificial job traces. The experiment results demonstrate that NetPack can improve over-job efficiency in various conditions, e.g., up to 45% when replaying production traces on a 5-server testbed. This paper makes the following contributions.

- We classify the switch memory usage modes in INA and analyze their differences. We also identify the necessity and challenges of designing a job placement system for DT jobs with statistical INA.
- We design NetPack to place jobs in a cluster with statistical INA, which improves the overall job efficiency. NetPack devises a new water-filling algorithm to derive the resource availability in the cluster and a dynamic programming algorithm to search for a high-value placement plan for a job efficiently.
- We prototype the system and conduct extensive experiments to show that NetPack improves the overall job efficiency and resource utilization.

## 2 Classification and Comparison of INA Modes

Existing INA solutions use the switch memory in synchronous or statistical multiplexing modes, and statistical INA has advantages in resource utilization, control-plane simplicity, and management safety.

### 2.1 INA Preliminaries

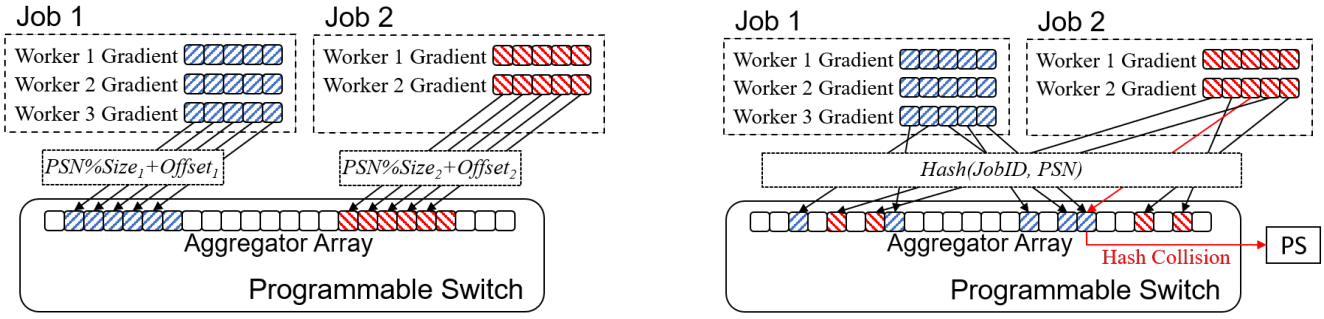
A DT job has multiple workers, each with one or several GPUs. All workers have a synchronized model. Each worker holds a non-overlapping data set partition. The model training is an iterative process with multiple *iterations*; in each iteration, a worker uses the model and its local data partition to compute a *gradient*. All workers' gradients are aggregated, and the result is returned to each worker. Then, each worker updates its model locally and proceeds to the next iteration. The gradient aggregation process is also called *AllReduce*. There several existing methods to perform AllReduce, such as Ring AllReduce [40], Parameter Servers (PS)[24], half-doubling[40], etc.

INA accelerates AllReduce by offloading the gradient aggregation onto a programmable switch. On each worker, the gradient is chunked into a sequence of packets, each with a *Packet Sequence Number (PSN)*. All workers start their PSN from 0. The switch memory is organized as an *aggregator array*, each aggregator accessed by its index in the array. When a packet arrives at the switch, the switch addresses the packet to an aggregator by its PSN, where the aggregator merges packets from different workers. When the aggregator completes the aggregation, it forwards the aggregation result to downstream devices (e.g., the parameter server (PS) or all workers). The switch releases the aggregator for future packets; there are two ways of aggregator recycling depending on the addressing methods as we describe below (Section 2.2).

INA accelerates the AllReduce operation in two aspects. First, it aggregates the raw data streams into one result stream, reducing traffic volume and the consequent transmission time. Second, the switch chip can perform the aggregation operation faster than the CPU. In addition, INA could also free up the network bandwidth usage, which is friendly to multi-job bandwidth sharing in the cluster [11].

### 2.2 Two Ways of INA Memory Management

Although INA could promote job performance, it consumes a new resource — *switch memory*, which can significantly




**(a)** Synchronous INA [27, 37, 44]

**(b)** Statistical INA [22]

**Figure 1.** Illustration of two INA modes. In synchronous INA, Job 1 uses memory region $(Offset_1, Size_1)$, and Job 2 uses $(Offset_2, Size_2)$; in statistical INA, both jobs share the whole memory.

impact performance. Insufficient switch memory provision could cancel the performance gain or even degrade the job performance [44]. By summarizing existing INA solutions, the switch memory is provided to jobs in two modes — synchronous multiplexing and statistical multiplexing. We elaborate on the two modes and the advantages of statistical INA below, and discuss synchronous INA in Section 2.3.

**INA with Synchronous Multiplexing.** As shown in Figure 1a, a central controller isolates the switch memory into non-overlapping regions and assigns each region to a job. Assume a region starts from address $Offset$ with size $Size$, then a packet with $PSN$ is addressed to

$$aggregator.index \leftarrow PSN\%Size + Offset.$$

An aggregator is not released immediately when the ACK is sent back because some ACKs may be lost, and retransmitted packets need to fetch the result again; synchronous INA usually lets a future packet "one window away" to release a stale aggregator [27, 37]. The switch memory region is reserved for the entire life of a job. Solutions like SwitchML and NetReduce [27, 37] adopt this memory allocation scheme. INAlloc [44] proposes to periodically schedule the switch memory allocation to jobs whose scheduling time granularity is typically a few seconds to a few minutes ($\geq$ 10 seconds). We refer to this mode as *synchronous INA*.

**INA with Statistical Multiplexing.** As is shown in Figure 1b, the switch memory is maintained as a shared pool of aggregators. For a job with ID $JobID$, its packet with $PSN$ is addressed to

$$aggregator.index \leftarrow Hash(JobID, PSN).$$

An aggregator serves jobs' packets in First-Come-First-Serve (FCFS): it is reserved by the first packet observed (with its Job ID) and is released by the ACK packet on the return trip. Statistical INA does not need the aggregator to keep the result for one window because retransmitted packets can fetch results from the PS; an aggregator is allocated and released transiently, with a granularity of an RTT (tens of microseconds). In the case of addressing failure (due to hash collision), the failed packet *falls back* to the PS to complement the aggregation. Each aggregator is reserved and released transiently by packets within one round trip. ATP [22] adopts this memory allocation scheme. We refer to this mode as *statistical INA*.

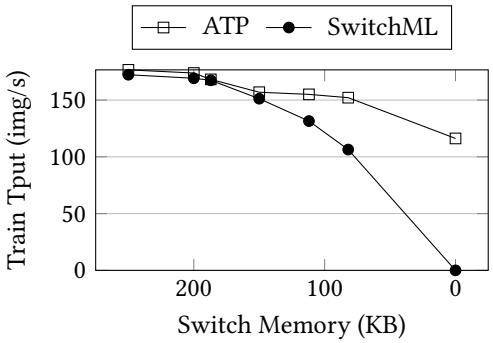



**Figure 2.** Statistical INA's advantage when switch memory is not sufficient, cited from [44]. ATP applies statistical INA, and SwitchML applies synchronous INA.

Statistical INA makes *decentralized resource allocation* to jobs. In a cluster with multiple jobs, each job applies a congestion control mechanism on endpoints (i.e., workers) which adjusts the sending rate with an additive-increase-multiplicative-decrease (AIMD) scheme [2]. The multiple jobs interact in the cluster and contend for network resources, and eventually converge to a *steady state* where each job gets a max-min fair share of the resource.

**Statistical INA's Advantages.** The comparison of synchronous INA and the statistical INA is like that between circuit switching and packet switching for computer networks in the 1970s. Circuit switching applies synchronous time-division multiplexing to reserve link bandwidth, and packet switching applies statistical time-division multiplexing to share

link bandwidth; similarly, the two classes of INA solutions utilize the switch memory differently.

Section 1 briefly states the advantages of statistical INA, and we illustrate them with examples. First, statistical INA improves the switch memory utilization compared with synchronous INA. Figure 2 shows the job performance comparison of the two modes, and the statistical INA shows an equal or higher job throughput (images/second) than the synchronous one. In model training, the job workers iterate to compute and transmit the gradient; in synchronous INA, when the job is in computation, its switch memory is idle without performing aggregation; in statistical INA, an aggregator is transiently reserved and released for one round-trip time (RTT), and thus jobs could multiplex the memory in fine granularity. In addition, when there is no switch memory, statistical INA could fall back traffic to the PS for computation, but synchronous INA would halt.

Second, statistical INA does not incur the integration complexity and potential bottleneck to the DT job manager. Statistical INA does not need a controller to make runtime allocation (and reallocation) of switch memory to jobs; because statistical INA jobs apply congestion control on each endpoint (i.e., workers), and jobs converge to the steady state in a decentralized way. In contrast, synchronous INA requires the central controller to compute the resource share and enforces the allocation; for example, INAlloc [44] has $\sim 2K$ lines of Python code for the remote controller and achieves a minimum 10-second resource scheduling interval.

Third, statistical INA does not need inter-domain operation between the network and the application in cluster management. In many production data centers, different business teams manage the network and the DT application; the network team installs the INA switch program to serve the application team, and the application team does not have access to switch memory for safety reasons. For synchronous INA, even if we can encapsulate switch memory allocation functionality as interfaces for the applications, it is still unwise for applications to decide switch memory usage by themselves, as they are not familiar with the INA performance model (Section 4.1).

### 2.3 Related Work

**Synchronous INA.** INA solutions in synchronous multiplexing mode are in a different scope with NetPack. DAIET [36] first proposes a proof-of-concept in programmable switches without end-to-end system design; SwitchML [37] designs a streaming aggregation protocol to prevent storing entire DNN models in switches; MLfabric [41] discusses the communication pattern optimization of DT jobs with INA. NetRPC [45] provides user-friendly interfaces to overcome the programming difficulty of INA development. There are also architectural design [20] and FPGA [12, 26, 27], RISC-V [9], middlebox [29], and high-performance server [7, 25] based implementations. In the control plane, INAlloc [44] builds a switch memory manager to allocate memory regions to jobs. ASK is asynchronous INA for key-value streams instead of vectors [17]. HIRE [6] targets a broader class of in-network computing applications and applies isolated switch resource allocation among jobs.

Synchronous INA requires the DT application and the network to be in the same administrative domain. It can achieve predictable performance for jobs (e.g., with deadlines) because jobs do not interfere with each other in memory usage; INAlloc [44] makes such a case.

**Statistical INA.** Existing INA solutions in statistical multiplexing mode focus on the data plane protocol design, and NetPack complements them by designing the job placement algorithm. ATP [22] proposes a dynamic, best-effort INA solution to support multi-rack multi-tenant multi-switch scenarios, which could inter-operate with NetPack seamlessly. GRID [10] proposes to co-design INA with routing in a topology to maximize the throughput.

**Job Placement.** Compared with existing solutions, NetPack considers more complete factors, especially the new primitive INA, to achieve multi-job efficiency in job placement. Optimus [32] sorts candidate servers by available GPU resources and evenly distributes workers and parameter servers among the top-k server subset. Tetris [14] uses an alignment score to sort the candidate servers, which is a weighted dot product between the vector of servers' available resources and the tasks' requirements. AFS [18] proposes network packing, a placement mechanism enforcing the GPU usage of each job to be a factor or a multiple of the number of GPUs per server. Network packing puts strict constraints on the scheduling strategy and cluster GPU distribution and requires complete replacement in each scheduling epoch, resulting in large-scale and costly job migration. Harmony [3] utilizes a deep reinforcement learning framework to minimize interference and maximize performance, which, unfortunately, raises the cost of sampling data and retraining for dynamic workloads.

**Job Scheduling.** Job scheduling is about (re)allocating resources to DT jobs in the runtime. The scheduling algorithms could improve synchronous INA but not statistical ones because the latter does not have a central controller to enforce the network resource allocation. NetPack is not in the same scope as the job scheduling solutions. Dorm [39] dynamically partitions the cluster for each job to optimize resource efficiency and fairness. DPS [43], SLAQ [42], and Optimus [32] build performance models to estimate the training speed of each job and dynamically adjust resource allocation to existing jobs. OASiS [4] forms the training job scheduling as an integer linear program to maximize the system throughput. Themis [28] proposes a new resource allocation algorithm to balance efficiency and fairness.

**Job Acceleration.** Prior arts like REEF [8, 16, 33] are geared toward squeezing the most out of GPU performance. NetPack and GPU-oriented optimization complement each other. Higher GPU performance leads to a higher communication

time ratio, where NetPack can play a more significant role in overall performance.

**Water-filling Algorithm.** Network resources are allocated in a decentralized way, and NetPack needs to estimate the resource usage and availability in the network. The family of water-filling algorithms [5, 21, 35, 38] is a well-known approach for estimating flow max-min allocation in a network. NetPack extends this algorithm to two coupled resources by abstracting the switch memory to equivalent throughput, called Peak Aggregation Throughput (PAT, Section 4.1).

## 3 Overall Design

We first introduce the goal and challenges of the job placement problem, and then describe the overall architecture and workflow of NetPack. We elaborate details of the steady-state estimation algorithm and the dynamic programming-based job placement algorithm in NetPack in Section 4 and 5, respectively.

### 3.1 Goal and Challenges

**Goal.** When multiple jobs share a cluster with INA, the switch memory allocation to each job makes a notable impact on the job performance. In statistical INA, each job would be placed at a local region in the cluster and consume multi-dimensional resources such as GPU, bandwidth, and switch memory, and jobs converge to a max-min fair share of network resources; thus, the *job placement* plays a critical role in deciding each job's resource share in the steady state. A good placement plan should try to utilize all-dimensional resources evenly without letting one resource become the bottleneck and another be wasted. This paper aims to build a job placement system that improves overall resource utilization and the consequent job efficiency. We name the proposed system NetPack.

We also make the following assumptions about the application scenario. First, the jobs to place are independent, and can execute concurrently in the cluster. Second, all jobs are benign and would not maliciously occupy network resources. That is, they follow the same network protocol and converge to max-min fairness in resource allocation in a decentralized way. Third, GPUs are allocated to jobs and not preemptable until the job finishes, because GPU context switching is too costly [32].

**Intuition.** NetPack's intuition in job placement is to heuristically score the candidate locations, i.e., servers, in the cluster according to their remaining resources, including computation and network resources, and find the locations with the maximum score (Section 3.2).

We demonstrate the dilemma when weighing the complex factors in performance optimization using the example in Figure 3. The figure shows a Clos network with GPU servers, Top-of-Rack (ToR) switches, and aggregation switches. As jobs start and terminate with time, the available resources become "fragmental": the dark (blue) GPUs are in usage by running jobs and the empty ones are available, $\bar{C}$ represents the remaining available link capacity, and $\bar{A}$ represents the remaining available GPU processing capacity on the switch [2]. Suppose a client submits a new job request that demands 4 GPUs. Various factors can affect the job efficiency. If we consider using neighboring GPUs for the job to reduce traffic volume, we can choose $S1$ and $S3$; if we consider reducing the transmission time, we can choose $S1$ and $S6$, which have the most available bandwidth; if we consider leveraging INA to accelerate AllReduce, we can choose $S4$ and $S6$ as ToR 2 has the most processing capacity. The job placement algorithm should weigh the multiple factors and make a reasonable tradeoff between them. The design of NetPack faces two challenges.

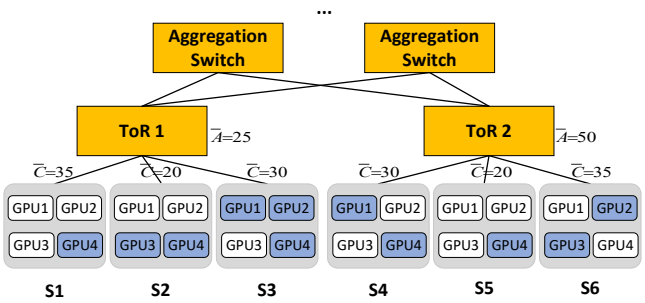



**Figure 3.** Example of Job Placement.

**Challenge 1: NetPack cannot directly observe the real-time network resource availability.** Statistical INA makes decentralized resource allocation to jobs without a central controller and explicit enforcement. Thus, NetPack does not have a way to get the resource availability information in real time in the cluster.

NetPack extends the water-filling algorithm that was originally for bandwidth estimation to two-resource (link bandwidth and switch memory) estimation for INA. The algorithm iteratively allocates a bottleneck resource with max-min fairness among jobs until all resources are allocated (i.e., the steady state). And thus, NetPack estimates the resource usage of existing jobs and acquires the remaining resources for the new job (Section 4).

**Challenge 2: the search space for a job's placement plan is exponential, and NetPack needs to devise an efficient algorithm.** All servers of the cluster are the candidate locations to place a job. In production clusters, after GPUs are allocated to a job, they are not rescheduled or preempted by other jobs due to the GPU context switching overhead [32]. Thus, the search space for a job is exponential to the size of the cluster. Finding the optimal placement plan can be formulated as a Mixed Integer Programming (MIP) problem, which is NP-complete.

[2]The switch processing capacity is defined and quantified as "Peak Aggregation Throughput" (PAT) in Section 4.1

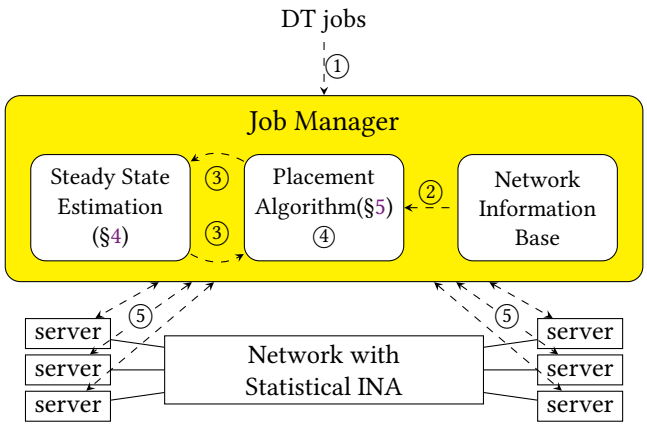



**Figure 4.** NetPack architecture

NetPack heuristically scores each server a value and devises a dynamic programming algorithm to find the candidate location with the maximum value. We observe that the score of a placement plan which involves multiple servers is the sum of the score of individual servers. Thus, a plan's value can be computed recursively by a subproblem on a smaller scale. This property allows us to solve the problem from the bottom-up using dynamic programming with a good estimation (Section 5).

### 3.2 Architecture and Workflow

Figure 4 shows the system architecture of NetPack. The cluster data plane runs the statistical INA service described in Section 2.2. NetPack runs as a cluster-wide DT job manager. Complying with the decentralized network resource allocation, the NetPack job manager does not interact with the network switches for resource allocation. It only interacts with each server by an agent to start/stop DT jobs.

Users submit DT jobs to the manager with the model, the dataset, and the GPU requirements (Step ①). The NetPack job manager batches incoming jobs and periodically places them in the cluster. NetPack does not migrate running jobs because cross-GPU model migration is extremely costly in practice [3, 30].

For each job, NetPack job manager inquires the network information base about the static network configurations, including the topology, link capacity, switch memory, and the existing job placement (Step ②), and runs the steady state estimation algorithm to infer the resource usage by existing jobs and the resource availability for the new job (Step ③, in Section 4).

The job manager further scores a value for each server by its remaining available resources (the server's GPU and access link's bandwidth) and runs a job placement algorithm to find servers with the highest value for the job (Step ④, in Section 5).

NetPack iterates on the batch of jobs with Step ③ and ④ to get each job's placement plan. Finally, the job manager sends the job placement plan to server agents to enforce the job execution (Step ⑤).

When a DT job is accelerated by INA, it has the following runtime properties. We leverage these facts when designing NetPack's abstraction and algorithms. First, after aggregation, the ACK packets on the return path piggyback the aggregation result to workers and release the aggregator, which is a multicast process on the same path as the aggregation process. Due to the symmetry, we can view the topology as an undirected graph and a flow's bandwidth consumption as on undirected links. Second, all workers of a single job stream gradient packets at the same throughput. Each aggregator sends the aggregation result only when all workers' packets arrive, effectively forcing the faster workers to wait for the slower ones. Thus, all workers proceed with the gradient sending at the same time at the same rate.

## 4 Steady State Estimation

We first analyze one job's aggregation model with given resources, and then analyze multi-job resource share in the steady state.

### 4.1 Modeling Single-Job Aggregation

**Problem Definition.** To estimate the steady state of multiple jobs in a cluster, we first model one job's aggregation pattern given a certain amount of resources. A DT job has multiple workers and one or several PSes, and we model one PS and its INA behavior with workers (AllReduce with multiple PSes is composed of multiple one-PS AllReduces). We assume all workers stream gradients at a rate $C$, a switch has a memory of size $M$ for INA, and the round-trip time (RTT) between a worker and the PS is $T$ [3]. The goal of modeling single-job aggregation is to find the portion of aggregated and unaggregated traffic, and the number of flows on each link on the path from workers to the PS.

We define a switch attribute *Peak Aggregation Throughput (PAT)* to describe the maximum aggregation throughput that a switch can support, denoted by the symbol $A$. In a typical transport protocol, the sender delivers a window of packets to the network in one RTT, where the window size equals the *bandwidth-delay product (BDP)*. When the switch processes the traffic, it needs to allocate an aggregator to each packet in the stream. In one RTT, if the switch memory $M$ (unit is one packet) is larger than the window size, the switch can aggregate all packets in the window; otherwise, the switch can only process up to $M$ packets in the window. Thus, a switch can aggregate traffic at a maximum rate of $M/T$, which is its PAT.

[3] RTT $T$ is estimated as the propagation delay plus the switch ECN threshold over the link bandwidth.

**Table 1.** Aggregation model of a switch in an aggregation hierarchy

The switch has $K$ subtrees, each with $n_i$ flows ($1 \le i \le K$).

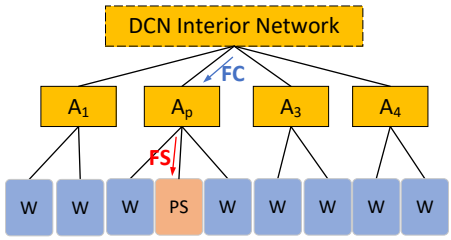
| | $A \ge C$ | $A < C$ |
|---|---|---|
| **Number of Flows** | 1 | $\sum_i n_i$ |
| **Aggregated Traffic** | $C$ | $A$ |
| **Unaggregated Traffic** | 0 | $(C-A) \times \sum_i n_i$ |

**Aggregation Model in a Single Rack.** We first model the trivial case where a job is in a single rack, i.e., all workers and the PS are connected to one switch. If the switch PAT is larger than or equal to the worker's streaming rate ($A \ge C$), all traffic is aggregated and the link from the switch to the PS has only one flow. Otherwise ($A < C$), the aggregated traffic is $A$ and the unaggregated one is $C - A$, and the number of flows on the switch-to-PS link is equal to the number of workers, contributed by the unaggregated traffic.

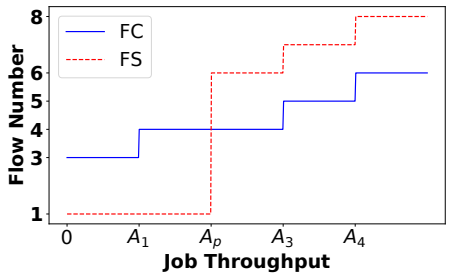
**Aggregation Model in a Hierarchical Topology.** When a job spans multiple racks, the aggregation is hierarchical. We assume INA is provided on the ToR switches because the INA hierarchy is hard to deploy on the multiple paths between ToR switches with ECMP [22]. The worker ToR switches and the PS ToR switch form a two-level aggregation hierarchy, where the PS's ToR switch is the root and the workers' ToR switches are leaves.

The aggregation model of each switch in the hierarchy can be computed bottom-up. At each switch, if the worker sending rate is smaller than the switch PAT ($C < A_i$), all traffic can be aggregated, and the switch outputs a single flow upward along the hierarchy. If the worker sending rate exceeds the switch PAT ($C \ge A_i$), the gradient traffic will be partially aggregated, and the number of upward flows from the switch will become the sum of the flows from all its subtrees, denoted as $\sum_i n_i$. The aggregated traffic is the PAT $A$, and the unaggregated one is $(C - A) \times \sum_i n_i$. The traffic pattern of the hierarchical aggregation model is summarized in Table 1.

**Example.** Figure 5a shows a job in the cluster spanning across multiple racks. This example DCN can be viewed as "one big switch" [1, 23] for estimation (For simplicity, we assume the DCN has full-bisection bandwidth; NetPack algorithms and experiments do not make this assumption). The job spans four racks, two workers per rack. We assume the PAT of four ToR switches $A_1, A_p, A_3, A_4$ satisfy $A_1 < A_p < A_3 < A_4$. $FS$ denotes the number of flows on the link $\text{ToR}^{\text{PS}}$-PS, and $FC$ denotes the total number of flows on the link DCN-$\text{ToR}^{\text{PS}}$ (the sum of all $\text{ToR}^{\text{worker}}$-DCN). We tune the worker sending rate $C$ from 0 to a value larger than $A_4$, and $FS$ and $FC$ would vary accordingly as in Figure 5b. When $C$ exceeds a switch PAT, the number of flows on the switch link towards the PS leaps from 1 to the number of the switch's subtrees' flows. For example, when the sending rate




**(a)** Topology of the example




**(b)** Number of flows varying with the sending rate

**Figure 5.** Hierarchical aggregation model of a single job

is small, all traffic is aggregated: $FC = 3$ and $FS = 1$. When the sending rate exceeds the PAT of a leaf ToR, the switch contributes two flows (two workers in a rack) to $FC$. With the largest sending rate (larger than $A_4$), no switch can fully aggregate all traffic, and $FC = 6$ and $FS = 8$.

### 4.2 Modeling Multi-job Aggregation in a Cluster

**Problem Definition.** Modeling multiple jobs' INA in a cluster is different from modeling one job. Each job does not stream traffic at an arbitrary rate. Instead, the limited network resources (link bandwidth, switch memory) are shared by multiple jobs with max-min fairness. The goal of the multi-job modeling is to find out the eventual converged resource share on each link/switch for each job, which lays the foundation for NetPack job placement in Section 5.

**Algorithm.** There is a class of algorithms called water-filling algorithms [5], which aim to estimate multiple TCP flows' eventual bandwidth sharing in a topology with max-min fairness. The intuition of the water-filling algorithm is to gradually fill in the same traffic volume to each flow like water (complying with max-min fairness), and the physical links would be filled in with traffic volume from its traversing flows as well. The link with more flows would be filled more quickly; if a link is filled, all its traversing flows are frozen without being filled in traffic anymore. The algorithm iteratively fills in traffic and freezes filled links and corresponding flows until all flows are frozen, and then gives an eventual steady state with bandwidth sharing among flows.

The steady-state estimation for NetPack faces a different scenario than existing water-filling algorithms: there are

**Algorithm 1:** WaterFilling Algorithm

```
   Input: Jobs, Links, ToRs
   Output: {l.bw | l ∈ Links}
1  WaterFiling()
2      while |Frozen| < |Jobs| do
3          for j ∈ Jobs\Frozen do UpdateFlows(j.PS) ;
4          for l ∈ Links do Count l's flows in Jobs\Frozen;
5          for r ∈ ToRs do Count r's jobs in Jobs\Frozen;
6          bw1 ← min_{l∈Links and l.bw≠0}(l.bw / l.flows);
7          bw2 ← min_{r∈ToRs and r.PAT≠0}(r.PAT / r.jobs);
8          new_frozen ← Augment(min(bw1, bw2));
9          Frozen ← Frozen ∪ new_frozen;
10 UpdateFlows(job.hierarchy.node)
11     if node is worker then node.flows ← 1, return;
12     for c ∈ node.children do UpdateFlows (c);
13     if node is PS then node.flows ← 0, return;
       /* otherwise, the node is a switch        */
14     if node.PAT > 0 then node.flows ← 1;
15     else node.flow ← Σ_{c∈node.children} c.flows;
16 Augment(bw)
17     frozen ← {}; // record new frozen jobs
18     for j ∈ Jobs\Frozen do
19         for (child, parent) ∈ j.hierarchy do
20             for link l from child to parent do
21                 l.bw ← l.bw − bw × node.flows;
22                 if l.bw = 0 then
23                     frozen ← frozen ∪ {jobs on l}
24         if child is switch and child.PAT > 0 then
25             child.PAT ← child.PAT − bw;
26     return frozen;
```

two kinds of network resources to share among DT jobs — link bandwidth and switch memory. The two kinds of resources are not independent, and thus, cannot be applied with the water-filling algorithm separately. Instead, the two resources are coupled and mutually influence each other: for an INA job, allocating more switch memory to it could aggregate more traffic and reduce bandwidth consumption. Fortunately, the abstraction of PAT allows us to convert the switch memory into *equivalent aggregation throughput* in the statistical INA, which gives us an approach to handle the two coupled resources and allows us to customize the water-filling algorithm.

Algorithm 1 shows the INA-specific water-filling algorithm to compute the steady state for multi-jobs in a cluster. In the algorithm initialization, all jobs are not given traffic volume and are marked as active. The algorithm takes iterations to fill in traffic into active jobs: it computes the minimum share of remaining bandwidth/PAT on each link/switch (line 6-7), augments the minimum share to all active jobs (line 8 and 16-26), and freezes the jobs that are bottlenecked by the link/switch with the minimum share (line 9 and 22-23). At the beginning of each iteration, the number of flows on the links of each aggregation hierarchy is updated (line 3 and 10-15) as the single-job aggregation model in Section 4.1, because the number of flows would be used in later minimum share computation and augment. The algorithm eventually terminates because each iteration at least freezes one link/switch and some jobs, and all jobs are frozen when the algorithm terminates (line 2).

**Table 2.** Notations and their Meanings

| Notation | Meaning |
|---|---|
| $n^{(j)}$ | number of workers for job $j$ |
| $a^{(j)}$ | aggregated throughput of job $j$ |
| $b^{(j)}$ | unaggregated throughput of job $j$ |
| $v^{(j)}$ | total throughput of job $j$ |
| $g^{(j)}$ | GPU requirement of job $j$ |
| $d^{(j)}$ | model size of job $j$ |
| $G$ | available GPU number of severs |
| $C_s$ | link capacity from sever $s$ to its ToR switch |
| $A[r]$ | PAT of ToR switch in rack $r$ |
| $CBW$ | cross-rack bandwidth |
| $x_i^{(j)}$ | job $j$'s workers in server $i$ |
| $y_i^{(j)}$ | job $j$'s PS in server $i$ |
| $z_r^{(j)}$ | whether to enable INA on ToR $r$ for job $j$ |

**Complexity.** In the one-big switch abstraction, there are server links (to ToR) and rack links to DCN, so there are $|Servers| + |ToRs|$ links. As we find at least one bottleneck in each iteration, the estimation algorithm completes within $|ToRs| + (|ToRs| + |Servers|)$ rounds. We can bound the time complexity of one iteration with the total number of flows. Each flow originates from a GPU, and there are at most $|GPUs| \times |Servers|$ flows, where $|GPUs|$ denotes the number of GPUs per server and is constant. Therefore the overall time complexity of the estimation algorithm is $O\left(|Servers|^2\right)$ (assuming $|ToRs| < |Servers|$). Our experiment results in Section 6 indicate that the time overhead of estimation is affordable even in large clusters.

## 5 Job Placement Algorithm

We first model the job placement as a formal Mixed Integer Programming problem, and then give an efficient dynamic programming algorithm to find an approximation.

### 5.1 Mixed Integer Programming Algorithm

Placing a batch of jobs in a shared cluster can be modeled as a *Mixed Integer Programming (MIP)* problem. Table 2 summarizes the notation in MIP, and Table 3 shows the formal definition of the MIP problem. Since the total training time includes initialization time, computation time, and communication time, and the former two factors do not vary with the

**Table 3.** Offline Placement as a Formal Constraint Optimization

$$\text{minimize } \sum_{j\in Jobs}\sum_{i\in Servers} y_i^{(j)} d^{(j)}/v^{(j)}, \text{ s.t.,}$$

$$\sum_{j\in Jobs} w_i^{(j)} \le G, \forall i \in Servers; \quad (1)$$

$$\sum_{i\in Servers} w_i^{(j)} = g^{(j)}, \forall j \in Jobs; \quad (2)$$

$$\sum_{j\in Jobs}\left[x_i^{(j)} v^{(j)} + y_i^{(j)}\left(a^{(j)} + \sum_{k\in S} x_k^{(j)} b^{(j)}\right)\right] \le C_i, \forall i \in Servers; \quad (3)$$

$$\sum_{j\in Jobs} a^{(j)} z_r^{(j)} \le A[r], \forall r \in Racks; \quad (4)$$

$$w_i^{(j)}\left(1 - x_i^{(j)}\right) = 0, \forall i \in Servers, \forall j \in Jobs; \quad (5)$$

$$\left(\sum_{i\in Servers} x_i^{(j)} - 1\right)\left(1 - \sum_{i\in Servers} y_i^{(j)}\right) = 0, \forall j \in Jobs; \quad (6)$$

$$v(j)\left(1 - \sum_{i\in Servers} y_i^{(j)}\right) = 0, \forall j \in Jobs; \quad (7)$$

$$a^{(j)}\left(1 - \sum_{r=0}^{R} z_r^{(j)}\right) = 0, \forall j \in Jobs; \quad (8)$$

$$x_i^{(j)}, y_i^{(j)}, z_r^{(j)} \in \{0, 1\}; \quad (9)$$

$$w_i^{(j)} \in Z^*. \quad (10)$$

placement location; therefore, the objective is to minimize the total communication time. The constraints include: each job's GPU requirement is satisfied (Eq. 1); the total GPU usage in each server is under the limit (Eq. 2); bandwidth/memory usage in each link/switch is under the limit (Eq. 3 and 4); worker placement and GPU usage should be consistent (Eq. 5); there must be at least one PS for multi-worker jobs (Eq. 6); only jobs with a PS can generate network traffic (Eq. 7); only INA-enabled jobs can generate aggregated traffic (Eq. 8); and the integer variable constraints (Eq. 9 and 10).

An MIP problem cannot be solved in polynomial time. By our test, it takes more than four hours to solve the MIP problem above with 100K jobs in a 1K-rack cluster (with Gurobi optimizer [15]). The time complexity is not acceptable, and thus, we turn to an efficient heuristic algorithm.

### 5.2 Efficient Dynamic Programming Algorithm

In the placement algorithm, NetPack first finds a subset of jobs in the batch to place and then takes iterations to place each job in the subset. In each job's placement, NetPack applies heuristics to place workers and the PS separately. Finally, NetPack selectively enables INA for placed jobs. Algorithm 2 summarizes the four-step procedure.

❶ **Find a job subset to place.** The batch of jobs may require more GPUs than currently available ones. Thus, NetPack first chooses a subset of jobs to place in the current period. Various policies can decide the subset, e.g., FIFO, and NetPack formulates a *knapsack* problem.

NetPack gives each job a value according to its importance. The user can specify this value. To avoid starvation, when a job fails to be selected for placement, NetPack increases its value. NetPack regards the cluster as a knapsack with the capacity of $|GPUs|$ and each job as an item with weight $job.GPUs$ and value $job.value$. Thus, the problem becomes finding a subset of jobs with the maximum value and whose total GPU requirements fit in the cluster. We use the common

**Algorithm 2:** Job Placement Algorithm

```
   Input: Jobs
   Output: {job.place | job ∈ Jobs}
1  JobPlacement()
2      JobsToPlace ← FindSubset(Jobs); // Step ❶
3      for job in JobsToPlace in value descending order do
4          if ∃server.GPUs > job.GPUs then
5              job.place ← server;
6              continue;
7          WaterFilling(); // Find Steady State
8          V ←WorkerPlacement(job); // Step ❷
9          job.place ← PSPlacement(job, V); // Step ❸
10     INAEnable(JobsToPlace); // Step ❹
11     Enforce(JobsToPlace);
12 WorkerPlacement(job)
13     W ← job.GPUs; V_{*,*,*} ← 0;
14     for s in [1, ··· , |Servers|] do
15         server ← Servers[s]; w ← server.GPUs;
16         v ← server.bw − (C − server.bw)/(server.flows + 1);
17         for (i,j) in [0, ··· , FS_max] × [0, ··· , W + G] do
18             f ← max(i, server.flows);
19             if V_{(s−1),f_max,j} < V_{(s−1),i,(j−w)} + v then
20                 V_{s,f,j} ← V_{(s−1),i,(j−w)} + v;
21                 V_{s,f,j}.servers ←
                       V_{(s−1),i,(j−w)}.servers ∪ {server};
22             else
23                 V_{s,f,j} ← V_{(s−1),f,j};
24                 V_{s,f,j}.servers ← V_{(s−1),f,j}.servers;
25     return V_{|Servers|,*,*};// candidate plans
```

dynamic programming method [34] to solve the problem and omit it here (line 2).

Note that switch memory and link capacity are not "reserved" by jobs but shared with max-min fairness among them. Placing a new job would change the steady fair share of jobs, so NetPack reruns the water-filling algorithm to estimate the resource share of all running jobs (line 7).

❷ **Place a job's workers.** NetPack needs to find some servers to satisfy the GPU requirement of the job. NetPack preferably places the job on a single server instead of across servers because a single-server placement avoids the overhead of cross-server connections and communication (line 4-6).

If a job has to span across servers, we invoke the steady state estimation algorithm to get the available resources (bandwidth) on each server (line 7). NetPack similarly formulates the server selection problem as a knapsack problem (line 8 and 12-25): the job's GPU requirement as the knapsack and servers as the items. Each server $s$ has a weight $s.GPUs$, and its value is heuristically derived from (1) the server's available bandwidth ($s.\overline{bw}$), and (2) the existing job

throughput loss if the new job were added to the server, i.e.,

$$s.value \leftarrow s.\overline{bw} - \frac{C - s.\overline{bw}}{s.flows + 1}.$$

We use $V_{sfg}$ to denote the maximum value to place a job with $g$ GPUs and a maximum $f$ flows on servers in $[1, s]$, and $V_{sfg}.servers$ to denote the corresponding servers to achieve the value. In this formulation, the knapsack's "weight" is a two-dimensional tuple $(f, g)$. The reason to make the knapsack weight two-dimensional (i.e., recording the highest-value plan for each specific $f$) is to punish "hot-spot" servers in step ❸, making the number of flows distributed more evenly among servers.

We can recursively derive $V_{sfg}$ from $V_{(s-1),*,*}$: if the server $s$ is not chosen,

$$V^{(1)}_{sfg} \leftarrow V_{(s-1),f,g};$$

otherwise,

$$V^{(2)}_{sfg} \leftarrow \max_{\max(i,s.flows)=f,\forall i} \left(V_{(s-1),i,(g-s.GPUs)} + s.value\right).$$

We determine the final $V_{sfg}$ by choosing the maximum one of the above two parts:

$$V_{sfg} \leftarrow \max\left(V^{(1)}_{sfg}, V^{(2)}_{sfg}\right).$$

The recursive algorithm to find maximum $V_{sfg}$ can be solved by a dynamic programming bottom up as described by the `WorkerPlacement()` function in Algorithm 2 (line 12-25).

Note that the algorithm above chooses "all or none" GPUs of a server. Therefore, it may not find a set of servers with exactly the same number of GPUs $job.GPUs$. In this case, NetPack searches placement plans with the total number of GPUs in the range $[jobs.GPU, jobs.GPU + G]$, where $G$ is the number of GPUs per server. If the final plan (after step ❸) allocates more GPUs than the job requirement, NetPack releases the extra GPUs on the least-loaded server. NetPack eventually derives several worker placement plans after considering all servers ($V_{|Server|,*,*}$) and gives them to the PS placement as candidates.

**❸ Place a job's PS.** Each $V_{|Servers|,i,j}(\forall i, j)$ denotes a placement plan for workers. NetPack finds the best PS placement location within each plan and returns the one (worker placement plan and the PS placement) with the highest value.

For each candidate PS location (a server $s$) in one plan, NetPack first updates the flows: if the job's workers are not on the server $s \notin V^{(worker)}_{|Servers|,i,j}.servers$, there is one more flow (from the PS) on the server, denoted as $\epsilon \leftarrow 1$; otherwise, $\epsilon \leftarrow 0$. NetPack also computes the global maximum number of flow in the plan after adding the PS $f_{max} \leftarrow \max(i, s.flow + \epsilon)$.

We evaluate each PS placement location (a server) of one plan based on several factors, including (1) the workers' value $V_{|Servers|,i,j}$, (2) the server's available bandwidth $s.\overline{bw}$, (3) the server's job throughput decrement caused by the PS $\frac{C - s.\overline{bw}}{s.flows+\epsilon+1}$, and (4) a penalty to punish plans with "hotspot" servers $\frac{C}{f_{max}+1}$, i.e.,

$$V_{|Servers|,i,j} + s.\overline{bw} - \frac{C - s.\overline{bw}}{s.flows + \epsilon + 1} - \frac{C}{f_{max}}. \quad (1)$$

By exhausting $i$, $j$, and $s$, NetPack returns a plan with the maximum overall value (by Equation 1).

**❹ Selectively Enable INA.** The steps above enable INA for all jobs, but jobs would benefit differently from INA due to their fan-in degree and job throughput. NetPack further shifts the INA resource to jobs that benefit more from INA. In addition, shifting INA among jobs can tune the tradeoff between bandwidth usage and switch memory utilization as well. To maximize switch memory usage, we favor jobs with a larger fan-in degree where the bandwidth reduction by INA is the most significant.

To capture the preference given to the jobs with a large fan-in degree, NetPack defines a metric "aggregation efficiency" (AE) for jobs. For a job $j$, its aggregation efficiency is the product of the job throughput and the number of flows to aggregate at its programmable switches, i.e.,

$$j.Throughput \times \sum_{s \in j.switches} s.incoming\_flows.$$

NetPack sorts jobs by their AE in descending order and enables INA for these jobs with priority determined by this order, until using up the switch memory. In this way, we allocate INA resources to jobs with a larger fan-in degree.

**In Oversubscribed Networks.** The cross-rack bandwidth may become a new bottleneck in this case. NetPack updates the penalty to be $\max_r\left(\frac{C^{(rack)}}{FC_r+n_r}, \frac{C}{f_{max}+1}\right)$ (in Equation 1, and line 9 in Algorithm 2), where $C^{(rack)}$ is the link bandwidth across racks, $FC_r$ is the existing flow count of a rack $r$, and $n_r$ is the flows the current job introduces. The new penalty prevents the algorithm from placing jobs across multiple racks.

**Complexity.** Step ❶'s complexity is $O(|Jobs| \times |GPUs|)$; Step ❷ and ❸ has $|Jobs|$ rounds and both have complexity of $O(|Servers| \times |FS_{max}| \times |job.GPUs|)$ in each round; Step ❹'s complexity is $O(|Jobs| \times |job.switches|)$. The water-filling algorithm has a complexity of $O\left(|Servers|^2\right)$, and we need to run it for $|Jobs|$ times. Note that $FS_{max}$ is bounded by the number of GPUs per server (a constant); $job.switches$ is bounded by $|ToRs|$ as well as $|Servers|$; and $|Jobs|\times|job.GPUs|$ is the total number of GPU requirements of all jobs and is bounded by $|Servers|$. Thus, the overall complexity is $O\left(|Servers|^2 \times |Jobs|\right)$.

## 6 Evaluation

We evaluate NetPack and show that (1) NetPack can significantly improve multi-job efficiency under various conditions (Section 6.2,6.3); (2) NetPack is efficient to handle job placement in large-scale infrastructures (Section 6.2); and (3) the

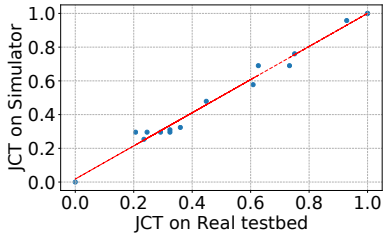



**Figure 6.** Simulator validation, comparing normalized JCT between simulator and testbed.

water-filling algorithm's estimation on multi-job resource fair sharing is consistent with the experimental results on the real testbed (Section 6.4).

### 6.1 Experiment Settings

**Implementation.** We prototype NetPack with a controller as in Figure 4 and agents on both hosts and switches to enforce the placement (job setups/shutdowns). The implementation has 2K lines of Python code and will be open-sourced [31] on publication of this paper.

**Testbed.** We evaluate NetPack on a testbed with five servers, each containing two NVIDIA GeForce RTX 2080Ti GPUs, 56 CPU cores at 2.20 GHz, and 192 GB RAM, and running Ubuntu 18.04. Each host has a Mellanox ConnectX-5 dual-port 100 Gbps NIC. A $32 \times 100$ Gbps Barefoot Tofino programmable switch connects all the servers.

**Simulator.** To evaluate the scalability of NetPack, we also implement a discrete-time flow-level simulator with a tunable number of racks and oversubscription in a fat-tree topology, 16 machines per rack, 4 GPUs per machine. In the following experiments, we set the default number of racks as 16, oversubscription to $1:1$, and available switch PAT to 1Tbps. We use the training speed, iteration, and server configuration information collected from our testbed experiments for the simulation. We validate our simulator on various workloads to match real hardware behavior (see below).

**Baselines.** We compare NetPack with three heuristic algorithms: *GPU-balance* (*GB*, prefers to select servers with more GPU resource), *Flow-balance* (*FB*, prefers to select servers with a smaller number of flows), *least-fragmentation* (*LF*, prefers to use up GPU resource of running servers), and two placement strategies from the prior arts: *Optimus* [32], and *Tetris* [14]. Note that NetPack is the first work with INA considered in the placement; the baselines do not consider INA in placement but have INA (ATP) enabled silently and transparently in the experiment runtime.

**Workloads.** We evaluate six common deep learning models: VGG11, VGG16, VGG19, AlexNet, ResNet50, and ResNet101. These models cover communication-intensive (VGG16) and computation-intensive workloads (ResNet50). Each model trains on the ImageNet dataset. For job traces, we use real-world training logs from Microsoft [19] (labeled `Real` in the following figures). We construct each job's training time and GPU requirement from the logs' record of the start time, end time, and the number of GPUs. As the logs do not contain information about the model type, we randomly pick one from our model pool as prior works do [32]. By default, we use the real-world trace in the experiments without extra explanation, but we also run experiments on two extra synthetic job traces where the jobs' GPU requirements follow a Poisson / normal distribution as a comparison (Figure 7 and 8).

**Metrics.** We quantify how "good" a placement is using two metrics — *average job completion time (JCT)* and *distribution efficiency (DE)*. JCT is widely used to evaluate job management systems. In the evaluation, we normalize each group of JCT by setting JCT from NetPack to 1 to reflect the relative performance of algorithms. We define distribution efficiency as

$$DE = \frac{1}{|Jobs|} \sum_{j \in Jobs} \frac{JCT\_with\_1\_GPU}{Real\_JCT \times No\_of\_GPUs}.$$

Distribution Efficiency can exclude other factors like the model size to influence JCT, only reflecting the effect of placement. If the system linearly scales without any network overhead, the DE is supposed to be one (numerator equals denominator).

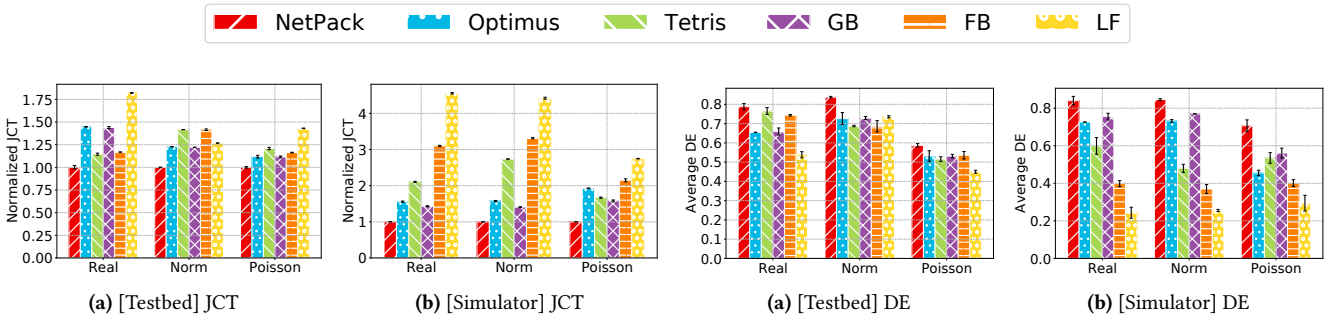
When measuring JCT and DE, each experiment is repeated ten times. The JCT and DE bars and error bars show the average value and standard deviation.

**Simulation Accuracy Validation.** We launch the same set of job traces both on the testbed and in the simulator with the same cluster configuration and plot the normalized JCT in Figure 6. The experiment results show that the testbed average JCT is highly correlated with the simulator estimated JCT in a linear way. Using a linear regression (the solid/red line in Figure 6), we can get a correlation coefficient of 98%, demonstrating that we have an accurate simulator that reflects the real system behavior.

### 6.2 Overall Performance

**Acceleration.** Both Figure 7 and Figure 8 show the overall performance with default settings. NetPack effectively improves both metrics in all experiments compared with baselines, e.g., 13% – 45% JCT reduction and 13% – 46% DE improvement with real-world traces on the testbed. NetPack provides more significant acceleration in a larger cluster (up to 78% JCT reduction and 2.4× DE in simulation) because of the larger optimization space of multi-rack scenarios. Other placement methods without explicitly considering switch memory all experience severe performance degradation.

**Scalability.** We measure the average JCT of a $4K$-job real workload with varying cluster sizes (100 to $10K$ servers, 16 racks in the cluster). Figure 9 shows the experiment results. NetPack provides an average JCT reduction of 31% compared to baselines.




**(a)** [Testbed] JCT **(b)** [Simulator] JCT

**Figure 7.** Average job completion time




**(a)** [Testbed] DE **(b)** [Simulator] DE

**Figure 8.** Average distribution efficiency

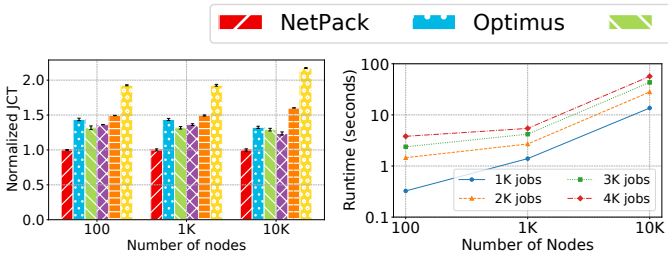



**Figure 9.** [Simulator] JCT, varying cluster scale




**Figure 10.** Placement algorithm execution time

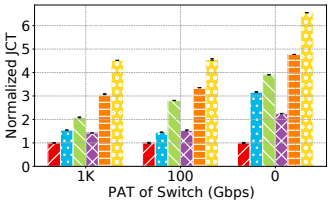



**Figure 11.** [Testbed] JCT, varying switch memory

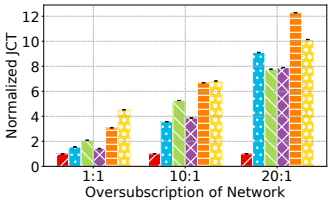



**Figure 12.** [Simulator] JCT, varying bandwidth oversubscription

Figure 10 shows the execution time of NetPack placement algorithm on varying cluster sizes and jobs. For clusters with 100 to $10K$ servers, NetPack can compute the placement of $4K$ jobs within 1 minute. The total placement time grows linearly with the job number for the same cluster, indicating that the per-job placement time keeps constant for the same configuration. The per-job placement time is short and grows linearly with the cluster size: from $3.25 \times 10^{-4}$ second ($10^2$ nodes) to $1.36 \times 10^{-2}$ second ($10^4$ nodes), which is consistent with the theoretical time complexity analysis in Section 5. The algorithm efficiency is sufficient for production clusters. In practice, we only need to place dozens of new-arrived jobs in every scheduling period [19]. Since a job could last for hours, running NetPack for 1 minute to place it is acceptable.

### 6.3 Performance with Limited Resources

**Limited Switch Memory.** Other functions may occupy the switch memory in practice. Thus, we vary the switch memory and show the experiment results in Figure 11. NetPack provides 30% $\sim$ 92% JCT reduction than baselines. Meanwhile, we find that the performance advantage grows with less switch memory. This is because (1) the network becomes more congested with less aggregation; (2) switch memory plays a more significant role in the placement decision-making; (3) NetPack's heuristics also improves the utilization of GPU and network bandwidth, e.g., 87% $\sim$ 92% JCT reduction even without INA ($PAT = 0$).

**Limited Cross-rack Bandwidth.** In the experiments summarized in Figure 12, we tune the cross-rack link bandwidth to observe algorithm performance in an oversubscribed network. NetPack selectively enables INA for jobs using Algorithm 2. Other baselines do not consider INA in placement but enable INA for all jobs and allow them to share the INA resource in the simulation. NetPack outperforms these algorithms, and the performance improvement is more significant with higher oversubscription ratios, e.g., average JCT reduction (than other baselines) increases from 52% with 1 : 1 oversubscription to 89% with 20 : 1 oversubscription. This is because (1) with cross-rack overhead in consideration, NetPack avoids placing a job across racks by applying a higher penalty to cross-rack plans, saving cross-rack bandwidth; (2) NetPack's selective INA-enabling strategy finds a good tradeoff between switch memory and cross-rack bandwidth, and thus maximize the bandwidth reduction.

### 6.4 Validation of Design Choices

**Validation of Multi-Resource Joint Optimization.** All baseline results of GB, FB, and LF indicate that it is insufficient to consider only a single resource for placement, and NetPack indicates that jointly optimizing multiple resources achieves a better performance. To show that the "joint optimization" in NetPack is essential to achieve a performance improvement, we further compare NetPack with a naive combination strategy `Comb`, where resources are separately

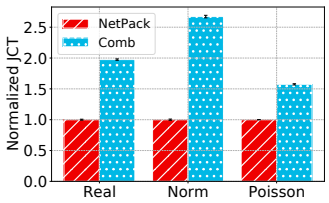
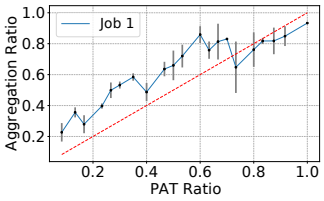
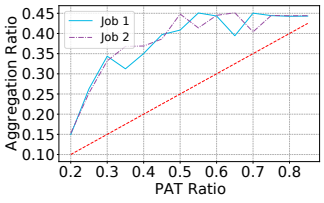



**Figure 13.** [Testbed] Comparison with naive combination




**(a)** Single Job Aggregation




**(b)** Multiple Job Aggregation

**Figure 14.** [Testbed] Job aggregation ratio, varying PAT ratio

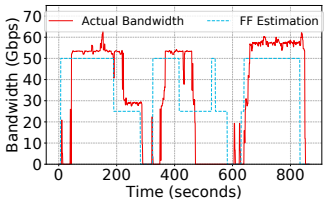



**Figure 15.** Bandwidth measurement v.s. estimation (WF)

considered. `Comb` sorts servers by available GPUs, ToR memory, and link bandwidth in descending order and places jobs to the first matched servers. Figure 13 shows that NetPack outperforms `Comb` by at most 63% JCT reduction on all three workloads, indicating that NetPack balances various new factors well and achieves a good tradeoff.

**Validation of Aggregation Pattern.** Aggregation pattern modeling (Table 1) estimates job flows as a basis of the water-filling algorithm. In the experiments, three servers are in a rack, and the two workers and one PS are on separate servers, respectively. We fix the job throughput at 10 Gbps and adjust the available switch memory for INA. On the $x$-axis, $PATratio = 1$ means the switch memory can support full aggregation $RTT \times throughput$. The $y$-axis displays the portion of switch aggregated throughput over the total job throughput. Theoretically, $y = x$. Figure 14a shows the results, and the real aggregation ratio is close to the predicted aggregation model with only a small deviation.

**Resource Fair Sharing Validation.** Achieving max-min resource fairness is also the basis of the water-filling algorithm (`Augment()` in Algorithm 1). We add one more job with 10 Gbps throughput in the experiment above (the 100% PAT is still for one job). Figure 14b shows that the aggregation ratio of each job grows with the available switch memory. The theoretical model is $y = 0.5x$. The two jobs have a similar aggregation ratio, indicating that they share the switch memory fairly. The actual aggregation throughput can be higher because ML jobs have an iterative "computation-communication" pattern and can take turns to use the switch memory during the training process [22].

**Water-filling Algorithm Accuracy.** We compare the water-filling algorithm estimation result with the monitoring data from the testbed. Figure 15 shows the result. The FF estimation results approximately fit the real testbed bandwidth usage. The testbed traffic has a small delay in taking effect because it takes a while to deliver messages from the control plane to the data plane and to set up servers' jobs.

## 7 Conclusion

Statistical INA brings significant performance gain in DT, but the dynamic network resource allocation nature also challenges the DT job placement in a cluster. We design a system named NetPack for the job placement problem in clusters with statistical INA. NetPack places jobs to the cluster with cluster-wide uniform resource usage in consideration. It overcomes the challenge of resource availability estimation with a water-filling algorithm and that of efficient placement plan searching with a dynamic programming algorithm. We prototype NetPack and conduct experiments to show the performance gain achieved by NetPack compared with the state-of-the-art solutions. In future work, we would explore the joint job placement and scheduling for further performance optimization.

## References

[1] Mohammad Al-Fares, Alexander Loukissas, and Amin Vahdat. 2008. A scalable, commodity data center network architecture. *ACM SIGCOMM computer communication review* 38, 4 (2008), 63–74.

[2] Mohammad Alizadeh, Albert Greenberg, David A Maltz, Jitendra Padhye, Parveen Patel, Balaji Prabhakar, Sudipta Sengupta, and Murari Sridharan. 2010. Data center tcp (dctcp). In *Proceedings of the ACM SIGCOMM 2010 Conference*. 63–74.

[3] Yixin Bao, Yanghua Peng, and Chuan Wu. 2019. Deep learning-based job placement in distributed machine learning clusters. In *IEEE INFOCOM 2019-IEEE conference on computer communications*. IEEE, 505–513.

[4] Yixin Bao, Yanghua Peng, Chuan Wu, and Zongpeng Li. 2018. Online job scheduling in distributed machine learning clusters. In *IEEE INFOCOM 2018-IEEE Conference on Computer Communications*. IEEE, 495–503.

[5] D Bertsekas and R Gallager. 1987. Max-min flow control. *Data Networks* (1987), 448–455.

[6] Marcel Blöcher, Lin Wang, Patrick Eugster, and Max Schmidt. 2021. Switches for HIRE: Resource Scheduling for Data Center in-Network Computing. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems* (Virtual, USA) *(ASPLOS '21)*. Association for Computing Machinery, New York, NY, USA, 268–285. https://doi.org/10.1145/3445814.3446760

[7] Paolo Costa, Austin Donnelly, Antony IT Rowstron, and Greg O'Shea. 2012. Camdoop: Exploiting In-network Aggregation for Big Data Applications.. In *NSDI*, Vol. 12. 3–3.

[8] Daniel Crankshaw, Xin Wang, Guilio Zhou, Michael J Franklin, Joseph E Gonzalez, and Ion Stoica. 2017. Clipper: A {Low-Latency} online prediction serving system. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*. 613–627.

[9] Daniele De Sensi, Salvatore Di Girolamo, Saleh Ashkboos, Shigang Li, and Torsten Hoefler. 2021. Flare: Flexible in-network allreduce. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–16.

[10] Jin Fang, Gongming Zhao, Hongli Xu, Changbo Wu, and Zhuolong Yu. 2023. GRID: Gradient Routing With In-Network Aggregation for Distributed Training. *IEEE/ACM Transactions on Networking* (2023), 1–14. https://doi.org/10.1109/TNET.2023.3244794

[11] Nadeen Gebara, Paolo Costa, and Manya Ghobadi. 2021. PANAMA: In-network Aggregation for Shared Machine Learning Clusters. In *Conference on Machine Learning and Systems (MLSys) 2021*. https://www.microsoft.com/en-us/research/publication/panama-in-network-aggregation-for-shared-machine-learning-clusters/

[12] Nadeen Gebara, Manya Ghobadi, and Paolo Costa. 2021. In-network Aggregation for Shared Machine Learning Clusters. *Proceedings of Machine Learning and Systems* 3 (2021), 829–844.

[13] Richard L Graham, Devendar Bureddy, Pak Lui, Hal Rosenstock, Gilad Shainer, Gil Bloch, Dror Goldenerg, Mike Dubman, Sasha Kotchubievsky, Vladimir Koushnir, et al. 2016. Scalable hierarchical aggregation protocol (SHArP): a hardware architecture for efficient data reduction. In *2016 First International Workshop on Communication Optimizations in HPC (COMHPC)*. IEEE, 1–10.

[14] Robert Grandl, Ganesh Ananthanarayanan, Srikanth Kandula, Sriram Rao, and Aditya Akella. 2014. Multi-resource packing for cluster schedulers. *ACM SIGCOMM Computer Communication Review* 44, 4 (2014), 455–466.

[15] Gurobi. 2022. Gurobi: The Fastest Solver. https://www.gurobi.com/.

[16] Mingcong Han, Hanze Zhang, Rong Chen, and Haibo Chen. 2022. Microsecond-scale preemption for concurrent {GPU-accelerated}{DNN} inferences. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*. 539–558.

[17] Yongchao He, Wenfei Wu, Yanfang Le, Ming Liu, and ChonLam Lao. 2023. 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*. 33–47.

[18] Changho Hwang, Taehyun Kim, Sunghyun Kim, Jinwoo Shin, and KyoungSoo Park. 2021. Elastic Resource Sharing for Distributed Deep Learning. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*. USENIX Association, 721–739. https://www.usenix.org/conference/nsdi21/presentation/hwang

[19] Myeongjae Jeon, Shivaram Venkataraman, Amar Phanishayee, Junjie Qian, Wencong Xiao, and Fan Yang. 2019. Analysis of large-scale multi-tenant GPU clusters for DNN training workloads. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. 947–960.

[20] Benjamin Klenk, Nan Jiang, Greg Thorson, and Larry Dennison. 2020. An in-network architecture for accelerating shared-memory multiprocessor collectives. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 996–1009.

[21] Mari Kobayashi and Giuseppe Caire. 2006. An iterative water-filling algorithm for maximum weighted sum-rate of Gaussian MIMO-BC. *IEEE Journal on Selected Areas in Communications* 24, 8 (2006), 1640–1646.

[22] ChonLam Lao, Yanfang Le, Kshiteej Mahajan, Yixi Chen, Wenfei Wu, Aditya Akella, and Michael Swift. 2021. ATP: In-network Aggregation for Multi-tenant Learning. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*. USENIX Association, 741–761. https://www.usenix.org/conference/nsdi21/presentation/lao

[23] Charles E Leiserson. 1985. Fat-trees: universal networks for hardware-efficient supercomputing. *IEEE transactions on Computers* 100, 10 (1985), 892–901.

[24] Mu Li, David G Andersen, Jun Woo Park, Alexander J Smola, Amr Ahmed, Vanja Josifovski, James Long, Eugene J Shekita, and Bor-Yiing Su. 2014. Scaling distributed machine learning with the parameter server. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*. 583–598.

[25] Mu Li, David G Andersen, Jun Woo Park, Alexander J Smola, Amr Ahmed, Vanja Josifovski, James Long, Eugene J Shekita, and Bor-Yiing Su. 2014. Scaling distributed machine learning with the parameter server. In *11th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 14)*. 583–598.

[26] Youjie Li, Iou-Jen Liu, Yifan Yuan, Deming Chen, Alexander Schwing, and Jian Huang. 2019. Accelerating distributed reinforcement learning with in-switch computing. In *2019 ACM/IEEE 46th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 279–291.

[27] Shuo Liu, Qiaoling Wang, Junyi Zhang, Wenfei Wu, Qinliang Lin, Yao Liu, Meng Xu, Marco Canini, Ray CC Cheung, and Jianfei He. 2023. In-Network Aggregation with Transport Transparency for Distributed Training. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*. 376–391.

[28] Kshiteej Mahajan, Arjun Balasubramanian, Arjun Singhvi, Shivaram Venkataraman, Aditya Akella, Amar Phanishayee, and Shuchi Chawla. 2020. Themis: Fair and Efficient GPU Cluster Scheduling. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*. 289–304.

[29] Luo Mai, Lukas Rupprecht, Abdul Alim, Paolo Costa, Matteo Migliavacca, Peter Pietzuch, and Alexander L Wolf. 2014. Netagg: Using middleboxes for application-specific on-path aggregation in data centres. In *Proceedings of the 10th ACM International on Conference on emerging Networking Experiments and Technologies*. 249–262.

[30] Deepak Narayanan, Aaron Harlap, Amar Phanishayee, Vivek Seshadri, Nikhil R Devanur, Gregory R Ganger, Phillip B Gibbons, and Matei Zaharia. 2019. PipeDream: Generalized pipeline parallelism for DNN training. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*. 1–15.

[31] Netsack. 2021. NetPack Source Code. https://anonymous.4open.science/r/ATP-Controller-35D4.

[32] Yanghua Peng, Yixin Bao, Yangrui Chen, Chuan Wu, and Chuanxiong Guo. 2018. Optimus: an efficient dynamic resource scheduler for deep learning clusters. In *Proceedings of the Thirteenth EuroSys Conference*. 1–14.

[33] Yanghua Peng, Yibo Zhu, Yangrui Chen, Yixin Bao, Bairen Yi, Chang Lan, Chuan Wu, and Chuanxiong Guo. 2019. A generic communication scheduler for distributed dnn training acceleration. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*. 16–29.

[34] David Pisinger. 1995. Algorithms for knapsack problems. (1995).

[35] Qilin Qi, Andrew Minturn, and Yaoqing Yang. 2012. An efficient water-filling algorithm for power allocation in OFDM-based cognitive radio systems. In *2012 International Conference on Systems and Informatics (ICSAI2012)*. IEEE, 2069–2073.

[36] Amedeo Sapio, Ibrahim Abdelaziz, Abdulla Aldilaijan, Marco Canini, and Panos Kalnis. 2017. In-network computation is a dumb idea whose time has come. In *Proceedings of the 16th ACM Workshop on Hot Topics in Networks*. 150–156.

[37] Amedeo Sapio, Marco Canini, Chen-Yu Ho, Jacob Nelson, Panos Kalnis, Changhoon Kim, Arvind Krishnamurthy, Masoud Moshref, Dan RK Ports, and Peter Richtárik. 2019. Scaling distributed machine learning with in-network aggregation. *arXiv preprint arXiv:1903.06701* (2019).

[38] Gesualdo Scutari, Daniel P. Palomar, and Sergio Barbarossa. 2009. The MIMO Iterative Waterfilling Algorithm. *IEEE Transactions on Signal Processing* 57, 5 (2009), 1917–1935. https://doi.org/10.1109/TSP.2009.2013894

[39] Peng Sun, Yonggang Wen, Nguyen Binh Duong Ta, and Shengen Yan. 2017. Towards distributed machine learning in shared clusters: A dynamically-partitioned approach. In *2017 IEEE International Conference on Smart Computing (SMARTCOMP)*. IEEE, 1–6.

[40] Rajeev Thakur, Rolf Rabenseifner, and William Gropp. 2005. Optimization of collective communication operations in MPICH. *The International Journal of High Performance Computing Applications* 19, 1 (2005), 49–66.

[41] Raajay Viswanathan, Arjun Balasubramanian, and Aditya Akella. 2020. Network-accelerated distributed machine learning for multi-tenant settings. In *Proceedings of the 11th ACM Symposium on Cloud Computing*. 447–461.

[42] Haoyu Zhang, Logan Stafman, Andrew Or, and Michael J Freedman. 2017. Slaq: quality-driven scheduling for distributed machine learning. In *Proceedings of the 2017 Symposium on Cloud Computing*. 390–404.

[43] Xueying Zhang, Ruiting Zhou, John CS Lui, and Zongpeng Li. 2020. Dynamic Pricing and Placement for Distributed Machine Learning Jobs. In *2020 6th International Conference on Big Data Computing and Communications (BIGCOM)*. IEEE, 152–160.

[44] Bohan Zhao, Chang Liu, Jianbo Dong, Zheng Cao, Wei Nie, and Wenfei Wu. 2023. Enabling Switch Memory Management for Distributed Training with In-Network Aggregation. In *IEEE INFOCOM 2023-IEEE conference on computer communications*. IEEE.

[45] Bohan Zhao, Wenfei Wu, and Wei Xu. 2022. NetRPC: Enabling In-Network Computation in Remote Procedure Calls. *arXiv preprint arXiv:2212.08362* (2022).