

LEVELLER: Fair Communication Scheduling via Progress-Rate Awareness in Multi-Tenant Training Clusters

Geng Li

China Telecom Cloud Computing Research Institute

Mingyuan Zang

China Telecom Cloud Computing Research Institute

Yang Li

China Telecom Cloud Computing Research Institute
Beijing University of Posts and Telecommunications

Jie Wu

China Telecom Cloud Computing Research Institute
Temple University

Abstract

The rapid expansion of Large Language Models (LLMs) has established Deep Learning Training (DLT) as a major workload in modern multi-tenant GPU clusters. While fair computational scheduling is well-studied, fair communication scheduling in handling execution-time network contention remains underexplored. Current DLT communication schedulers focus primarily on efficiency-oriented objectives, often causing disparate performance degradation or even job starvation when concurrent jobs compete for shared bandwidth. In this paper, we propose LEVELLER, the first communication scheduling system that achieves max-min fairness specifically for DLT workloads. To bridge the gap between traditional flow-level fairness and DLT job-level fairness, LEVELLER introduces a novel online metric, *normalized progress rate*, which quantifies training experience by measuring actual progress against a contention-free ideal. We theoretically prove that a greedy, progress-rate-aware flow scheduling strategy achieves at least a $1/2$ -approximation of max-min fairness for arbitrary workloads, even with lazy priority updates. Building on theoretical results, LEVELLER employs preemptive, dynamic-priority scheduling to achieve near-optimal fairness on commodity infrastructure without hardware or protocol modifications. We prototype LEVELLER on both RDMA and TCP, evaluating it through a physical GPU cluster and trace-based simulations. Experimental results on 10 LLMs show that LEVELLER improves the minimum progress rate by 57% and fairness by 17% compared to state-of-the-art solutions (including CASSINI, CRUX, Symphony and Sincronia), while maintaining competitive cluster efficiency.

CCS Concepts

• **Networks** → **Network control algorithms**; *Network protocol design*; • **Computing methodologies** → *Machine learning*; • **Computer systems organization** → **Cloud computing**.

Keywords

Communication Scheduling, Fairness, Deep Learning, Multi-Tenancy

ACM Reference Format:

Geng Li, Yang Li, Mingyuan Zang, and Jie Wu. 2026. LEVELLER: Fair Communication Scheduling via Progress-Rate Awareness in Multi-Tenant Training Clusters. In *ACM SIGCOMM 2026 Conference (SIGCOMM '26)*, August 17–21, 2026, Denver, CO, USA. ACM, New York, NY, USA, 20 pages. <https://doi.org/10.1145/3789240.3829174>

1 Introduction

In recent years, the rapid advancement of large language models (LLMs) has been reshaping AI industries [25, 27, 31]. From enterprises deploying domain-specific LLMs for business innovation to researchers leveraging open-source foundations for scientific discovery, training custom LLMs has become a widespread demand [40, 78]. Deep learning training (DLT), including fine-tuning and pre-training, has emerged as the dominant workload in modern datacenters, consuming massive amounts of accelerated computing resources (e.g., GPUs) [63, 79]. While a few organizations possess the infrastructure to build private GPU clusters, most users rely on renting cloud-based GPUs to support their training workloads [25, 51]. This trend is driving cloud providers to deploy and operate large-scale, shared, multi-tenant GPU clusters, where multiple DLT jobs from different tenants are running concurrently [10, 24, 49, 51, 77].

Ensuring fairness among DLT jobs is crucial in multi-tenant clusters to maintain cloud integrity [49, 70]. This ensures that equal financial costs result in consistent performance, preventing a “performance lottery” where some tenants effectively pay for the overhead of others [11]. Accordingly, substantial research efforts (e.g., THEMIS [49], Shockwave [88]) have been proposed to investigate fairness in job scheduling, specifically solving how computational resources (e.g., GPUs) are allocated among jobs.

However, computational fairness in job scheduling does not guarantee fairness during execution. A job consists of both computation and communication which may interact in a complex way [39, 76, 79]. When concurrent DLT jobs run, they inevitably contend for bandwidth on shared network links for collective communication, leading to disparate communication time delays. Unfortunately, existing DLT communication schedulers mainly focus on *efficiency-oriented* objectives while de-emphasizing *fairness*; e.g., CASSINI [65] and MCCS [77] optimize job completion time, CRUX maximizes GPU utilization [10]. Consequently, the fairness in DLT communication scheduling remains an open problem.

Fairness for DLT jobs is fundamentally different from flow or coflow fairness, which has been extensively studied in prior work [14,



This work is licensed under a Creative Commons Attribution 4.0 International License. *SIGCOMM '26, Denver, CO, USA*

© 2026 Copyright held by the owner/author(s).

ACM ISBN 979-8-4007-2467-1/26/08

<https://doi.org/10.1145/3789240.3829174>

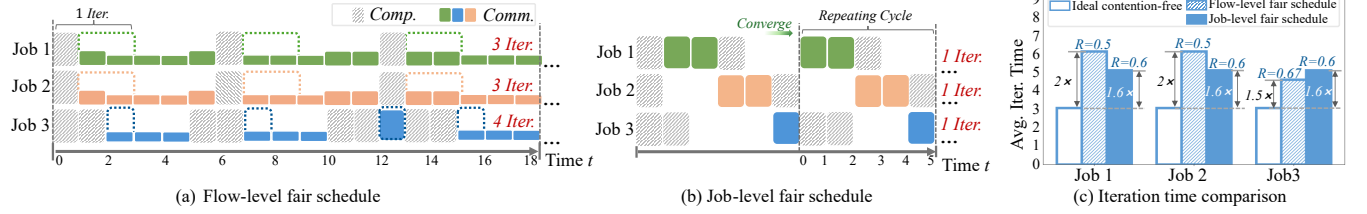


Figure 1: Example of three jobs with identical ideal iteration times of 3 but different communication-to-computation ratios. (a) Flow-level fair schedule converges to a repeating cycle of 18, with average iteration times of 6, 6, and 4.5 for the three jobs, respectively. (b) Job-level fair schedule converges to a repeating cycle of 5, with average iteration times of 5, 5 and 5, respectively. (c) Comparison of three jobs, where the *normalized progress rate*, R , is approximately equal to $1/\text{iteration-time-slowdown}$.

18, 21, 26, 52, 74]. Flow-level fairness equalizes the “network experience” (e.g., bandwidth share [9, 42, 90] or flow/coflow completion time [13, 18, 82]), while DLT job-level fairness should focus on the “training experience” (e.g., training speed), regardless of communication intensity. As shown in Figure 1(a), three jobs with identical iteration times but varying communication-to-computation ratios contend for a single link. Under standard flow-level fair sharing (e.g., DCQCN [90], Swift [42]), bandwidth is equally divided among contention flows. However, this flow-level fair scheduling results in unfair job-level outcomes. By time $t = 18$, Jobs 1 and 2 have completed only 3 iterations, whereas Job 3 has completed 4. As this execution pattern repeats, Job 3 consistently outpaces the others by 33%, leading to a substantial accumulation of unfairness over time.

To provide a unified measure of how communication scheduling impacts different DLT jobs, we introduce a novel online fairness metric, *normalized progress rate*, defined as the ratio of a job’s actual progress to its ideal progress when using dedicated network resources. Inversely proportional to iteration time *slowdown* or *stretch* in classical scheduling [7, 54], this metric captures the relative job-level impact on a unified $[0, 1]$ scale. For example, a 2s per-iteration delay is catastrophic for a 1s-iteration job ($\text{rate}=1/3$), but negligible for a 100s-iteration job ($\text{rate}=100/102$). As shown in Figure 1(c), the respective rates of three jobs at $t = 18$ are 0.5, 0.5 and 0.67, clearly quantifying the progress disparity. Based on this metric, we analyze state-of-the-art schedulers using workload traces and topology data from an in-production cluster [4]. Results reveal significant unfairness, with certain schedulers inducing near starvation at a minimum rate of 0.2.

In this paper, we systematically explore communication scheduling fairness among DLT jobs in multi-tenant clusters, and design a practical system, LEVELLER, that leverages preemptive, dynamic-priority flow scheduling to achieve near optimal fairness. The goal of LEVELLER is max-min fairness, which captures both efficiency and fairness. As shown in Figure 1(b, c), LEVELLER ensures max-min fair scheduling where the *normalized progress rates* of all three jobs converge to 0.6 over the long term. This job-level fair scheduling provides a 20% performance gain for Jobs 1 and 2 at the cost of only a 10% reduction for Job 3, effectively improving the average job performance by 8%. Our contributions are as follows.

Contribution 1. This is the first work to formally investigate the fair communication scheduling specifically for DLT workloads to the best of our knowledge. We characterize its unique challenges through production-trace analysis (§2) and develop a theoretic framework with a novel metric and objective (§3). Starting from a

single-link model, we mathematically prove that a simple yet powerful greedy flow scheduling strategy based on real-time *normalized progress rate* achieves at least a $1/2$ -approximation of max-min fairness for arbitrary workloads and communication intensities (§3.2). More importantly, we extend this theoretical guarantee even under periodic (lazy) priority update (§3.3) and when scaled from a single link to a complex network topology (§3.4). From a practical perspective, these results are significant: the lazy update enables periodic priority adjustment that drastically reduces synchronization overhead, and scaling to general networks allows for a global priority design.

Contribution 2. Building on the theoretical results, we develop LEVELLER, a preemptive, dynamic-priority flow scheduling system for max-min fairness in DLT clusters. LEVELLER employs lightweight global coordination to periodically assign global priorities for DLT flows, effectively offloading the complexities of flow scheduling to existing priority-enabled transport layers and network fabrics. To accommodate the practical constraint of limited priority levels in NICs and switches, LEVELLER introduces a novel graph-based compression algorithm in priority assignment that minimizes fairness loss §4.2. Additionally, LEVELLER incorporates an optimization that exploits communication-computation overlap to further enhance end-to-end performance §4.3. This design makes LEVELLER immediately deployable on commodity infrastructure, requiring no hardware or protocol changes.

Contribution 3. We prototype LEVELLER on both RDMA and TCP, and evaluate its performance on a cluster testbed with 12 Nvidia A100 GPUs. In real-model experiments (e.g., Llama [73], ChatGLM [27], Gemma [72], Deepseek [31]), LEVELLER improves the minimum and average *normalized progress rate* by 57% and 6%, respectively (§6.1). Furthermore, large-scale, production-trace-based simulations involving over 1024 GPUs and thousands of jobs demonstrate that under various cluster topologies and network conditions, LEVELLER improves the minimum rate by 28% to 120× and fairness by 17% to 2.7×, compared to state-of-the-art approaches (CASSINI [65], CRUX [10], Symphony [8]), Sincronia [3], while maintaining high system-wide efficiency (§6.2).

This work does not raise any ethical issues.

2 Background and Motivation

This section presents background and motivation of LEVELLER.

2.1 Inter-Job Network Contention

Deep learning training (DLT). Effective deep learning training is critical to modern AI capability. To reduce training time, or to

meet computational and memory demands, a DLT job is distributed across multiple accelerators using hybrid parallelization strategies (e.g., data [43, 86], pipeline [36, 55] and tensor parallelism [56]). DLT jobs are executed over a vast number of iterations. Within each iteration, inter-GPU communication is performed using collective primitives (e.g., AllReduce, Send/Recv, and Gather/Scatter operations) to synchronize states [38, 50]. Thus, a DLT job is composed of multiple computation and communication phases that repeat periodically throughout the training process [66].

Multi-tenant training clusters. Distributed DLT jobs are trained in GPU clusters [40], each of which is a network of interconnected GPU hosts, each comprising one or more GPUs, CPUs, memory, and storage [40, 64]. Intra-host GPU communication is facilitated by high-speed interconnects such as NVLink [60], whereas inter-host communication relies on RDMA/TCP protocols through NICs and usually Clos-based networks [24, 48]. To achieve better resource utilization and reduced development overheads, GPU clusters are often shared among multiple tenants, training multiple jobs concurrently [39, 49, 64, 77, 79]. Reported from a in-production trace of a 2,000-GPU cluster, the number of concurrent jobs exceeds 30 in peak hours [4].

Inter-job network contention. In multi-tenant clusters, GPUs are typically dedicated to different jobs [39], but the network is shared among them [65]. Due to bandwidth oversubscription [29, 63, 76], imperfect load balancing (e.g., hash collision in ECMP [24]), or link/node failures [44], different jobs may contend on the same link, resulting in inescapable network contention [62]. Analysis of Alibaba traces [4] shows that over 36% of DLT jobs suffer from the impact of inter-job contention (primarily on cross-host network links [10]), leading to training slowdown [8, 65, 66] and reduced GPU utilization [10]. Consequently, inter-job network contention is inherent and inescapable, presenting a persistent challenge for multi-tenant training clusters.

2.2 State-of-the-Art Communication Schedulers

To overcome inter-job network contention, substantial efforts have been proposed, including job scheduling and communication scheduling. Network-aware job schedulers [35, 80, 81, 85, 87] attempt to reduce contention by co-locating jobs within compact locality (e.g., within the same host/rack). However, in production environments, dynamic job arrivals and departures cause resource fragmentation, forcing many models to be trained across distributed GPU hosts [49, 88]. Therefore, even with careful job scheduling, inter-job contention cannot be entirely eliminated [10]. This paper focuses on communication scheduling during training execution, which is orthogonal and complementary to job scheduling.

Existing communication schedulers can be categorized into two orientations: one focusing on general flows/coflows and the other specialized for DLT jobs. General advanced flow/coflow schedulers, such as pFabric[5], PDQ[34], PIAS[6], and Sincronia [3], approximate Shortest Remaining Processing Time (SRPT)-first schedule to minimize flow/coflow completion time. However, these approaches, focusing solely on communication and ignoring computation and iterative natures, are not always optimal for DLT jobs [8, 66].

DLT-specific communication schedulers have recently received significant attention [8, 10, 65, 66]. Muri [87] and CASSINI [65] utilize a centralized scheduler to compute a time shift for each DLT

job's start time to promote interleaving of communication demands. MLTCP [66] approximates this shift through congestion control. CRUX [10] employs strict and static priority scheduling based on GPU intensity to maximize GPU computation utilization. Symphony [8] achieves First-come-first-serve (FCFS) interleaving for large collectives, aiming at reducing the average communication time. MCCC [77] introduces a centralized service-based design that offloads collective communication to the cloud provider, allowing global communication optimization.

2.3 Limitations of Existing Communication Schedulers on Job Fairness

Prior communication schedulers mainly focus on *efficiency-oriented* objectives, e.g., measured by the average performance across all jobs. While efficiency is important, it is perhaps equally, if not more important, to achieve *fairness-oriented* objectives (e.g., assessed by the minimum of all or the total variance) in multi-tenant cluster [49, 70, 88]. An unfair flow schedule can lead to disproportionate treatment, causing some jobs to suffer severe performance degradation or even starvation, while others finish rapidly.

Normalized progress rate as a fairness metric. Fairness is a complex concept spanning computer science, economics, and social justice [23]. While scheduling fairness has been studied in the context of flow/coflow [13, 18, 82], using metrics such as flow/coflow completion time [21] or bandwidth share [9, 42, 90], fairness in DLT remains understudied. DLT workloads are fundamentally different from general flows, as DLT jobs exhibit distinct iterative patterns and varying computation-to-communication ratios [39, 76, 79]. Therefore, flow-level metrics that focus on isolated communication events fail to capture the end-to-end progress of training.

For most DLT jobs, users mainly care about their jobs progressing fast [77, 79]. Therefore, a fairness metric that reflects user experience should be based on progress rate, i.e., the number of iterations completed within a given timeframe. However, raw progress rates are difficult to compare as they depend on the iteration workload (e.g., computation time, communication volume) and available bandwidth. To normalize these values, we introduce *normalized progress rate*, denoted as R , which represents the ratio of a job's actual progress to its ideal progress when provided with dedicated network resources. The intuition behind this metric is straightforward: it quantifies the relative impact of communication delays from a job's perspective by measuring how closely a job approaches its maximum potential rate. Mathematically, the *normalized progress rate* is inversely proportional to iteration time *slowdown* or *stretch* in classical scheduling literature [7, 54]. We prefer the rate-based metric because it serves as an online metric with a unified range of $[0, 1]$. The formal definition will be given in §3.1.

Unfairness in existing schedulers. Based on the *normalized progress rate* metric, we observe significant inter-job unfairness in existing communication schedulers. To demonstrate, we simulate various schedulers using a large-scale production trace from a three-layer Clos network containing over 2,000 GPUs and 5,000 jobs [4]. We extracted 12,213 contention instances (ranging from 2 to 10 jobs) based on the original job characteristics and the network topology. For each instance, we conduct a flow-level simulation, and calculate the distribution statistics of R (including the minimum, maximum, and key quartiles) among contention jobs.

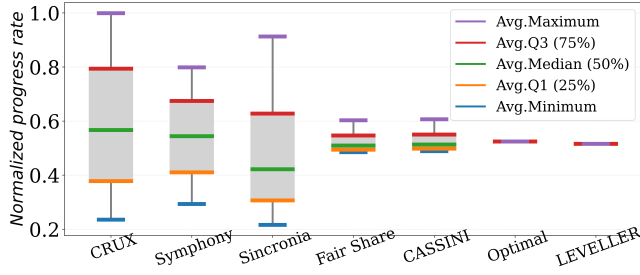


Figure 2: Aggregated box plot of *normalized progress rate* (R) under different schedulers from production traces [4].

Figure 2 shows the aggregated results, demonstrating existing schedulers lead to a significant discrepancy in R among competing jobs. In particular, heuristic schedulers introduce inherent biases: e.g., CRUX and Sincronia (SRPT) favor computation-intensive jobs, whereas Symphony (FCFS) benefits communication-intensive ones. This bias leads to significant unfairness, with some jobs reaching near-starvation levels (e.g., minimum $R \approx 0.2$). More notably, Fair Share and CASSINI, despite ensuring flow-level fairness, fail to maintain job-level fairness (especially when handling three or more jobs). This is primarily because varying communication demands lead to disparate delays among different jobs.

2.4 Our Goal: Max-Min Fairness

To capture both efficiency and fairness, we adopt the max-min fairness in *normalized progress rate* as our objective (formal definition in §3.1). This objective maximizes the worst performance across all jobs, preventing any job from suffering a slow progress rate. Max-min fairness is fundamentally more challenging than basic fairness, which simply aims at equal performance across jobs. Figure 3 illustrates a simple example of two DLT jobs using different schedulers on a single link. As shown in Figure 3(b) and (d), both Fair Share and CASSINI yield fair rates of 0.75 for Job 1 and Job 2, while the max-min fair schedule (optimal) shown in Figure 3(f) can increase rates of both to 0.86, with a Pareto improvement of $\sim 15\%$. **Insight: preemptive, dynamic priority scheduling.** Despite the hardness, we identify the following insights. First, fine-grained, per-flow bandwidth sharing might be unnecessary. Instead, strict priority scheduling effectively exploits the alternating nature of computation and communication, allowing a prioritized job to complete communication rapidly while others perform computation. Second, achieving max-min fairness requires dynamic priorities that change over time. As illustrated in Figure 3(f), the optimal scheduler prioritizes Job 1 at $t = 1$ to preempt Job 2’s communication, and later prioritizes Job 2 at $t = 5$ to block Job 1. These insights motivate the design of a preemptive, dynamic-priority-based flow scheduler to effectively realize max-min fairness.

3 Model and Theory

This section presents the core of LEVELLER— a systematic theoretic framework for max-min fair flow scheduling specific to DLT jobs. We begin by formulating the scheduling problem (§3.1) and then introduce a simple yet powerful rate-aware greedy scheduling strategy, which can be theoretically proved to provide a $1/2$ -approximation guarantee of max-min fairness on a single link (§3.2).

To drive a highly efficient and scalable system design, we extend this fundamental theory in two dimensions: 1) Extension from real-time update to lazy update (§3.3) enables periodic priority adjustment that significantly reduces synchronization overhead, and 2) Extension from a single link to general networks (§3.4) facilitates a global priority assignment design that is compatible with existing transport layers and network fabrics.

3.1 Problem Formulation

Consider a set of DLT jobs $\mathcal{J}$ running in a GPU cluster, each of them assigned to a set of independent and dedicated GPUs, but competing for the network. Each job $j \in \mathcal{J}$ is released at time t_j^0 , and consists of a number of iterations that repeat over time. Each iteration consists of multiple computation and communication phases, which might partially overlap. Each communication phase involves a collective of flows (e.g., AllReduce) with given paths computed from routing and load balancing. Computation times remain constant across different iterations, while communication duration fluctuates depending on network contention.

Fairness metric: *normalized progress rate*. Assuming job j has completed $N_j(t) \gg 1$ iterations by time t , we define the progress of job j as $N_j(t)$, which neglects partial computation or communication that does not constitute a full iteration. Let T_j^{ideal} denote the ideal (minimum) iteration time of job j , defined as the time to complete one iteration when the job is executed alone with full bandwidth on its paths. Therefore, the ideal progress is obtained by $(t - t_j^0)/T_j^{\text{ideal}}$.

DEFINITION 1. *Normalized Progress Rate.* Denoted by $R_j(t)$, the normalized progress rate of job j at time t is defined as the ratio of actual progress over ideal progress, namely

$$R_j(t) = \frac{\text{ActualProgress}(t)}{\text{IdealProgress}(t)} = \frac{N_j(t)}{(t - t_j^0)/T_j^{\text{ideal}}} = \frac{N_j(t)T_j^{\text{ideal}}}{t - t_j^0} \quad (1)$$

As illustrated in Figure 3, the long-term behavior of $R_j(t)$ as $t \rightarrow \infty$ is shown for various scheduling strategies. Intuitively, the metric is bounded in $[0, 1]$, and in the absence of communication delay, $R_j(t) \rightarrow 1$.

Objective. Our objective is to design a flow-scheduling policy $S = \{s(t) | t \in [T_{\text{begin}}, T_{\text{end}}]\}$ that achieves max-min fairness in long-term job *normalized progress rate*. Specifically, we assume that $[T_{\text{begin}}, T_{\text{end}}]$ is a sufficiently long execution window, such that $T_{\text{begin}} \leq t_j^0$ for all j , and no job completes before T_{end} . A scheduling policy is max-min fair if and only if no job’s R can be increased without decreasing the R of another job whose R is no greater than the former. Accordingly, the objective can be formulated as the following lexicographic maximization problem (details in Appendix A):

$$\mathcal{P}_1 : \max_S \text{sort}_{\uparrow} \{R_j(T_{\text{end}})\} | j \in J, \quad (2)$$

where $\text{max sort}_{\uparrow}(\cdot)$ denotes sorting vector in non-decreasing order followed by lexicographic maximization.

3.2 Near-Optimal Schedule on a Single Link

For clarity of presentation, we start with a simple model in which a set of jobs $\mathcal{J}^e$ compete on a single link e , with constant bandwidth

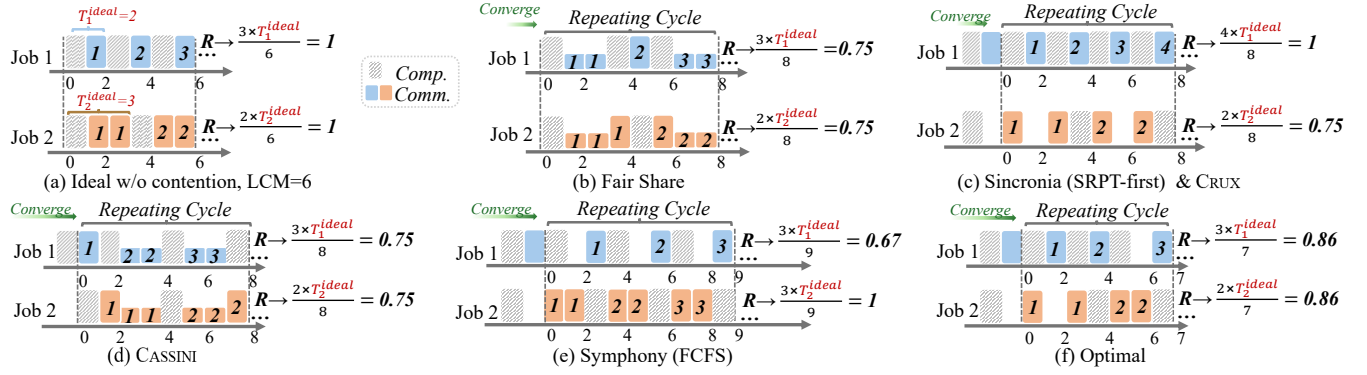


Figure 3: Example of two jobs contending a single link under different schedulers. Each scheduler can converge to a stable cycle that repeats over time. Assuming both jobs involve large number of iterations, then the long-term *normalized progress rate* (R) asymptotically approximates the rate observed within these cycles, which neglects the transient progress made during the initial setup or the final, incomplete cycle. The values for R displayed in each subfigure are computed based on Equation 1.

B_e . For each iteration of job j , we denote the total duration of computation phases by w_j , and the total volume to be transferred on link e in communication phases by vol_j^e . Therefore, $T_j^{\text{ideal}} \leq w_j + vol_j^e/B_e$, where the equality holds when there is no overlap among any computation and communication phases.

Transforming max-min into min makespan. Directly solving the max-min fairness problem on a single link remains a significant challenge due to the multi-objective lexicographic optimization. To address this, we propose a novel transformation of the original max-min objective into a makespan minimization problem.

LEMMA 1. Let $\text{LCM}(\mathcal{J}^e)$ denote the Least Common Multiple of the ideal iteration times $T_j^{\text{ideal}}, \forall j \in \mathcal{J}^e$. Define the normalized iteration number for each job as $\bar{N}_j = \text{LCM}(\mathcal{J}^e)/T_j^{\text{ideal}}$. Under max-min fair scheduling, all jobs achieve a uniform optimal rate R^* , given by

$$R^* = \frac{\text{LCM}(\mathcal{J}^e)}{\min \text{makespan}(\bar{N}_j \mid j \in \mathcal{J}^e)}, \quad (3)$$

where $\min \text{makespan}(\bar{N}_j \mid j \in \mathcal{J}^e)$ denotes the minimum completion time required to complete $\bar{N}_j$ iterations for every job $j \in \mathcal{J}^e$. Then, the max-min fairness problem $\mathcal{P}_1$ can be solved equivalently by repeating the schedule $\bar{S}$ from the makespan minimization problem:

$$\mathcal{P}_2 : \quad \underset{\bar{S}}{\text{minimize}} \quad \text{makespan}(\bar{N}_j \mid j \in \mathcal{J}^e) \quad (4)$$

The formal proof of this lemma can be found in Appendix B. Here, we offer an intuitive explanation based on the example in Figure 3. Given $\text{LCM}(\mathcal{J}^e) = 6$, $\bar{N}_1 = 3$ and $\bar{N}_2 = 2$, the optimal schedule achieves a minimum makespan of 7, while other schedulers result in longer makespans of 8 or 9. Repeating the optimal sequence over time allows both $R_1(t)$ and $R_2(t)$ to converge to $R^* = 6/7 = 0.86$ (Figure 3(f)).

NP-hardness. $\mathcal{P}_2$ can be reduced to a chain scheduling problem on a single machine with time lags in classic scheduling [32, 53, 84]. This problem has been proven to be strongly NP-hard even with equal time lags [32, 33], implying that $\mathcal{P}_1$ and $\mathcal{P}_2$ NP-hard. Therefore, finding an optimal solution is computationally intractable, making an approximation algorithm the most viable approach.

High-level idea. To minimize makespan, each job's communication phases should be scheduled within the computation windows of other concurrent jobs. This interleaving must stay balanced; otherwise, some jobs may run alone at the end, leading to underutilized computation windows and increased idle time. To maintain the balance, a simple greedy strategy is to alternatively prioritize communication with the slowest relative progress rate, such that jobs can be executed "neck and neck" to achieve uniform interleaving. While conceptually simple, this rate-aware greedy scheduling provides a theoretical 1/2-approximation for max-min fairness, and achieves near-optimal performance in practice (§6).

DEFINITION 2. $\sigma(t)$ -Preserving Scheduling. Let $\sigma(t) : [\mathcal{J}^e]$ be a total order over jobs $\mathcal{J}^e$ at time t . A flow scheduling mechanism is said to be $\sigma(t)$ -preserving on link e if, at any time t , a flow belonging to job $\sigma_k(t)$ can be blocked if and only if the link is occupied by a flow from job $\sigma_i(t)$ with $i < k$.

THEOREM 1. Consider a set of jobs $\mathcal{J}^e$ competing for a single link e . Let $\sigma(t) := \arg \text{sort}_{j \in \mathcal{J}^e} (R_j(t))$ be an ordering of jobs in $\mathcal{J}^e$ sorted by non-decreasing R : $R_{\sigma_1}(t) \leq R_{\sigma_2}(t) \leq \dots \leq R_{\sigma_{|\mathcal{J}^e|}}(t)$. Then, any work-conserving, preemptive, and $\sigma(t)$ -preserving scheduling S^σ on link e achieves:

- (1) A 2-approximation of the optimal makespan in $\mathcal{P}_2$;
- (2) A 1/2-approximation of the max-min fair R in $\mathcal{P}_1$. That is, $R_j^\sigma \geq R_j^*/2, \forall j \in \mathcal{J}^e$ where R_j^σ and R_j^* denote the rate obtained under S^σ and the max-min fair scheduling.

The proof of Theorem 1 smartly uses the inequality $\max(a, b) \geq \frac{1}{2}(a+b)$, and the fact that S^σ scheduling keeps jobs "neck-and-neck" toward targets $\bar{N}_j$ (i.e., if Job 1 finishes $\bar{N}_{j_1}$, Job 2 will finish $\bar{N}_{j_2}$). We provide a proof sketch below, and the complete proof can be found in Appendix C.

PROOF. Let ms^* and ms^σ denote the makespan under the optimal and S^σ scheduling. The lower bound of the optimal makespan is constrained by either communication only or computation only, whichever is greater. Specifically, $ms^* \geq \max(C_e^{\text{total}}, W_e^{\text{max}})$, where $C_e^{\text{total}} = \sum_{j=1}^{|\mathcal{J}^e|} \bar{N}_j vol_j^e/B_e$ denoted the total communication time of all jobs, and $W_e^{\text{max}} = \max_{j \in \mathcal{J}^e} \bar{N}_j w_j$ denotes the total computation

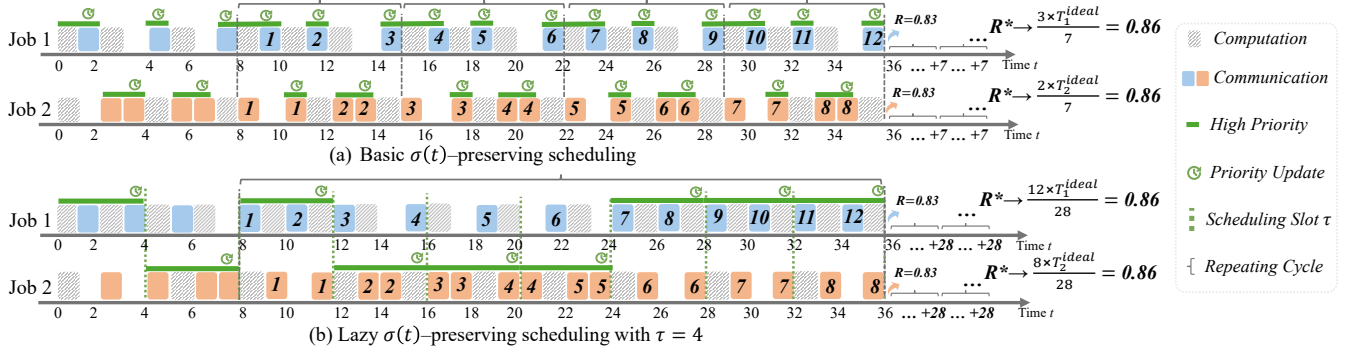


Figure 4: The greedy $\sigma(t)$ -preserving scheduling for the example in Figure 3. While the specific execution patterns differ, the basic and lazy greedy schedules exhibit equivalent progress rates across cycles, both converging to the optimal with $R^* = 0.86$.

time of the job with the longest computation. Under S^σ scheduling, the link only operates in two possible states: busy in communication with a total duration of C_e^{total} , or idle when all jobs are performing computation. The idle duration is upper-bounded by the shortest total computation time, denoted by $W_e^{\min} = \min_{j \in \mathcal{J}^e} \bar{N}_j w_j$. Therefore, the total makespan is bounded by the sum of these two durations: $ms^\sigma \leq C_e^{\text{total}} + W_e^{\min}$.

Consequently, $ms^\sigma / ms^* \leq (C_e^{\text{total}} + W_e^{\min}) / \max(C_e^{\text{total}}, W_e^{\max})$. Since $\max(C_e^{\text{total}}, W_e^{\max}) \geq 1/2(C_e^{\text{total}} + W_e^{\max})$, and $W_e^{\min} \leq W_e^{\max}$, then $ms^\sigma / ms^* \leq 2 \cdot (C_e^{\text{total}} + W_e^{\min}) / (C_e^{\text{total}} + W_e^{\max}) \leq 2$. Based on Lemma 1, $\mathcal{P}_1$ and $\mathcal{P}_2$ are equivalent, therefore $y_j^\sigma \geq y_j^*/2$. $\square$

Example. Figure 4 (a) illustrates the dynamic priority execution under the $\sigma(t)$ -preserving scheduling for the same example in Figure 3. Both jobs are released at time 0, and the priorities are determined by $R_j(t)$: a lower $R_j(t)$ indicates a higher priority (highlighted by green overlines). In the event of a tie, Job 1 is prioritized over Job 2. As shown in Figure 4 (a), from $t = 8$, the communication schedule evolves into a repeating cycle of length 7, which is exactly the same as the optimal cycle in Figure 3(f). A transient phase (i.e., 8) occurs before they converge to the optimal periodic cycle, leading to some loss in optimality (e.g., $R = 0.83$ at time 36). As the schedule progresses, this loss is amortized over the repeated cycles, and both jobs eventually converge to the optimal.

3.3 Extending with Lazy Update

Although the greedy $\sigma(t)$ -preserving scheduling is conceptually simple, its practical implementation can be challenging due to the need for real-time updates on $\sigma(t)$, causing substantial synchronization and update overhead.

To cope with that, we extend the greedy scheduling with a lazy update mechanism. Specifically, time is divided into discrete scheduling slots of length τ . The job ordering $\sigma(t)$ is updated only at the start of each slot and remains fixed throughout the slot. We surprisingly find that as long as $\tau \ll T_{\text{end}}$, this lazy greedy scheduling maintains near-optimal as the basic greedy scheduling, preserving the 1/2-approximation guarantee. The intuition lies in the “self-correcting” nature of the rate-aware strategy: any temporary delay in the current slot due to stale priority can be compensated for by an increased priority in the subsequent slot. In the worst-case scenario, the duration for which a job is incorrectly preempted is bounded by τ . Therefore, at T_{end} , the maximum rate loss is proportional to

τ / T_{end} , which is negligible when $\tau \ll T_{\text{end}}$. We formalize the lazy greedy scheduling in Corollary 1 (proof available in Appendix D).

COROLLARY 1. *The 1/2-approximation guarantee of max-min fairness from Theorem 1 holds when the ordering $\sigma(t)$ is updated periodically every τ units of time, where $\tau \ll T_{\text{end}}$, such that $\sigma(t) := \arg \text{sort}_{j \in \mathcal{J}^e} (R_j(k\tau))$, $\forall t \in [k\tau, (k+1)\tau)$.*

Example of lazy update. Figure 4(b) illustrates the execution under the lazy greedy $\sigma(t)$ -preserving scheduling with $\tau = 4$. While an initial unfairness is observed at the first slot (e.g., $R_1(4) = 1$, $R_2(4) = 0$), the rates of the two jobs rapidly converge in subsequent slots. From $t = 8$, the communication schedule converges into a repeating cycle with a period of 28 time units. Within this cycle, the lazy greedy scheduling achieves optimal max-min fairness: though generates a different execution pattern compared to the basic greedy scheduling, but completes the same progress of $4\bar{N}_1$ iterations for Job 1 and $4\bar{N}_2$ for Job 2. As this cycle repeats, the rates of both jobs can approach the optimal value $R^* = 0.86$.

The extension with lazy update also provides inherent robustness and fault tolerance in update, as any scheduling inaccuracies caused by missed updates are automatically self-corrected during the subsequent slot.

3.4 Extending to General Networks

In large-scale cluster networks, a single job typically involves multiple flows (with potential dependencies), which may traverse multiple links and compete with different jobs on different links.

To extend our single-link theoretical results to a general network topology, we adopt an abstraction approach similar to CRUX [10] by reducing the network to a bottleneck link. Specifically, LEVELLER defines a job’s bottleneck as the link with the minimum local converged rate among all its constituent flows. Here, the local converged rate of a link is the rate to which all traversing flows would converge if restricted to that link alone (detailed definition in Appendix E). Unlike generic bandwidth-based bottlenecks (where a job/coflow may have multiple), a job’s progress rate is strictly governed by this single bottleneck in LEVELLER. The intuition is that because this primary bottleneck restricts the job to the minimum progress rate, background flows on other links necessarily possess higher rates. Consequently, when this job traverses any other link, its lower progress rate automatically translates into the highest

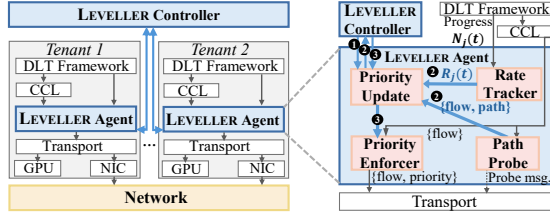


Figure 5: LEVELLER system architecture.

relative priority, ensuring it encounters no secondary bottlenecks elsewhere. This property reduces the network and collective communication complexity to single-link theory. We formalize this result in Corollary 2 and provide a complete proof in Appendix E.

COROLLARY 2. *Consider a set of jobs $\mathcal{J}$ running in a cluster. The greedy $\sigma(t)$ -preserving scheduling on any link in the network, defined by $\sigma(t) := \arg \text{sort}_{j \in \mathcal{J}} (R_j(t))$, and its lazy-update variant, maintain a $1/2$ -approximation of the max-min fair R , $\forall j \in \mathcal{J}$.*

Corollary 2 enables a preemptive, dynamic-priority design: by ensuring appropriate global priority assignments, the complexities of flow scheduling and rate allocation can be offloaded to existing transport and network fabrics that support priority-based scheduling, requiring no further hardware or protocol modifications.

4 System Design

Building upon the previous theoretical results, this section details the design of LEVELLER. §4.1 provides an overview of LEVELLER, §4.2 describes a compression algorithm to compute priority assignment under the practical constraint of limited priority levels, and §4.3 introduces an optimization by exploiting the overlap between communication and computation.

4.1 System Overview

Figure 5 depicts the system architecture of LEVELLER, consisting of a centralized controller and per-tenant agents.

LEVELLER Controller. In order to coordinate network-wide priorities, LEVELLER employs a logically centralized controller that computes global priority assignments for each DLT flow and periodically synchronizes these updates with LEVELLER Agents for enforcement. The specific priority computation algorithm is detailed in §4.2.

LEVELLER Agent. Deployed on each tenant host, the LEVELLER Agent operates as a shim layer between the Collective Communication Library (CCL) layer (e.g., NCCL [59], MPI [1], UCX [69]) and the transport layer (e.g., TCP, RDMA). The primary function of LEVELLER Agents is to assign priorities to DLT flows generated from CCL. Since contention arises primarily within inter-host networks [10], the agent intercepts only inter-host flows while transparently passing intra-host traffic to the underlying transport (e.g., NVLink). LEVELLER is compatible with various of transport mechanisms and network fabrics, as long as strict priority is enabled at both NICs and switches. The LEVELLER Agent consists of the following 4 components:

- **Rate Tracker** computes the job rate by reading the job progress (i.e., iteration counts) from the DLT framework.

- **Path Probe** retrieves flow metadata from CCL and injects the same 5-tuple probe messages to measure flow paths.
- **Priority Update** periodically reports job rates and paths to the controller and receives priority assignments in return.
- **Priority Enforcer** simply sets priorities for inter-host flows based on the latest assignment from Priority Update.

Workflow of priority update. In LEVELLER, DLT flow priorities are updated network-wide through a periodic synchronization process (as depicted in Figure 5): ① LEVELLER controller broadcasts an update request to all agents at each update interval τ . ② Upon receiving the request, the Priority Update component on each agent collects the current job $R_j(t)$ from the Rate Tracker and $\{\text{flow}, \text{path}\}$ mappings from the Path Probe. These metrics, along with the corresponding job IDs, are transmitted back to the controller. ③ LEVELLER controller then computes new priority assignments based on the aggregated state and updates the agents.

The choice of update interval τ involves a trade-off between update overhead and scheduling performance. In particular, a larger τ can reduce the frequency of global synchronization, but it may delay convergence toward optimality, as the worst-case rate loss is proportional to τ/T_{end} . We provide an extensive evaluation of this trade-off in §6.3.

Ideal iteration time T_j^{ideal} profiling and update. To obtain the ideal iteration time T_j^{ideal} , each DLT job, upon arrival, is assigned with a unique highest priority for profiling during the initial iterations. To handle any instability throughout the training process (e.g., path change, GPU slowdown), LEVELLER employs an *online profiling* mechanism to dynamically calibrate T_j^{ideal} . Specifically, LEVELLER schedules re-profiling at semi-stochastic intervals to prevent synchronized noise. Accordingly, T_j^{ideal} is updated using Exponential Moving Average (EMA) or a sliding window to sustained distribution changes. The controller manages the (re)profiling schedule across concurrent jobs.

When T_j^{ideal} changes, the progress rate $R_j(t)$ also needs to be updated to reflect the transition between different ideal conditions. Assuming T_j^{ideal} is updated at t' , we recalculate the rate $R_j(t)$ as:

$$R_j(t) = \frac{N_j(t)}{\frac{t' - t_j^0}{T_j^{\text{ideal1}}} + \frac{t - t'}{T_j^{\text{ideal2}}}}, \quad (5)$$

where T_j^{ideal1} and T_j^{ideal2} denote the ideal iteration times before and after the update, respectively.

4.2 Global Priority Computation

To compute the global priority assignment, a straightforward solution is to assign priorities to flows in ascending order of their job R . However, commodity NICs and switches typically support a limited number of physical priority levels (e.g., 8 [30]), which is often fewer than the number of concurrent jobs (averaging 15–20) [4, 8]. Furthermore, some levels are frequently reserved for dedicated usages [10]. This hardware constraint forces flows with different rates to share the same priorities, potentially violating the $\sigma(t)$ -preserving scheduling and hence undermining max-min fairness. Therefore, priority compression that minimizes the max-min fairness loss is required. **Insight.** Consider an example in Figure 6(a), where Jobs 1–4 comprise five total flows, but must be mapped to only two available

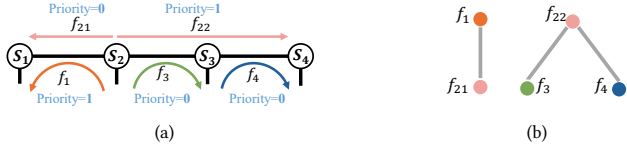


Figure 6: Priority computation insights. Four jobs (ordered by increasing R) mapped to two priority levels, where flows f_1, f_3 , and f_4 belong to Jobs 1, 3, and 4, and $\{f_{21}, f_{22}\}$ belong to Job 2. (a) Optimal priority assignment. (b) Contention graph.

priority levels. Despite this hardware constraint, an optimal priority assignment that maintains $\sigma(t)$ -preserving scheduling is still achievable. This example reveals the following two insights: First, flows belonging to the same job (e.g., f_{21} and f_{22}) can be assigned with different priorities. Second, certain flow groups can have their priorities assigned independently without affecting scheduling performance. For example, the assignment for $\{f_1, f_{21}\}$ can be independent of $\{f_{22}, f_3, f_4\}$, as these two groups do not contend for any resources. However, these priority dependencies are often non-obvious and transitive. For instance, while f_3 and f_4 do not share a link, their priorities must be computed jointly because both flows contend with f_{22} .

Contention graph and connected subgraph. To capture the priority dependencies among flows, we introduce the contention graph data structure. A contention graph is an undirected graph, where each node represents a flow, and an edge connects two nodes if the corresponding flows traverse at least one common link and belong to different jobs. Intra-job contention is excluded from the graph by design, since such contention is either absent (e.g., occurring at non-overlapping times) or resolved by intra-job communication schedulers [47, 50, 68]. A contention graph may consist of several connected subgraphs. As shown in Figure 6(b), if two flows belong to different connected subgraphs, priority assignment on one will not affect the other.

Algorithm. Based on the contention graph, we design an iterative, graph-pruning algorithm: Step 1, for each connected subgraph, assign the highest available priority to the flow with the lowest R ; Step 2, remove these flow nodes and all their neighbor edges from the contention graph to create a new graph; Iteratively repeat Steps 1 and 2 until one priority remains, then assign that priority to all remaining flows.

Figure 7 illustrates an example of the algorithm. While the algorithm cannot guarantee unique priorities for flows on each link, it significantly reduces the number of priority conflicts to a minimal level, and ensures low-rate flows to use non-conflicting high priorities. We evaluate this algorithm with different numbers of priorities in §6.3. The results are highly promising: with just two priority levels, the algorithm achieves less than 1% fairness loss compared to the ideal.

4.3 Scheduling with Comm-Comp Overlap

To accelerate training, modern distributed training frameworks (e.g., Pytorch [43], Megatron-LM [56]) extensively overlap communication with computation [12, 61, 83]. For example, Pytorch’s Distributed Data Parallel (DDP) can overlap gradient All-Reduces with backward computation using the `async-collective-and-wait()` approach [43]. Similarly, Fully Sharded Data Parallel (FSDP) utilizes

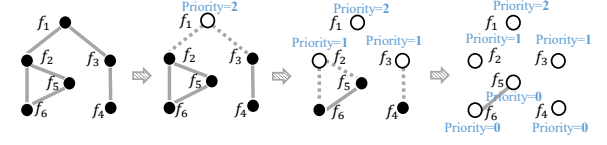


Figure 7: Example of the priority computation algorithm to assign three priority levels to six flows with increasing R .

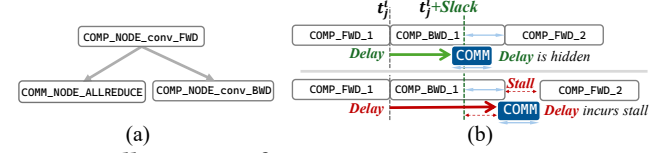


Figure 8: Illustration of communication-computation overlap in scheduling optimization. (a) Training DAG with overlap. (b) Within $[t_j^i, t_j^i + slack]$, communication can be safely delayed, but after $t_j^i + slack$, delay might incur stall.

asynchronous All-Gathers to overlap parameter prefetching with both forward and backward [86]. In a training DAG, this overlap is represented by parallel branches as shown in Figure 8(a).

Overlap-aware optimization can further improve max-min fairness, as communication delays hidden by computation do not stall the training progress [10]. In the greedy scheduling, the job with the lowest R is granted the highest priority. However, our key insight is that if this job’s communication is overlapped with its computation, its high priority is not necessary. As shown in Figure 8(b), let t_j^i be the start time of an overlapped communication phase for the i -th iteration of job j , and let *slack* denote the maximum duration for which this communication can be delayed before blocking the next operation. Within this slack window $[t_j^i, t_j^i + slack]$, even if job j has the lowest R , its priority can be safely downgraded to the lowest level without delaying its overall progress. This allows other jobs to improve their own R . Once this window expires, any further delay might stall the progress just like non-overlapped jobs.

Priority correction. To leverage this, we introduce a dynamic correction to the centrally computed priorities $Priority_j(t)$ from the controller. This correction can be performed locally at each agent, requiring no global coordination. Specifically, for each overlapped communication flow, Priority Enforcer at LEVELLER agent assigns the lowest priority (e.g., Level 0) within its slack window. If the communication persists beyond the window, its priority is immediately restored to the original value $Priority_j(t)$ to prevent rate loss. The corrected priority, $Priority'_j(t)$, is computed as follows:

$$Priority'_j(t) = \begin{cases} \text{The lowest} & t_j^i \leq t < t_j^i + slack \\ Priority_j(t) & t \geq t_j^i + slack \end{cases} \quad \forall \text{iteration } i. \quad (6)$$

By exploiting the overlap of the lowest-rate jobs to speed up others, this optimization can further improve global max-min fairness, specifically enhancing the average and maximum rates. We evaluate this optimization in §6.3.

5 Implementation

We implement LEVELLER using 2,000+ lines of C++ and Python code, prototyping it on both RDMA and TCP. The current implementation is on PyTorch [43, 86] and NCCL [59], while the design is generalizable to other frameworks and CCLs.

LEVELLER agent is implemented as a resident daemon within each tenant instance, serving as a user-space shim layer between the CCL and the underlying transport layer. Using library interposition via LD_PRELOAD, the agent transparently intercepts underlying communication calls and injects control logic, without modifying user training code. In particular, the Rate Tracker extracts job progress by parsing framework’s standard output logs. The Path Probe intercepts NCCL connection primitives (such as `ncclIbRtrQp`) to embed consistent 5-tuple identifiers into INT [41] probe packets at the C++ shim layer. The Priority Update maintains a persistent channel with a centralized lightweight controller service.

Dynamic priority on TCP. For TCP traffic, Priority Enforcer dynamically manages mangle table rules through Linux Netfilter. Using iptables hooks, it identifies flows based on 5-tuple signatures to remark DSCP values, effectively mapping specific traffic to different hardware queues.

Dynamic priority on RDMA. For RoCE/InfiniBand traffic, where each Reliable Connection Queue Pair (QP) maps to a unique point-to-point flow, the Enforcer maintains a $\{Flow \rightarrow QP\}$ mapping to enable granular control. We implement a QP resurrection mechanism to bypass Verbs API restrictions on modifying the Traffic Class (TC) during the Ready-to-Send (RTS) state. Specifically, when priority adjustment is required for a specific flow, the Enforcer triggers a state machine transition on the corresponding QP, involving the following three atomic steps: 1) a Drain Check to monitor Send Queue (SQ) and to ensure all in-flight packets are acknowledged, 2) a Force Reset to switch the QP state to RESET via `ibv_modify_qp`, and 3) a State Reconstruction (INIT→RTR→RTS) where the new TC is injected into the global route header. This ensures seamless priority switching without packet loss, reordering, or application-level changes.

6 Evaluation

To comprehensively evaluate LEVELLER, we conduct experiments on a 12-GPU physical testbed and a customized flow-level simulator based on production traces. The testbed demonstrates the practicality of LEVELLER in realistic DLT pipelines, while the simulator enables scalable evaluation in large-scale settings. We compare LEVELLER against representative state-of-the-art schedulers, including CASSINI [65], CRUX (w/o routing) [10], Symphony [8], and Sincronia [3], as well as widely adopted Fair Share mechanisms (e.g., DCQCN [90]). Due to space constraints, Appendix F provides further details on topologies, workloads, and additional results.

6.1 Real-World Evaluation

Testbed setup. Our testbed comprises three hosts, each with four NVIDIA A100 GPUs and four Mellanox ConnectX-6 NICs. We construct five logical switches over two physical switches to emulate a 3-leaf, 2-spine topology. Each NIC connects to a leaf, and all inter-switch links are rate-limited to 50 Gbps. ECMP and DCQCN are used by default. We implement CASSINI and CRUX on our testbed for comparison.

Testbed DLT workloads. We selected four representative LLMs as training workloads, including Llama-3.1 (4B), ChatGLM-3 (6B), Gemma (2B), and DeepSeek-Coder (1.3B). The first two are trained

with FSDP [86], while the latter two use DDP [43]. These models provide a wide range of model sizes and diverse communication-computation patterns for evaluating heterogeneous DLT workloads.

Case study on RDMA. To highlight LEVELLER’s advantages, we create an RDMA-based inter-host contention scenario on our testbed. We deploy ChatGLM, Gemma, and DeepSeek (two GPUs each) sharing a single bottleneck link for collective communication. We use two priority levels and visualize bandwidth usage for all jobs within the same time window.

Figure 9 shows bandwidth usage and $R_j(t)$ under different schedulers. The figure reveals clear distinctions among them: CASSINI follows a fair share bandwidth allocation, CRUX applies static priorities, and LEVELLER exhibits dynamic prioritization. Within the same interval, LEVELLER completes 14, 27, and 14 iterations for the three jobs, compared to 13, 26, and 15 for CASSINI, and 12, 23, and 17 for CRUX. LEVELLER achieves a minimum progress rate of 0.69, improving on CASSINI (0.63) and CRUX (0.61) by 9.4% and 13.9%, respectively. This demonstrates that LEVELLER enhances max-min fairness through progress-rate-aware scheduling.

Different tenant numbers on TCP and RDMA. To validate the applicability of LEVELLER’s preemptive, dynamic-priority scheduling across different priority-enabled transport layers, we evaluate LEVELLER under both RDMA and TCP with varying network contention by scaling the number of tenants. In each run, every tenant is allocated 2–4 GPUs, and job placement is randomized. Each experiment is repeated at least ten times, with results averaged to mitigate runtime variability.

As shown in Figure 10, on both RDMA and TCP, LEVELLER consistently outperforms CASSINI in terms of the minimum and average progress rates across jobs. In the four-tenant setting, LEVELLER improves the minimum and average rates by 21% and 6% on RDMA, and by 17% and 4% on TCP, respectively. Compared with CRUX, LEVELLER improves the minimum progress rate by up to 57% (3 jobs with RDMA) at a cost of up to 15% decrease in average (4 jobs with RDMA), demonstrating LEVELLER substantially improves fairness with a minimal impact on overall efficiency. Despite higher actual and ideal progress rates (i.e., faster training), RDMA shows lower normalized rates than TCP because its superior inter-host bandwidth leads to higher contention.

LEVELLER overhead. We evaluate LEVELLER’s runtime overhead in a four-tenant scenario. Testbed measurements show negligible Controller-Agent communication latency ($< 200 \mu s$). Priority-switching overhead is only a few milliseconds, and as switching occurs at a minute-level cadence, its amortized cost is also negligible. Furthermore, the Controller-Agent communication and priority switching overhead does not increase with cluster scale, confirming the practicality of LEVELLER for large-scale deployments.

6.2 Trace-Based Simulation

Simulation setup. Our flow-level simulations model a cluster of 1,024 GPUs across 128 racks using two topologies: (1) Oversubscribed (8:1): a three-tier Clos architecture with 128 ToR, 16 leaf, and 8 spine switches. (2) Non-blocking (1:1): two separate planes, each with 64 racks and 8 leaf switches, with each leaf switch connected to 64 GPUs. Both use a leaf-spine structure with 100 Gbps

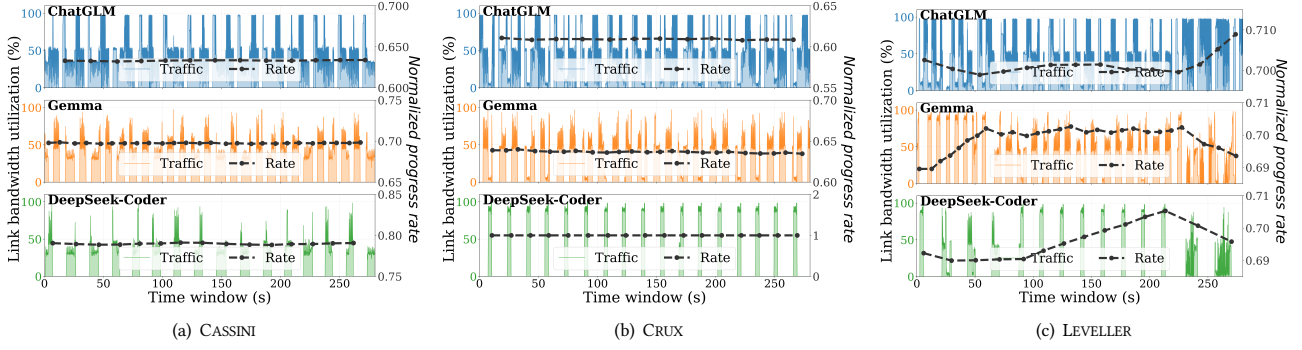


Figure 9: Real-time link utilization and $normalized\ progress\ rate\ R_j(t)$ under different schedulers on the RDMA testbed.

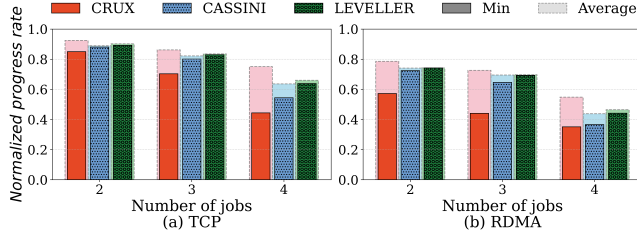


Figure 10: Minimum and average $normalized\ progress\ rates$ with different tenant numbers on TCP and RDMA testbeds.

links and ECMP load balancing. Results are averaged over 1,000 independent runs.

Simulation DLT workloads. To capture workload diversity in multi-tenant DLT clusters, we use the STG Generator [2] to synthesize traces in Chakra (CET) format [71]. Each CET trace is then fed into ASTRA-sim [75] to model the per-iteration behavior of each job in isolation. We evaluate ten models—GPT-2 (Small, Medium, Large, and XL), GPT-3 (6.7B, 13B), LLaMA-1 (7B, 13B), LLaMA-2 (7B), and Mistral (7B)—across two parallel strategies (DP, FSDP) and five GPU scales (8–128). In addition to synthetic traces, we also incorporate empirical workloads from [4].

Performance comparison. We first evaluate the performance of all schedulers under varying tenant scales and topologies. As shown in Figure 11(a), progress rates decline as the tenant scale increases for all schedulers. However, LEVELLER consistently achieves the highest minimum progress rate. Compared to CASSINI, the strongest baseline, LEVELLER improves this metric by 14%–28% and 6%–14% under the two topologies. LEVELLER maintains efficiency comparable to most schedulers, with the exception of CRUX and Sincronia. Particularly, LEVELLER trades a minor decrease in average rate (3%–24%) to eliminate the starvation seen in both, improving the minimum progress rate by up to 120%.

Figure 11(b) shows Jain’s Fairness Index [37] across varying tenant scales. LEVELLER consistently maintains close to 1, whereas other schedulers degrade as tenant scale grows. LEVELLER improves fairness by up to 17% over Fair Share and CASSINI, and up to 2.7× over CRUX and Sincronia.

Case study. To explain these gains, we present a detailed case study. Figure 12 shows the real-time rates $R_j(t)$ of two jobs with different sizes and communication-to-computation ratios: GPT-2 Medium ($\approx 1:1$) and GPT-2 Small ($\approx 2:1$). Results reveal that CRUX

favors computation-intensive jobs, while Sincronia prioritizes those with smaller communication volumes. Both approaches result in significant unfairness, with one job’s rate close to 1 while the other drops as low as 0.6. CASSINI and Fair Share adopts flow-level fairness and achieves basic job fairness with both job rates reaching 0.75. However, LEVELLER, targeting at max-min fairness, increases both to 0.79, with a Pareto improvement of 5%.

Job completion time (per-iteration). In addition to the $normalized\ progress\ rate$, we evaluate the traditional Job Completion Time (JCT) metric across different schedulers. Because total training time varies heavily with the number of iterations or target accuracy, we use the average per-iteration time to represent JCT. As illustrated in Figure 13, higher tenant numbers or bandwidth oversubscription lead to higher network contention, increasing average and tail (95th-percentile) JCT for all schedulers. LEVELLER consistently achieves the lowest JCT, particularly at the tail. Compared to alternative schedulers, LEVELLER reduces tail JCT and average JCT by up to 17.8% and 1.4%, respectively. These results confirm that improving the minimum progress rate can translate to a reduction in tail JCT, further validating the effectiveness of our approach.

6.3 LEVELLER Deep Dive

Impact of update frequency. The update interval τ represents a trade-off between overhead and performance: a larger τ reduces synchronization frequency, but may delay convergence towards optimality. Figure 14 illustrates the convergence behavior of LEVELLER under different frequencies. Higher job contention naturally lead to lower progress rates and slower convergence. In practice, $\tau = 200s$ typically achieves convergence within 20 minutes across both low and high contention scenarios, resulting in negligible performance degradation in production-scale environments.

Impact of available priority levels. Figure 15 shows the performance loss of LEVELLER across different numbers of priority levels, measured relative to an ideal eight-priority configuration. Our priority compression algorithm designed to minimize max-min fairness loss achieves a highly promising result: even with an extremely infrequent update ($\tau = 2000$) and high contention (tenant number 30), our algorithm achieves less than 1% fairness loss with just two priority levels, and with three or more levels, it achieves near-zero loss in large-scale training scenarios.

Overlap-aware scheduling. To validate the optimization with awareness of communication-computation overlap, we synthesize

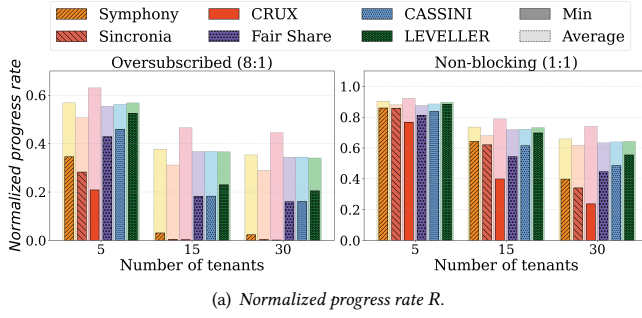


Figure 11: Impact of tenant scale and network topology on *normalized progress rate* and *Jain's Fairness Index*.

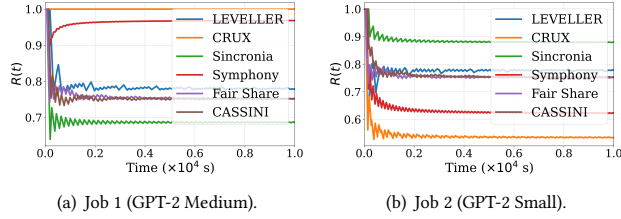


Figure 12: Case study of two jobs sharing a network link.

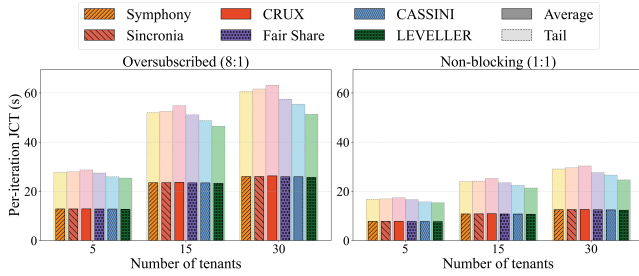


Figure 13: Comparison on average and tail JCT.

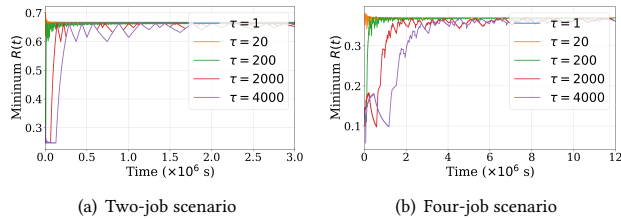


Figure 14: Impact of update frequency on convergence.

a set of jobs with pronounced overlap. By strategically exploiting the overlap of lowest-rate jobs to accelerate others, this optimization improves global max-min fairness and overall performance. As shown in Figure 16, overlap-aware scheduling enhances the average and maximum rates by 4.1% and 2.4%, but its impact on the minimum progress rate is modest. This is because the optimization primarily enables non-minimum-rate jobs to improve their rates without compromising the progress of the minimum-rate ones.

6.4 Integration with Job Schedulers

Communication and job schedulers are orthogonal yet complementary. Figure 17 evaluates the performance of different communication schedulers working with and without HiveD [85] a job scheduler, that allocates GPUs based on physical affinity to

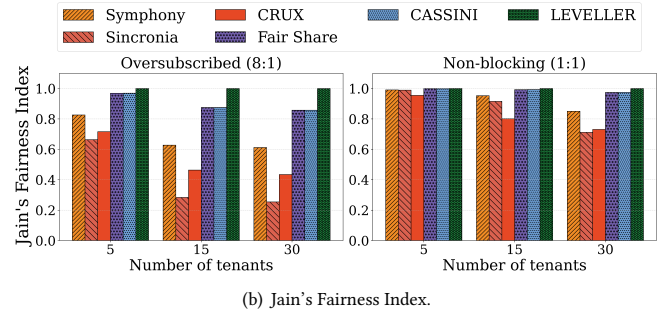


Figure 15: Impact of available priority levels.

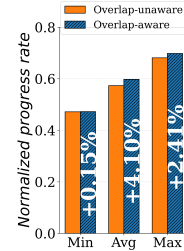


Figure 16: Overlap-aware scheduling.

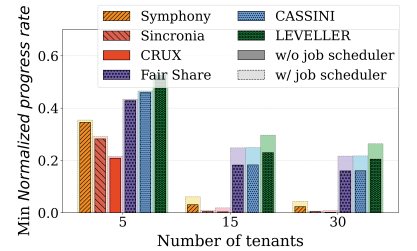


Figure 17: Integration with job scheduler HiveD [85].

reduce communication overhead and contention. The integration with HiveD improves the minimum rates by 3% to 29% across all communication schedulers. However, even with HiveD, LEVELLER outperforms CASSINI by an additional 9% to 21%. This proves that significant communication contention persists despite advanced job scheduling, leaving substantial room for LEVELLER to improve overall performance in both fairness and efficiency.

7 Discussion and Limitations

Why *normalized progress rate* is the right metric? Existing job-level performance metrics like completion time, GPU utilization or idle time fail to isolate the communication scheduling fairness because they are tied to tenant-side variables like workloads and algorithms. Instead, *normalized progress rate* provides a tenant-agnostic abstraction that decouples network contention from application-specific factors to quantify scheduler's direct impact on training performance. We advocate for the networking community to adopt this metric when evaluating future flow scheduling, congestion control, and traffic engineering research.

Efficiency and fairness trade-off. When discussing fairness, the primary concern is usually the potential compromise in efficiency, as improving the minimum rate often necessitates a reduction elsewhere. Our experiments confirm this trade-off: for highly efficient

but unfair schedulers such as CRUX, our approach significantly enhances fairness at the cost of some efficiency (up to 24%). However, compared to most other methods, our max-min fairness objective can improve fairness with minimal efficiency loss, or even improve both, demonstrating that max-min fairness does not necessitate a significant “tax” on cluster efficiency. Moving forward, we intend to explore this Pareto frontier by integrating more sophisticated scheduling heuristics, such as deadline-awareness or weighted fairness for tiered service levels.

Dynamic job arrival and departure. LEVELLER is an online scheduler for dynamic workloads. When a new job arrives, small iteration counts after profiling may briefly cause artificially high rates and low priority compared to long-running jobs that have accumulated more iterations and delays. But it can quickly stabilize and converges to a fair rate as the job progresses. Job departures require no special handling, as the priority assignment can be simply recomputed in the next scheduling slot.

Routing and load balancing. LEVELLER operates independently of underlying network routing protocols. Notably, rather than directly reducing network contention itself, LEVELLER achieves a more fair (and potentially more efficient) schedule that maximizes the performance of the lowest-rate jobs under any given level of contention. Therefore, LEVELLER is complementary to existing load balancing mechanisms, and focuses on fairness optimization. For single-path dynamic routing (e.g., adaptive routing [57] or Fast Reroute [16] for failures), LEVELLER detects path changes and immediately triggers its profiling process to adjust progress rate calculation for the new path. For multi-path routing (e.g., packet spraying [17]), LEVELLER profiles T^{ideal} with highest priority across all paths and dynamically updates execution priorities along each path. We provide an evaluation with packet spraying in Appendix G, confirming that the fairness improvements of LEVELLER persist in multi-path environments. Furthermore, similar to CRUX [10], LEVELLER can integrate with routing by proactively steering low-rate jobs toward uncongested paths to further improve fairness and efficiency.

Handling different parallelisms. LEVELLER focuses on inter-host links, where Data Parallelism (DP), Pipeline Parallelism (PP), and Expert Parallelism (EP) operate [24, 45]. LEVELLER performs effectively with DP and PP, as T_j^{ideal} remains nearly constant, subject to small system noise, naturally handled by our rate-based scheduling. In EP [22, 46], although communication and computation time varies across iterations, these variations are not entirely random. Instead, they typically follow a specific distribution driven by expert load skews. Our *Online Profiling* in §4.1 can effectively handle EP-specific distribution.

Robustness to non-network bottlenecks. For periodic workloads (excluding EP), the ideal iteration time T^{ideal} remains highly stable [65, 79], which is also confirmed in the evaluation. However, in rare instances, iteration durations can be impacted by non-network bottlenecks, such as slow GPUs or local I/O contention. LEVELLER detects persistent non-network bottlenecks via *Online Profiling* or when a higher priority fails to increase the progress rate. Once detected, LEVELLER re-calibrates T^{ideal} , and its self-correcting design naturally compensates for any prior resource over-allocation.

8 Related Work

Fairness in flow/coflow scheduling. Fairness is an essential criterion for communication scheduling in contending flows [9, 21, 52]. Traditional congestion control often prioritizes flow rate fairness, aiming to equalize bandwidth at bottlenecks [9, 42, 90]. To better align with user experience, advanced schedulers use Flow Completion Time (FCT) [21] or slowdown (normalized FCT) [82] as more representative fairness indicators. Beyond individual flows, Coflow scheduling introduces a higher-level abstraction by considering a collection of flows as a single semantic unit [3, 15]. Similarly, Coflow fairness has been disused [13, 18], often based on notions of minimum resource allocation [14, 74] or Dominant Resource Fairness (DRF) [26]. However, as detailed in Section 2, the fairness of DLT jobs are fundamentally distinct from individual flows and Coflows, leaving DLT fairness an open challenge.

Inter-job communication scheduling. Inter-job network contention in multi-tenant clusters has catalyzed the development of DLT communication schedulers. Muri [87] and CASSINI [65] use “shift” strategies to interleave jobs, and MLTCP [66] approximates this via congestion control. CRUX [10] maximizes GPU utilization through intensity-based flow scheduling, while Symphony [8] and Harmonics [67] focus on minimizing average communication time via FCFS scheduling and rate assignment. MCCS [77] introduces a service-based architecture for centralized, topology-aware algorithm selection. While some schedulers (e.g., CASSINI, MLTCP and MCCS) realize flow-level fairness for multi-job scheduling, LEVELLER is the first communication scheduling system to achieve job-level max-min fairness while incurring a minimal impact on system-wide efficiency.

Intra-job communication scheduling. Collective communication scheduling within individual jobs (e.g., TACCL [68], TECCL [47] and SYNDICATE [50]) has been extensively studied for various topologies and collectives. Such approaches leverage job-specific training patterns to optimize performance of a single job, but agnostic to inter-job contention. LEVELLER is orthogonal to them, as it focuses on flow scheduling among different jobs.

Job scheduling. Existing work extensively explores GPU allocation and job placement to improve DLT job efficiency [35, 80, 81, 85, 87] or fairness [49, 88]. LEVELLER addresses network contention under arbitrary job schedules, making it a complementary, orthogonal solution to job schedulers.

9 Conclusion

LEVELLER provides a simple, scalable, and practical solution to the new problem of communication scheduling fairness in multi-tenant DLT clusters. By introducing *normalized progress rate*, we provide a unified way to quantify communication impact on job-level training performance. While our system establishes a foundational 1/2-approximation of max-min fairness, we hope this serves as a starting point for future research to explore tighter bounds and adopt this metric to further refine communication scheduling.

Acknowledgments

We thank all the anonymous reviewers and our shepherd for their constructive suggestions.

References

- [1] 1993. MPI: A message passing interface. In *Supercomputing '93:Proceedings of the 1993 ACM/IEEE Conference on Supercomputing*. 878–883. <https://doi.org/10.1145/169627.169855>
- [2] 2025. Symbolic Tensor Graph (STG) Generator. https://github.com/astra-sim/symbolic_tensor_graph.
- [3] Saksham Agarwal, Shijin Rajakrishnan, Akshay Narayan, Rachit Agarwal, David Shmoys, and Amin Vahdat. 2018. Sincronia: Near-optimal network design for coflows. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*. 16–29.
- [4] Alibaba. [n. d.]. Alibaba gpu cluster dataset 2023. <http://github.com/alibaba/alibaba-lingjun-dataset-2023>.
- [5] Mohammad Alizadeh, Shuang Yang, Milad Sharif, Sachin Katti, Nick McKeown, Balaji Prabhakar, and Scott Shenker. 2013. pFabric: Minimal near-optimal data-center transport. *ACM SIGCOMM Computer Communication Review* 43, 4 (2013), 435–446.
- [6] Wei Bai, Li Chen, Kai Chen, Dongsu Han, Chen Tian, and Hao Wang. 2017. PIAS: Practical information-agnostic flow scheduling for commodity data centers. *IEEE/ACM Transactions on Networking* 25, 4 (2017), 1954–1967.
- [7] Michael A Bender, Soumen Chakrabarti, and Sambavi Muthukrishnan. 1998. Flow and Stretch Metrics for Scheduling Continuous Job Streams.. In *SODA*, Vol. 98. 270–279.
- [8] Manaf Bin-Yahya, Amir Shani, Hossein Shafieirad, Seyed Hossein Mortazavi, Chen Ying, Aaron Wang, Geng Li, and Majid Ghaderi. 2025. Symphony: Collective Coordination in Multi-Tenant GPU Clusters. In *2025 IEEE 33rd International Conference on Network Protocols (ICNP)*. IEEE, 1–13.
- [9] Bob Briscoe. 2007. Flow rate fairness: Dismantling a religion. *ACM SIGCOMM Computer Communication Review* 37, 2 (2007), 63–74.
- [10] Jiamin Cao, Yu Guan, Kun Qian, Jiaqi Gao, Wencong Xiao, Jianbo Dong, Binzhang Fu, et al. 2024. Crux: Gpu-efficient communication scheduling for deep learning training. In *Proceedings of the ACM SIGCOMM 2024 Conference*. 1–15.
- [11] Shubham Chaudhary, Ramachandran Ramjee, Muthian Sivathanu, Nipun Kwatra, and Srinidhi Viswanatha. 2020. Balancing efficiency and fairness in heterogeneous GPU clusters for deep learning. In *Proceedings of the Fifteenth European Conference on Computer Systems (Heraklion, Greece) (EuroSys '20)*. Association for Computing Machinery, New York, NY, USA, Article 1, 16 pages.
- [12] Chang Chen, Xiuhong Li, Qianchao Zhu, Jiangfei Duan, Peng Sun, Xingcheng Zhang, and Chao Yang. 2024. Centauri: Enabling efficient scheduling for communication-computation overlap in large model training via communication partitioning. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*. 178–191.
- [13] Li Chen, Wei Cui, Baochun Li, and Bo Li. 2016. Optimizing coflow completion times with utility max-min fairness. In *IEEE INFOCOM 2016-The 35th Annual IEEE International Conference on Computer Communications*. IEEE, 1–9.
- [14] Mosharaf Chowdhury, Zhenhua Liu, Ali Ghodsi, and Ion Stoica. 2016. {HUG}: {Multi-Resource} fairness for correlated and elastic demands. In *13th USENIX symposium on networked systems design and implementation (NSDI 16)*. 407–424.
- [15] Mosharaf Chowdhury, Yuan Zhong, and Ion Stoica. 2014. Efficient coflow scheduling with varies. In *Proceedings of the 2014 ACM conference on SIGCOMM*. 443–454.
- [16] Cisco. 2023. IPv4 Loop-Free Alternate Fast Reroute. https://www.cisco.com/c/en/us/td/docs/ios-xml/ios/iproute_pi/configuration/15-s/iri-15-s-book/iri-ip-lfa-frf.pdf
- [17] Ultra Ethernet Consortium et al. 2023. Overview of and Motivation for the Forthcoming Ultra Ethernet Consortium Specification.
- [18] Francesco De Pellegrini, Vaibhav Kumar Gupta, Rachid El Azouzi, Serigne Gueye, Cedric Richier, and Jeremie Leguay. 2024. Fair coflow scheduling via controlled slowdown. *IEEE Transactions on Parallel and Distributed Systems* (2024).
- [19] DeepSeek-AI. 2025. Deepseek-Coder-1.3b-Instruct. ModelScope. <https://www.modelscope.cn/models/deepseek-ai/deepseek-coder-1.3b-instruct> [Online].
- [20] Advait Dixit, Pawan Prakash, Y Charlie Hu, and Ramana Rao Kompella. 2013. On the impact of packet spraying in data center networks. In *2013 proceedings ieee infocom*. IEEE, 2130–2138.
- [21] Nandita Dukkkipati and Nick McKeown. 2005. *Why Flow-Completion Time is the Right metric for Congestion Control and why this means we need new algorithms*. Technical Report. Citeseer.
- [22] William Fedus, Barret Zoph, and Noam Shazeer. 2022. Switch transformers: Scaling to trillion parameter models with simple and efficient sparsity. *Journal of Machine Learning Research* 23, 120 (2022), 1–39.
- [23] Nissim Francez. 2012. *Fairness*. Springer Science & Business Media.
- [24] Adithya Gangidi, Rui Miao, Shengbao Zheng, Sai Jayesh Bondu, Guilherme Goes, Hany Morsy, Rohit Puri, et al. 2024. Rdma over ethernet for distributed training at meta scale. In *Proceedings of the ACM SIGCOMM 2024 Conference*. 57–70.
- [25] Hao Ge, Junda Feng, Qi Huang, Fangcheng Fu, Xiaonan Nie, Lei Zuo, Haibin Lin, et al. 2025. ByteScale: Communication-Efficient Scaling of LLM Training with a 2048K Context Length on 16384 GPUs. In *Proceedings of the ACM SIGCOMM 2025 Conference*. 963–978.
- [26] Ali Ghodsi, Matei Zaharia, Benjamin Hindman, Andy Konwinski, Scott Shenker, and Ion Stoica. 2011. Dominant resource fairness: Fair allocation of multiple resource types. In *8th USENIX symposium on networked systems design and implementation (NSDI 11)*.
- [27] Team GLM, :, Aohan Zeng, Bin Xu, Bowen Wang, Chenhui Zhang, Da Yin, et al. 2024. ChatGLM: A Family of Large Language Models from GLM-130B to GLM-4 All Tools. arXiv:2406.12793 [cs.CL] <https://arxiv.org/abs/2406.12793>
- [28] Google. 2025. Gemma-2B. ModelScope. <https://www.modelscope.cn/models/AI-ModelScope/gemma-2b> [Online].
- [29] Juncheng Gu, Mosharaf Chowdhury, Kang G Shin, Yibo Zhu, Myeongjae Jeon, Junjie Qian, Hongqiang Liu, and Chuanxiong Guo. 2019. Tiresias: A {GPU} cluster manager for distributed deep learning. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*. 485–500.
- [30] Chuanxiong Guo, Haitao Wu, Zhong Deng, Gaurav Soni, Jianxi Ye, Jitu Padhye, and Marina Lipshteyn. 2016. RDMA over commodity ethernet at scale. In *Proceedings of the 2016 ACM SIGCOMM Conference*. 202–215.
- [31] Daya Guo, Qihao Zhu, Dejian Yang, Zhenda Xie, Kai Dong, Wentao Zhang, Guanting Chen, et al. 2024. DeepSeek-Coder: When the Large Language Model Meets Programming – The Rise of Code Intelligence. arXiv:2401.14196 [cs.SE] <https://arxiv.org/abs/2401.14196>
- [32] Jatinder ND Gupta. 1996. Comparative evaluation of heuristic algorithms for the single machine scheduling problem with two operations per job and time-lags. *Journal of Global Optimization* 9, 3 (1996), 239–253.
- [33] Juris Hartmanis. 1982. Computers and intractability: a guide to the theory of np-completeness (michael r. Garey and david s. Johnson). *Siam Review* 24, 1 (1982), 90.
- [34] Chi-Yao Hong, Matthew Caesar, and P Brighten Godfrey. 2012. Finishing flows quickly with preemptive scheduling. *ACM SIGCOMM Computer Communication Review* 42, 4 (2012), 127–138.
- [35] Qinghao Hu, Meng Zhang, Peng Sun, Yonggang Wen, and Tianwei Zhang. 2023. Lucid: A non-intrusive, scalable and interpretable scheduler for deep learning training jobs. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*. 457–472.
- [36] Yanping Huang, Youlong Cheng, Ankur Bapna, Orhan Firat, Dehao Chen, Mia Chen, HyukJoong Lee, et al. 2019. Gpipe: Efficient training of giant neural networks using pipeline parallelism. *Advances in neural information processing systems* 32 (2019).
- [37] Rajendra K Jain, Dah-Ming W Chiu, William R Hawe, et al. 1984. A quantitative measure of fairness and discrimination. *Eastern Research Laboratory, Digital Equipment Corporation, Hudson, MA* 21, 1 (1984), 2022–2023.
- [38] Abhinav Jangda, Jun Huang, Guodong Liu, Amir Hossein Nodehi Sabet, Saeed Maleki, Youshan Miao, Madanlal Musuvathi, et al. 2022. Breaking the computation and communication abstraction barrier in distributed machine learning workloads. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. 402–416.
- [39] 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.
- [40] Ziheng Jiang, Haibin Lin, Yinmin Zhong, Qi Huang, Yangrui Chen, Zhi Zhang, Yanghua Peng, et al. 2024. {MegaScale}: Scaling large language model training to more than 10,000 {GPUs}. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*. 745–760.
- [41] Changhoon Kim, Anirudh Sivaraman, Naga Katta, Antonin Bas, Advait Dixit, Lawrence J Wobker, et al. 2015. In-band network telemetry via programmable dataplanes. In *ACM SIGCOMM*, Vol. 15. 1–2.
- [42] Gautam Kumar, Nandita Dukkkipati, Keon Jang, Hassan MG Wassel, Xian Wu, Behnam Montazeri, Yaogong Wang, et al. 2020. Swift: Delay is simple and effective for congestion control in the datacenter. In *Proceedings of the 2020 ACM conference on SIGCOMM*. 514–528.
- [43] Shen Li, Yanli Zhao, Rohan Varma, Omkar Salpekar, Pieter Noordhuis, Teng Li, Adam Paszke, et al. 2020. Pytorch distributed: Experiences on accelerating data parallel training. *arXiv preprint arXiv:2006.15704* (2020).
- [44] Heng Liao, Bingyang Liu, Xianping Chen, Zhigang Guo, Chuanning Cheng, Jianbing Wang, Xiangyu Chen, et al. 2025. UB-Mesh: a Hierarchically Localized nD-FullMesh Datacenter Network Architecture. *arXiv preprint arXiv:2503.20377* (2025).
- [45] Xudong Liao, Yijun Sun, Han Tian, Xinchun Wan, Yilun Jin, Zilong Wang, Zhenghang Ren, et al. 2025. MixNet: A Runtime Reconfigurable Optical-Electrical Fabric for Distributed Mixture-of-Experts Training. In *Proceedings of the ACM SIGCOMM 2025 Conference*. 554–574.
- [46] Bin Lin, Zhenyu Tang, Yang Ye, Jinfa Huang, Junwu Zhang, Yatian Pang, Peng Jin, et al. 2026. Moe-llava: Mixture of experts for large vision-language models. *IEEE Transactions on Multimedia* (2026).
- [47] Xuting Liu, Behnaz Arzani, Siva Kesava Reddy Kakarla, Liangyu Zhao, Vincent Liu, Miguel Castro, Srikanth Kandula, and Luke Marshall. 2024. Rethinking machine learning collective communication as a multi-commodity flow problem.

- In *Proceedings of the ACM SIGCOMM 2024 Conference*. 16–37.
- [48] Jie Lu, Jiaqi Gao, Fei Feng, Zhiqiang He, Menglei Zheng, Kun Liu, Jun He, et al. 2025. Alibaba stellar: A new generation rdma network for cloud ai. In *Proceedings of the ACM SIGCOMM 2025 Conference*. 453–466.
 - [49] 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.
 - [50] Kshiteej Mahajan, Ching-Hsiang Chu, Srinivas Sridharan, and Aditya Akella. 2023. Better together: Jointly optimizing {ML} collective scheduling and execution planning using {SYNDICATE}. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*. 809–824.
 - [51] Qingkai Meng, Hao Zheng, Zhenhui Zhang, ChonLam Lao, Chengyuan Huang, Baojia Li, Ziyuan Zhu, et al. 2025. Astral: A datacenter infrastructure for large language model training at scale. In *Proceedings of the ACM SIGCOMM 2025 Conference*. 609–625.
 - [52] Jeonghoon Mo and Jean Walrand. 2002. Fair end-to-end window-based congestion control. *IEEE/ACM Transactions on networking* 8, 5 (2002), 556–567.
 - [53] Alix Munier and Francis Sourd. 2003. Scheduling chains on a single machine with non-negative time lags. *Mathematical Methods of Operations Research* 57, 1 (2003), 111–123.
 - [54] Shanmugavelayutham Muthukrishnan, Rajmohan Rajaraman, Anthony Shaheen, and Johannes E Gehrke. 1999. Online scheduling to minimize average stretch. In *40th Annual Symposium on Foundations of Computer Science (Cat. No. 99CB37039)*. IEEE, 433–443.
 - [55] 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.
 - [56] Deepak Narayanan, Mohammad Shoeybi, Jared Casper, Patrick LeGresley, Mostofa Patwary, Vijay Korthikanti, Dmitri Vainbrand, et al. 2021. Efficient large-scale language model training on gpu clusters using megatron-lm. In *Proceedings of the international conference for high performance computing, networking, storage and analysis*. 1–15.
 - [57] NVIDIA. 2023. NVIDIA InfiniBand Adaptive Routing Technology Accelerating HPC and AI Applications. https://www.amax.com/content/files/2023/12/NVIDIA_InfiniBand_Adaptive_Routing_Technology_Insights_Whitepaper.pdf
 - [58] NVIDIA. 2025. Llama-3.1-Nemotron-Nano-4B-v1.1. ModelScope. <https://www.modelscope.cn/models/nv-community/Llama-3.1-Nemotron-Nano-4B-v1.1> [Online].
 - [59] NVIDIA. 2025. Nccl (nvidia collective communications library). <https://github.com/NVIDIA/nccl>
 - [60] NVIDIA. 2025. NVIDIA NVLink and NVLink Switch). <https://www.nvidia.com/en-us/data-center/nvlink/>
 - [61] 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.
 - [62] Chenchen Qi, Wenfei Wu, Yongcan Wang, Keqiang He, Yu-Hsiang Kao, Zongying He, Chen-Yu Yen, et al. 2025. SGLB: Scalable and Robust Global Load Balancing in Commodity AI Clusters. In *Proceedings of the ACM SIGCOMM 2025 Conference*. 626–644.
 - [63] Kun Qian, Yongqing Xi, Jiamin Cao, Jiaqi Gao, Yichi Xu, Yu Guan, Binzhang Fu, et al. 2024. Alibaba hpn: A data center network for large language model training. In *Proceedings of the ACM SIGCOMM 2024 Conference*. 691–706.
 - [64] Robi Rahman. 2024. AI training cluster sizes increased by more than 20x since 2016. <https://epoch.ai/data-insights/training-cluster-size> Accessed: 2025-11-03.
 - [65] Sudarsanan Rajasekaran, Manya Ghobadi, and Aditya Akella. 2024. {CASSINI}:{Network-Aware} job scheduling in machine learning clusters. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*. 1403–1420.
 - [66] Sudarsanan Rajasekaran, Sanjoli Narang, Anton A Zabreyko, and Manya Ghobadi. 2024. Mltpc: A distributed technique to approximate centralized flow scheduling for machine learning. In *Proceedings of the 23rd ACM Workshop on Hot Topics in Networks*. 167–176.
 - [67] Hossein Shafieirad, Amir Shani, Manaf Bin-Yahya, Seyed Hossein Mortazavi, Geng Li, Xinle Du, Tao Su, et al. 2025. Harmonics: Scalable Collective Scheduling in Multi-Tenant GPU Clusters. *Proceedings of the ACM on Networking* 3, CoNEXT'24 (2025), 1–20.
 - [68] Aashaka Shah, Vijay Chidambaram, Meghan Cowan, Saeed Maleki, Madan Musuvathi, Todd Mytkowicz, Jacob Nelson, et al. 2023. {TACCL}: Guiding collective algorithm synthesis using communication sketches. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*. 593–612.
 - [69] Pavel Shamis, Manjunath Gorentla Venkata, M Graham Lopez, Matthew B Baker, Oscar Hernandez, Yossi Itigin, Mike Dubman, et al. 2015. UCX: an open source framework for HPC network APIs and beyond. In *2015 IEEE 23rd Annual Symposium on High-Performance Interconnects*. IEEE, 40–43.
 - [70] Ying Sheng, Shiyi Cao, Dacheng Li, Banghua Zhu, Zhuohan Li, Danyang Zhuo, Joseph E Gonzalez, and Ion Stoica. 2024. Fairness in serving large language models. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*. 965–988.
 - [71] Srinivas Sridharan, Taekyung Heo, Louis Feng, Zhaodong Wang, Matt Bergeron, Wenyin Fu, Shengbao Zheng, et al. 2023. Chakra: Advancing performance benchmarking and co-design using standardized execution traces. *arXiv preprint arXiv:2305.14516* (2023).
 - [72] Gemma Team, Thomas Mesnard, Cassidy Hardin, Robert Dadashi, Surya Bhu-patiraju, Shreya Pathak, Laurent Sifre, et al. 2024. Gemma: Open Models Based on Gemini Research and Technology. [arXiv:2403.08295 \[cs.CL\]](https://arxiv.org/abs/2403.08295) <https://arxiv.org/abs/2403.08295>
 - [73] Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, et al. 2023. LLaMA: Open and Efficient Foundation Language Models. [arXiv:2302.13971 \[cs.CL\]](https://arxiv.org/abs/2302.13971) <https://arxiv.org/abs/2302.13971>
 - [74] Wei Wang, Shiyao Ma, Bo Li, and Baochun Li. 2017. Coflex: Navigating the fairness-efficiency tradeoff for coflow scheduling. In *IEEE INFOCOM 2017-IEEE Conference on Computer Communications*. IEEE, 1–9.
 - [75] William Won, Taekyung Heo, Saeed Rashidi, Srinivas Sridharan, Sudarshan Srinivasan, and Tushar Krishna. 2023. Astra-sim2.0: Modeling hierarchical networks and disaggregated systems for large-model training at scale. In *2023 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE, 283–294.
 - [76] Bingyang Wu, Zili Zhang, Zhihao Bai, Xuanzhe Liu, and Xin Jin. 2023. Transparent {GPU} sharing in container clouds for deep learning workloads. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*. 69–85.
 - [77] Yongji Wu, Yechen Xu, Jingrong Chen, Zhaodong Wang, Ying Zhang, Matthew Lentz, and Danyang Zhuo. 2024. MCCS: A Service-based Approach to Collective Communication for Multi-Tenant Cloud. In *Proceedings of the ACM SIGCOMM 2024 Conference*. 679–690.
 - [78] Zhiyi Yao, Pengbo Hu, Congcong Miao, Xuya Jia, Zuning Liang, Yuedong Xu, Chunzhi He, et al. 2025. Holmes: Localizing Irregularities in LLM Training with Mega-scale GPU Clusters. In *22nd USENIX Symposium on Networked Systems Design and Implementation (NSDI 25)*. USENIX Association, Philadelphia, PA, 523–540.
 - [79] Zhisheng Ye, Wei Gao, Qinghao Hu, Peng Sun, Xiaolin Wang, Yingwei Luo, Tianwei Zhang, and Yonggang Wen. 2024. Deep learning workload scheduling in gpu datacenters: A survey. *Comput. Surveys* 56, 6 (2024), 1–38.
 - [80] Menglu Yu, Bo Ji, Hridesh Rajan, and Jia Liu. 2022. On scheduling ring-all-reduce learning jobs in multi-tenant GPU clusters with communication contention. In *Proceedings of the Twenty-Third International Symposium on Theory, Algorithmic Foundations, and Protocol Design for Mobile Networks and Mobile Computing*. 21–30.
 - [81] Menglu Yu, Ye Tian, Bo Ji, Chuan Wu, Hridesh Rajan, and Jia Liu. 2022. Gadget: Online resource optimization for scheduling ring-all-reduce learning jobs. In *IEEE INFOCOM 2022-IEEE conference on computer communications*. IEEE, 1569–1578.
 - [82] Adrian Zapletal and Fernando Kuipers. 2023. Slowdown as a metric for congestion control fairness. In *Proceedings of the 22nd ACM Workshop on Hot Topics in Networks*. 205–212.
 - [83] Lin Zhang, Shaohuai Shi, Xiaowen Chu, Wei Wang, Bo Li, and Chengjian Liu. 2023. DeAR: Accelerating distributed deep learning with fine-grained all-reduce pipelining. In *2023 IEEE 43rd International Conference on Distributed Computing Systems (ICDCS)*. IEEE, 142–153.
 - [84] Xiandong Zhang. 2010. *Scheduling with time lags*. Number EPS-2010-206-LIS.
 - [85] Hanyu Zhao, Zhenhua Han, Zhi Yang, Quanlu Zhang, Fan Yang, Lidong Zhou, Mao Yang, et al. 2020. {HiveD}: Sharing a {GPU} cluster for deep learning with guarantees. In *14th USENIX symposium on operating systems design and implementation (OSDI 20)*. 515–532.
 - [86] Yanli Zhao, Andrew Gu, Rohan Varma, Liang Luo, Chien-Chin Huang, Min Xu, Less Wright, et al. 2023. Pytorch fsdp: experiences on scaling fully sharded data parallel. *arXiv preprint arXiv:2304.11277* (2023).
 - [87] Yihao Zhao, Yuanqiang Liu, Yanghua Peng, Yibo Zhu, Xuanzhe Liu, and Xin Jin. 2022. Multi-resource interleaving for deep learning training. In *Proceedings of the ACM SIGCOMM 2022 Conference*. 428–440.
 - [88] Pengfei Zheng, Rui Pan, Tarannum Khan, Shivaram Venkataraman, and Aditya Akella. 2023. Shockwave: Fair and efficient cluster scheduling for dynamic adaptation in machine learning. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*. 703–723.
 - [89] ZhipuAI and Tsinghua KEG Laboratory. 2025. ChatGLM3-6B. ModelScope. <https://www.modelscope.cn/models/ZhipuAI/chatglm3-6b> [Online].
 - [90] Yibo Zhu, Haggai Eran, Daniel Firestone, Chuanxiong Guo, Marina Lipshteyn, Yehonatan Liron, Jitendra Padhye, et al. 2015. Congestion control for large-scale RDMA deployments. *ACM SIGCOMM Computer Communication Review* 45, 4 (2015), 523–536.

Appendices

Appendices are supporting material that has not been peer-reviewed.

A Max-Min Fairness

We formally define the notion of max-min fairness adopted in this paper.

DEFINITION 3 (MAX-MIN FAIRNESS). Let $x \in \mathbb{R}^n$ be a vector of performance metrics (e.g., long-term normalized progress rates of jobs), and let $\text{sort}_\uparrow(x)$ denote the vector obtained by sorting the components of x in non-decreasing order. A vector x^* is said to be max-min fair if it is a solution to the following lexicographic maximization problem:

$$x^* \in \arg \max \text{sort}_\uparrow(x). \quad (7)$$

Equivalently, x^* is max-min fair if and only if for any other feasible vector x , the sorted vector $\text{sort}_\uparrow(x^*)$ is lexicographically no smaller than $\text{sort}_\uparrow(x)$; that is, at the first index where the two sorted vectors differ, x^* has a strictly larger value.

This definition captures the classical intuition of max-min fairness: no component of x^* can be increased without decreasing another component that is no larger than it. In our setting, this ensures that the worst-performing job is prioritized first, followed by the second worst, and so on, thereby balancing efficiency and fairness across all jobs.

B Proof of Lemma 1

PROOF. Recall Definition 1:

$$R_j(t) = \frac{N_j(t)T_j^{\text{ideal}}}{t - t_j^0}, \quad (8)$$

and the original objective $\mathcal{P}_1$ lexicographically maximizes $\text{sort}_\uparrow(\{R_j(T_{\text{end}}) \mid j \in \mathcal{J}^e\})$ over a sufficiently long window $[T_{\text{begin}}, T_{\text{end}}]$ with $T_{\text{begin}} \leq t_j^0$ for all j and no job completes before T_{end} .

Step 1: Start-time offsets are negligible in a long execution window. Let $t_{\min} \triangleq \min_j t_j^0$, $t_{\max} \triangleq \max_j t_j^0$, and $\Delta \triangleq t_{\max} - t_{\min}$. For any j ,

$$\frac{1}{T_{\text{end}} - t_{\max}} \leq \frac{1}{T_{\text{end}} - t_j^0} \leq \frac{1}{T_{\text{end}} - t_{\min}}. \quad (9)$$

Hence, for any two jobs i, j ,

$$\begin{aligned} \frac{T_{\text{end}} - t_i^0}{T_{\text{end}} - t_j^0} &\in \left[\frac{T_{\text{end}} - t_{\max}}{T_{\text{end}} - t_{\min}}, \frac{T_{\text{end}} - t_{\min}}{T_{\text{end}} - t_{\max}} \right] \\ &= \left[1 - \Theta\left(\frac{\Delta}{T_{\text{end}} - t_{\min}}\right), 1 + \Theta\left(\frac{\Delta}{T_{\text{end}} - t_{\max}}\right) \right]. \end{aligned} \quad (10)$$

Therefore, when T_{end} is sufficiently large (as assumed), the denominators $\{T_{\text{end}} - t_j^0\}$ differ only by a vanishing $o(1)$ relative factor. In particular, maximizing the lexicographic vector of $\{R_j(T_{\text{end}})\}$ is asymptotically equivalent to maximizing the lexicographic vector of the corresponding *actual progress* $\{N_j(T_{\text{end}})T_j^{\text{ideal}}\}$, and the long-term max-min fairness conclusions derived below are insensitive to the specific value of t_j^0 .

Step 2: Construct the normalized instance and relate rates to a makespan. Let $L \triangleq \text{LCM}(\mathcal{J}^e)$ be the least common multiple of

$\{T_j^{\text{ideal}} \mid j \in \mathcal{J}^e\}^1$, and define $\bar{N}_j \triangleq \frac{L}{T_j^{\text{ideal}}}$. Consider any schedule $\bar{S}$ that completes $\bar{N}_j$ iterations for every job j in makespan

$$ms(\bar{S}) \triangleq \text{makespan}(\bar{N}_j \mid j \in \mathcal{J}^e). \quad (11)$$

By repeating $\bar{S}$ periodically, after q periods, each job completes at least $q\bar{N}_j$ iterations in time $qms(\bar{S})$. Therefore, as t grows, the long-term normalized progress rate for each job j satisfies

$$\begin{aligned} \liminf_{t \rightarrow \infty} R_j(t) &\geq \lim_{q \rightarrow \infty} \frac{(q\bar{N}_j)T_j^{\text{ideal}}}{qms(\bar{S})} \\ &= \frac{\bar{N}_j T_j^{\text{ideal}}}{ms(\bar{S})} = \frac{L}{ms(\bar{S})}, \end{aligned} \quad (12)$$

where the last equality uses $\bar{N}_j T_j^{\text{ideal}} = L$. Hence, repeating a schedule with makespan $ms(\bar{S})$ yields a *uniform* long-term rate $L/ms(\bar{S})$ for all jobs.

Step 3: Optimal uniform rate equals L divided by the minimum makespan. Let $ms^* \triangleq \min_{\bar{S}} \text{makespan}(\bar{N}_j \mid j \in \mathcal{J}^e)$ be the optimal value of $\mathcal{P}_2$. According to Step 2, the repetition of an optimal schedule for $\mathcal{P}_2$ achieves

$$R_j \geq \frac{L}{ms^*}, \quad \forall j, \quad (13)$$

so the max-min (and thus the first key of the lexicographic objective) is at least L/ms^* .

Conversely, consider any schedule (not necessarily periodic) that achieves a long-term max-min *normalized progress rate* at least R , i.e., $\liminf_{t \rightarrow \infty} \min_j R_j(t) \geq R$. Then for sufficiently large t , every job satisfies

$$N_j(t)T_j^{\text{ideal}} \geq (R - o(1))(t - t_j^0). \quad (14)$$

Pick t large enough so that $o(1)$ is negligible and $t - t_j^0$ is arbitrarily close (in relative error) across jobs (Step 1). In particular, to make every job complete at least $\bar{N}_j$ iterations, it is necessary to accumulate $\bar{N}_j T_j^{\text{ideal}} = L$ actual progress for each job. Under rate R , this requires time at least L/R (up to the vanishing start-time effect), implying that any schedule completing $\{\bar{N}_j\}$ must have makespan at least L/R . Therefore,

$$ms^* \geq \frac{L}{R} \implies R \leq \frac{L}{ms^*}. \quad (15)$$

Combining both directions yields the optimal uniform rate

$$R^* = \frac{L}{ms^*} = \frac{\text{LCM}(\mathcal{J}^e)}{\min \text{makespan}(\bar{N}_j \mid j \in \mathcal{J}^e)}. \quad (16)$$

Step 4: Lexicographic optimality via repeating the minimum makespan schedule. The schedule obtained by repeating an optimal $\mathcal{P}_2$ solution makes all jobs achieve the same long-term rate R^* . Such a rate vector is lexicographically max-min fair: one cannot increase any component without decreasing another component that is no larger (indeed, all are equal). Hence, an optimal solution of $\mathcal{P}_2$ induces an optimal solution of $\mathcal{P}_1$ by repetition, and maximizing $\mathcal{P}_1$ is equivalent to minimizing $\mathcal{P}_2$. $\square$

¹If T_j^{ideal} are rational numbers, one can scale them by a common factor to make them integers without affecting the arguments below.

C Proof of Theorem 1

PROOF. By Lemma 1, for sufficiently large T_{end} , the start-time offsets $\{t_j^0\}$ only introduce a vanishing $o(1)$ effect and do not affect the long-term max-min fairness objective. Hence, it suffices to establish a constant-factor approximation bound on the makespan of Problem $\mathcal{P}_2$, and then map this bound back to Problem $\mathcal{P}_1$.

Step 1: From max-min fairness to makespan. By Lemma 1, the max-min fairness problem $\mathcal{P}_1$ is equivalent to the makespan minimization problem $\mathcal{P}_2$ defined over the normalized iteration numbers $\{\bar{N}_j \mid j \in \mathcal{J}^e\}$. In particular, under the max-min fair schedule, all jobs achieve a uniform optimal rate

$$R^* = \frac{\text{LCM}(\mathcal{J}^e)}{\min \text{ makespan}(\bar{N}_j \mid j \in \mathcal{J}^e)}. \quad (17)$$

Therefore, a ρ -approximation on the makespan of $\mathcal{P}_2$ directly implies a $1/\rho$ -approximation on the max-min fairness objective in $\mathcal{P}_1$. Hence, it suffices to show that the scheduling policy S^σ achieves a 2-approximation for the optimal makespan.

Step 2: A lower bound on the optimal makespan. Consider a set of jobs $\mathcal{J}^e$ competing for a single link e . For job j , each iteration incurs computation time w_j and communication volume vol_j^e on link e . To complete $\bar{N}_j$ iterations, job j requires total computation time $\bar{N}_j w_j$ and total communication time $\bar{N}_j \text{vol}_j^e / B_e$.

Let $C_e^{\text{total}} = \sum_{j=1}^{|\mathcal{J}^e|} \bar{N}_j \text{vol}_j^e / B_e$ denote the total communication time of all jobs on link e , and $W_e^{\text{max}} = \max_{j \in \mathcal{J}^e} \bar{N}_j w_j$ denote the largest total computation time among all jobs. Any feasible schedule must transmit all communication traffic and allow the slowest job to finish its computation. Therefore, the optimal makespan ms^* satisfies

$$ms^* \geq \max\left(C_e^{\text{total}}, W_e^{\text{max}}\right). \quad (18)$$

Step 3: An upper bound on the makespan of S^σ . Under the scheduling policy S^σ , the link is *work-conserving*, *preemptive*, and $\sigma(t)$ -preserving. As a result, at any time, the link operates in one of the following two states:

- (1) *Busy*: the link is transmitting data for some job;
- (2) *Idle*: all jobs are in their computation phases, with no communication traffic on the link.

Since the policy is work-conserving and preemptive, the total duration during which the link is busy is exactly C_e^{total} . The idle periods can only occur when all jobs are computing. Such periods are therefore upper-bounded by the shortest total computation time among all jobs, defined as

$$W_e^{\text{min}} = \min_{j \in \mathcal{J}^e} \bar{N}_j w_j. \quad (19)$$

Consequently, the makespan under S^σ , denoted ms^σ , satisfies

$$ms^\sigma \leq C_e^{\text{total}} + W_e^{\text{min}}. \quad (20)$$

Step 4: Approximation ratio. Combining the lower bound on ms^* and the upper bound on ms^σ , we obtain

$$\frac{ms^\sigma}{ms^*} \leq \frac{C_e^{\text{total}} + W_e^{\text{min}}}{\max(C_e^{\text{total}}, W_e^{\text{max}})}. \quad (21)$$

Using the inequality $\max(a, b) \geq \frac{1}{2}(a + b)$, we obtain

$$\max(C_e^{\text{total}}, W_e^{\text{max}}) \geq \frac{1}{2}(C_e^{\text{total}} + W_e^{\text{max}}). \quad (22)$$

Moreover, since $W_e^{\text{min}} \leq W_e^{\text{max}}$, it follows that

$$\frac{ms^\sigma}{ms^*} \leq 2 \cdot \frac{C_e^{\text{total}} + W_e^{\text{min}}}{C_e^{\text{total}} + W_e^{\text{max}}} \leq 2. \quad (23)$$

Thus, S^σ achieves a 2-approximation for the optimal makespan in $\mathcal{P}_2$.

Step 5: Implication for max-min fairness. By Lemma 1, the makespan approximation immediately translates to the max-min fairness objective. Specifically, letting R_j^σ and R_j^* denote the rate of job j under S^σ and under the max-min fair schedule, respectively, we have

$$R_j^\sigma \geq \frac{1}{2} R_j^*, \quad \forall j \in \mathcal{J}^e. \quad (24)$$

This completes the proof. $\square$

D Proof of Corollary 1

PROOF. We prove the corollary by quantifying the additional loss introduced by the *lazy* update of the ordering $\sigma(t)$. Recall Definition 1: for job j ,

$$R_j(t) \triangleq \frac{N_j(t) T_j^{\text{ideal}}}{t - t_j^0}. \quad (25)$$

Let S^σ denote the greedy $\sigma(t)$ -preserving scheduling with *instant* updates (i.e., $\tau \rightarrow 0$), and let $S^{\sigma, \tau}$ denote its *lazy-update* variant in which $\sigma(t)$ is refreshed only at slot boundaries $t_m = m\tau$ and kept fixed within each interval $[t_m, t_{m+1})$.

Step 1: Baseline guarantee under instant updates. By Theorem 1, for any sufficiently large time horizon t , the instant-update policy achieves a 1/2-approximation to the max-min fair normalized progress rate. In particular,

$$R_j^\sigma(t) \geq \frac{1}{2} R_j^* = \frac{\text{LCM}(\mathcal{J}^e)}{2 \min \text{ makespan}(\bar{N}_j \mid j \in \mathcal{J}^e)}. \quad (26)$$

Step 2: Lazy updates can delay correcting the “slowest” job by at most τ . Fix a slot index m and consider the interval $[t_m, t_{m+1})$. At time t_m , $S^{\sigma, \tau}$ computes an ordering $\sigma(t_m)$ based on the rates at t_m and keeps this ordering unchanged until t_{m+1} . During this interval, the identity of the slowest job (the one with the smallest $R_j(t)$) may change, but $S^{\sigma, \tau}$ will not re-rank jobs until t_{m+1} .

Consider the worst case for max-min fairness: there exists a job j such that, immediately after t_m , it becomes the slowest job under the *instant* policy (hence it should be prioritized), yet it is not ranked as the slowest under $\sigma(t_m)$ and thus may be continuously preempted by higher-ranked jobs throughout the entire slot. In this case, relative to the instant-update policy, job j may lose at most one full slot of “being served as the slowest”. Therefore, the *maximum delay* before $S^{\sigma, \tau}$ corrects the ordering to prioritize the true slowest job is upper bounded by τ .

Step 3: Translate a τ -delay into an additive loss on normalized progress rate. We next bound the loss on the *numerator* of $R_j(t)$ caused by the above delay. Since $N_j(t)$ is the number of completed iterations by time t , and the policy is work-conserving and preemptive, the worst-case effect of an incorrect ordering within a slot is that the currently slowest job j makes *no communication progress* during at most τ time units (until the next update), whereas under the instant-update policy it would have been prioritized during that

period. Because each job's per-iteration *ideal* time is T_j^{ideal} , the loss in *actual progress* $N_j(t)T_j^{\text{ideal}}$ due to being delayed by at most τ is at most τ :

$$N_j^\sigma(t)T_j^{\text{ideal}} \geq N_j^{\sigma,\tau}(t)T_j^{\text{ideal}} \geq N_j^\sigma(t)T_j^{\text{ideal}} - \tau, \quad (27)$$

i.e., compared with the instant-update policy, the lazy-update policy can reduce the *actual progress* of the worst-off job by at most an additive τ . Intuitively, τ is an upper bound on how long the system may keep prioritizing a non-slowest job before re-ranking; thus the slowest job can lose at most τ units of time in terms of normalized progress.

Dividing both sides of (27) by the denominator $(t - t_j^0)$ (Definition 1), we obtain

$$R_j^{\sigma,\tau}(t) \geq R_j^\sigma(t) - \frac{\tau}{t - t_j^0}. \quad (28)$$

Taking minimum over all jobs $j \in \mathcal{J}^e$ and using $t - t_j^0 \geq t - \max_i t_i^0$ yields

$$\min_{j \in \mathcal{J}^e} R_j^{\sigma,\tau}(t) \geq \min_{j \in \mathcal{J}^e} R_j^\sigma(t) - \frac{\tau}{t - \max_i t_i^0}. \quad (29)$$

Step 4: Combine with Theorem 1. Substituting the baseline guarantee (26) into (29) gives

$$\min_{j \in \mathcal{J}^e} R_j^{\sigma,\tau}(t) \geq \frac{1}{2} R_j^* - \frac{\tau}{t - \max_i t_i^0}. \quad (30)$$

Since t is assumed to be sufficiently large and the evaluation window is long (thus $(t - \max_i t_i^0) = \Theta(t)$), we have $\frac{\tau}{t - \max_i t_i^0} = O(\tau/t)$. Therefore, as long as $\tau/t \rightarrow 0$ (equivalently, $\tau \ll t$), the additive loss vanishes and the lazy-update mechanism still achieves a 1/2-approximation in the long-term *normalized progress rate*.

This completes the proof. $\square$

E Proof of Corollary 2

We extend the single-link results to general networks by reducing the scheduling behavior of each job to that on a bottleneck link, following an abstraction similar to CRUX [10].

E.1 Definition of Bottleneck Link

DEFINITION 4 (LOCAL CONVERGED RATE). *Given the network topology, flow routing paths, and a scheduling policy, the Local Converged Rate (LCR) of a link l_n is the normalized progress rate to which all traversing jobs converge when assuming all other links in the network have infinite bandwidth.*

Let $LCR_{l_n}^*$ and $LCR_{l_n}^\sigma$ denote the LCR of link l_n under the optimal policy and σ -preserving policy, respectively.

LCR example. As illustrated in Figure (a), assuming each job consists of a single flow, LCR_1^σ is the minimum converged rate achieved by jobs 1 and 2 under the σ -preserving policy on link 1, assuming link 2 has infinite bandwidth. Similarly, LCR_2^σ represents the minimum rate of jobs 1, 3, and 4 when considering only link 2.

DEFINITION 5 (BOTTLENECK LINK OF A JOB). *The bottleneck link of job i is defined as the link with the minimum LCR among all links traversed by the constituent flows of job i .*

DEFINITION 6 (GLOBAL BOTTLENECK LINK). *In a general network topology, the link with the minimum LCR is defined as the global bottleneck link.*

E.2 Property of Bottleneck Link

LEMMA 2 (BOTTLENECK REDUCTION UNDER THE OPTIMAL POLICY). *Under the optimal scheduling policy, the minimum progress rate across all jobs is exactly determined by the LCR of the global bottleneck link, i.e., $R^* = \min_{l_n \in \mathcal{E}} LCR_{l_n}^*$.*

PROOF. We establish $R^* = \min_{l_n \in \mathcal{E}} LCR_{l_n}^*$ by proving the upper and lower bounds separately.

Upper bound ($R^* \leq \min_{l_n \in \mathcal{E}} LCR_{l_n}^*$). By definition, $LCR_{l_n}^*$ denotes the maximum achievable minimum progress rate when optimizing scheduling objectives under the isolated bandwidth constraint of a single link l_n . As the global optimal network-wide progress rate R^* must simultaneously satisfy the joint constraints of all links, the upper bound holds.

Lower bound ($R^* \geq \min_{l_n \in \mathcal{E}} LCR_{l_n}^*$). Let l be the global bottleneck link under the optimal policy. We fix the scheduling of all flows on l and evaluate any other link l' via two cases:

- (1) If the flows on link l do not traverse any l' , then the scheduling has no impact on each other.
- (2) If some flows traverse both link l and l' , then the two links have dependencies. Assuming the scheduling pattern on link l is fixed, we can treat their occupied time slots on link l' as unavailable. On link l' alone, all flows could originally achieve at least $LCR_{l'}^*$. However, because the shared flows are bound by the bottleneck l , they only achieve a lower rate $LCR_l^* \leq LCR_{l'}^*$, leaving more available time for the remaining flows on link l' . Therefore, the achievable progress rate for these remaining flows must be at least $LCR_{l'}^*$. By mathematical induction, the progress rate of all jobs is bounded below by $\min_{l_n \in \mathcal{E}} LCR_{l_n}^*$. Since the optimal policy must perform at least as well as this constructive scheduling, we have $R^* \geq \min_{l_n \in \mathcal{E}} LCR_{l_n}^*$.

Combining the upper and lower bounds completes the proof. $\square$

LEMMA 3 (BOTTLENECK REDUCTION UNDER THE σ -PRESERVING POLICY). *Under the σ -preserving scheduling policy, the minimum progress rate across all jobs is exactly determined by the LCR of the global bottleneck link, i.e., $R^\sigma = \min_{l_n \in \mathcal{E}} LCR_{l_n}^\sigma$.*

PROOF. Following a similar logic to the proof of Lemma 2, it readily follows that $R^\sigma \leq \min_{l_n \in \mathcal{E}} LCR_{l_n}^\sigma$.

By definition, the rates of all flows on the global bottleneck link are the minimum globally. Based on the σ -preserving scheduling policy, these flows are assigned with the highest priority across the network. This means these flows will be completely free from interference by other flows on other links. This property is equivalent to the assumption in Lemma 2, allowing the remainder of the proof to follow the same. $\square$

Single-flow job example. As illustrated in Figure 18(a), since $LCR_2^\sigma(0.7) < LCR_1^\sigma(0.8)$, link L_2 is both the job-level and global bottleneck. For Job 1 (consisting of flow f_1), its rate is upper-bounded by L_2 , making $R_1^\sigma \leq 0.7$. Meanwhile, because Job 1 has the minimum rate in the network, it is granted the highest priority on link L_1

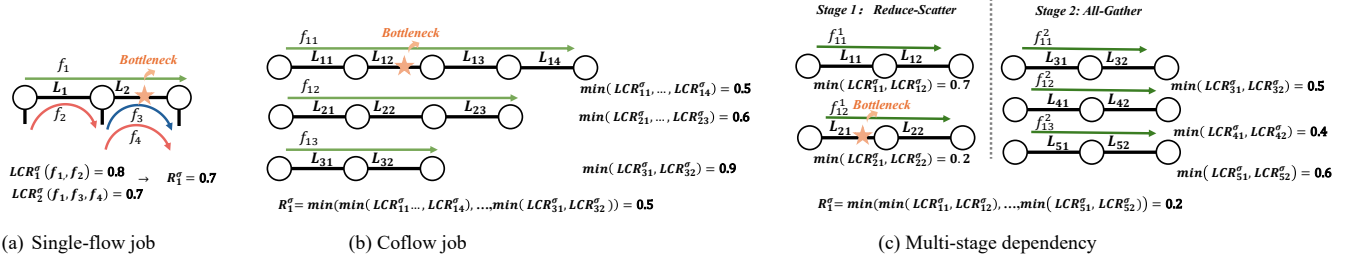


Figure 18: Bottleneck link examples in various scenarios. The minimum progress rate is determined by the global bottleneck link LCR.

under the σ -preserving policy, guaranteeing that $R_1^\sigma \geq 0.7$. Consequently, its progress rate converges precisely to the bottleneck LCR, i.e., $R_1^\sigma = \min(LCR_1^\sigma, LCR_2^\sigma) = 0.7$.

Coflow example. Similarly, consider a coflow scenario where Job 1 consists of parallel flows f_{11} , f_{12} , and f_{13} across different paths, as illustrated in Figure 18(b). Under the σ -preserving scheduling, link L_{12} serves as the job-level bottleneck. When considering all constituent flows simultaneously, Job 1's overall progress rate is strictly bounded by its slowest path, dictating $R_1^\sigma \leq 0.5$. Meanwhile, because Job 1 exhibits the minimum rate, LEVELLER grants it the highest priority across all traversed links. This isolation shields Job 1 from cross-traffic interference on other non-bottleneck links, guaranteeing that $R_1^\sigma \geq 0.5$. Consequently, its rate equals to the minimum LCR of all constituent links, yielding $R_1^\sigma = \min(\min(LCR_{11}^\sigma, \dots, LCR_{14}^\sigma), \dots, \min(LCR_{31}^\sigma, LCR_{32}^\sigma)) = 0.5$.

Multi-stage dependency example. Finally, consider an All-Reduce job partitioned into serialized Reduce-Scatter (Stage 1) and All-Gather (Stage 2) phases, as shown in Figure 18(c). Because Job 1 has the minimum rate, it gets the highest priority on all links. This means the scheduling of other flows on non-bottleneck links has no impact on Job 1. As a result, the overall job rate remains governed by a single overarching bottleneck across both stages, yielding $R_1^\sigma = \min(\min(LCR_{11}^\sigma, LCR_{12}^\sigma), \dots, \min(LCR_{51}^\sigma, LCR_{52}^\sigma)) = 0.2$.

E.3 Minimum Rate 1/2 Approximation

Let l_i^* and l_j^σ be the bottleneck links under the optimal policy and the σ -preserving policy, respectively. By Lemma 2 and Lemma 3, $R^* = LCR_{l_i^*}^*$ and $R^\sigma = LCR_{l_j^\sigma}^\sigma$.

We evaluate the approximation ratio based on two cases of the bottleneck link alignment:

- **Case 1** ($l_i^* = l_j^\sigma$): Applying Theorem 1 directly to the shared bottleneck link yields:

$$R^\sigma = LCR_{l_j^\sigma}^\sigma \geq \frac{1}{2} LCR_{l_j^\sigma}^* = \frac{1}{2} LCR_{l_i^*}^* = \frac{1}{2} R^*. \quad (31)$$

- **Case 2** ($l_i^* \neq l_j^\sigma$): By Theorem 1, $LCR_{l_j^\sigma}^\sigma \geq \frac{1}{2} LCR_{l_j^\sigma}^*$. Since l_i^* is the global bottleneck link under the optimal policy, we have $LCR_{l_j^\sigma}^* \geq LCR_{l_i^*}^*$. Combining these bounds yields:

$$R^\sigma = LCR_{l_j^\sigma}^\sigma \geq \frac{1}{2} LCR_{l_j^\sigma}^* \geq \frac{1}{2} LCR_{l_i^*}^* = \frac{1}{2} R^*. \quad (32)$$

In both cases, $R^\sigma \geq \frac{1}{2} R^*$ holds, completing the proof.

E.4 Max-min 1/2 Approximation

We establish the lexicographical max-min 1/2-approximation ratio via an iterative proof structure across four inductive steps:

- (1) **Identify bottleneck:** Locate the current global bottleneck link, and the jobs traversing it.
- (2) **Fix allocation:** Because these jobs have top priority, they face zero interference from other flows. We can therefore fix their transmission schedules and resource allocations.
- (3) **Apply bound:** By Theorem 1, these prioritized jobs are guaranteed to achieve at least a 1/2 approximation ratio relative to the optimal policy.
- (4) **Update and repeat:** Subtract the resources consumed by these fixed jobs from the network. On the remaining topology, use the residual resources to find and prove the bound for the second minimum, then the third minimum, and so on.

By resolving jobs sequentially in ascending order of their progress rates, this mathematical induction proves that σ -preserving achieves a global lexicographical max-min 1/2 approximation in general networks.

F Additional Evaluation Details

F.1 Testbed Topology

Figure 19 illustrates the logical topology of our physical testbed. The testbed consists of three GPU servers, each equipped with four GPUs and four NICs. Each NIC is connected to a dedicated logical leaf switch, resulting in five logical leaf switches in total. These leaf switches are implemented via virtual routing and switching on top of two physical Huawei CloudEngine 8875-24BQ8DQ switches.

The logical leaf switches are fully interconnected through the physical switches, forming a leaf-spine-like topology. All inter-switch links are rate-limited to 50 Gbps to emulate uniform bottleneck bandwidth across the fabric. This topology allows us to precisely control inter-job contention patterns while preserving the essential structural properties of a leaf-spine network.

By decoupling the logical topology from the underlying physical hardware, our testbed enables flexible emulation of multi-tenant communication contention scenarios and provides a faithful platform for evaluating network scheduling policies under controlled and reproducible conditions.

F.2 Testbed DLT Workloads

Table 1 summarizes the four representative LLM workloads used in our testbed evaluation. All models are publicly available and are obtained from ModelScope, ensuring reproducibility. We select these models to cover diverse parameter scales and communication-computation characteristics.

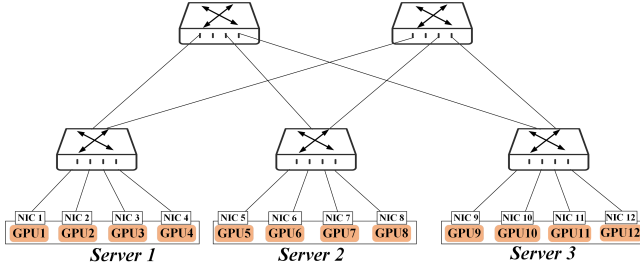


Figure 19: The topology of the testbed consisting of three servers. Each server is equipped with four Nvidia A100 GPUs.

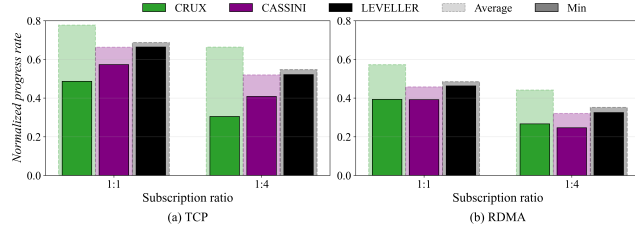


Figure 20: Physical testbed benchmark across different bandwidth oversubscription ratios.

Table 1: Detailed configurations of DLT workloads used in the testbed.

Model	Parameters	Parallelism
Llama-3.1-Nemotron-Nano-4B	~4B	FSDP
ChatGLM3-6B	~6B	FSDP
Gemma-2B	~2B	DDP
DeepSeek-Coder-1.3B-Instruct	~1.3B	DDP

Llama-3.1-Nemotron-Nano-4B [58]. This model is released by NVIDIA, comprises approximately 4B parameters. We train it on our testbed using FSDP, incurring fine-grained parameter sharding and frequent collective communication.

ChatGLM3-6B [89]. This model is developed by ZhipuAI and the Tsinghua KEG Laboratory and is a 6B-parameter bilingual large language model. We train it on our testbed with FSDP.

Gemma-2B [28]. This model is released by Google, comprising approximately 2B parameters. We train it on our testbed with DDP, where gradient synchronization occurs once per iteration, resulting in comparatively coarse-grained communication behavior.

DeepSeek-Coder-1.3B-Instruct [19]. This model is a code-oriented LLM with 1.3B parameters. It is also trained under the DDP paradigm and features computation-heavy iterations with relatively sparse communication phases.

Together, these workloads span a wide range of model sizes and communication-computation characteristics, enabling a comprehensive evaluation of scheduling policies under heterogeneous DLT workloads.

F.3 Jain’s Fairness Index

DEFINITION 7. Jain’s Fairness Index. To quantify the fairness of different scheduling schemes, we adopt Jain’s Fairness Index [37].

Given the normalized progress rates of N jobs, denoted by $\{R_1, \dots, R_N\}$, the index is defined as

$$F(R_1, \dots, R_N) = \frac{\left(\sum_{i=1}^N R_i\right)^2}{N \sum_{i=1}^N R_i^2}, \quad F \in \left[\frac{1}{N}, 1\right]. \quad (33)$$

A larger value of F indicates greater fairness in the normalized progress rates across jobs.

F.4 Benchmark on Bandwidth Oversubscription

To evaluate how contention intensity impacts scheduling efficiency and workload progress, we benchmark LEVELLER across various bandwidth oversubscription ratios on our physical testbed using both TCP and RDMA. As shown in Figure 20, performance degrades across all policies at 1:4 oversubscription, but LEVELLER consistently achieves the highest minimum progress rate. Compared to CASSINI, LEVELLER improves the minimum normalized progress rate by 18% under a 1:1 ratio and 30% under 1:4. Compared to CRUX, these improvements reach 35% and 73%, respectively. This demonstrates that higher bandwidth oversubscription ratios (and thus greater network contention) yield higher performance gains for LEVELLER. Moreover, LEVELLER maintains a higher average progress rate than CASSINI and remains only slightly below CRUX. These results confirm that LEVELLER robustly prevents job starvation under intense network contention with negligible efficiency loss.

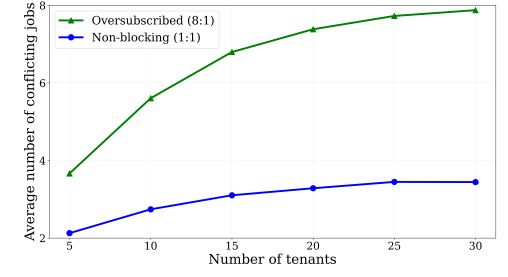
F.5 Benchmark on Conflicting Job Numbers

To investigate the impact of tenant scale and network topology, and to identify where LEVELLER’s scheduling benefits are most impactful, we conduct a simulation-based benchmark that analyzes two key factors: (i) the average number of conflicting jobs as tenant scale increases, and (ii) the performance of different schedulers on a single link as the number of concurrent jobs grows.

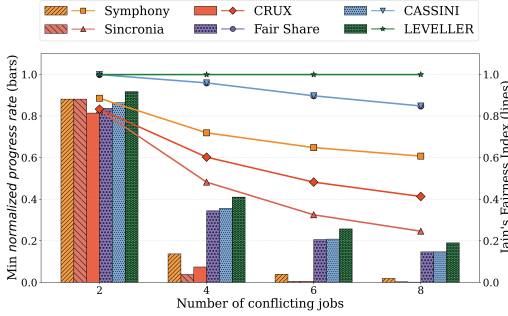
As shown in Figure 21(a), the average number of conflicting jobs on the bottleneck link increases with the tenant scale under both network topologies. The oversubscribed topology consistently leads to a significantly larger number of conflicting jobs than the non-blocking topology. Once the tenant population reaches a certain scale, the number of conflicting jobs tends to stabilize in both cases, since each link is associated with only a limited number of GPUs.

Figure 21(b) reports the performance of different schedulers as the number of jobs sharing a single link increases. As contention intensifies, both the minimum *normalized progress rate* and Jain’s Fairness Index decrease for all baseline policies. CRUX and Sincronia exhibit the most severe degradation, while CASSINI and Fair Share achieve comparable performance. In contrast, although the minimum rate of LEVELLER gradually decreases with increasing contention, its fairness remains largely stable, allowing it to consistently outperform all baselines. When the number exceeds four, the minimum progress rate drops to 0.5, and LEVELLER provides more than 20% performance gain. This demonstrates that as the number of conflicting jobs scales, the performance advantages of LEVELLER become increasingly significant.

When only two jobs share a single link, CASSINI and Fair Share achieve similar Jain fairness indices, while CASSINI achieves a higher minimum rate. This is because, in low-contention scenarios, job



(a) Average number of conflicting jobs under different tenant scales.



(b) Different schedulers' performance versus the number of jobs sharing a single bottleneck link.

Figure 21: Simulation-based benchmark on different numbers of conflicting jobs.

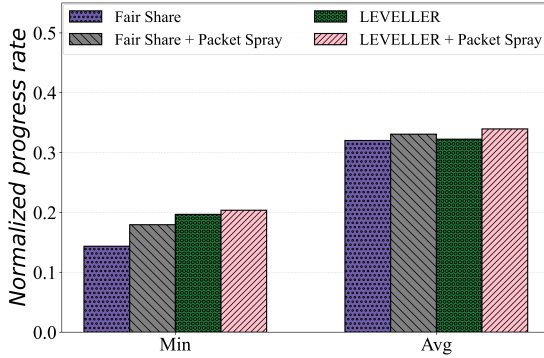


Figure 22: Comparison with packet spraying on minimum and average normalized progress rates.

compatibility is relatively high, enabling CASSINI to identify an appropriate time-shift offset that interleaves the communication phases of the two jobs. This observation highlights the distinction between fairness and max-min fairness. However, as the number of contending jobs increases, such time-shifting opportunities diminish due to intensified communication overlap, causing CASSINI's performance to gradually converge to that of Fair Share.

G Comparison with Packet Spraying

To investigate how LEVELLER interacts with load balancing mechanisms and how load balancing impacts LEVELLER's scheduling performance, we conduct an additional experiment with packet spraying in our simulator.

Setup. In this experiment, we reuse the trace-based simulation setup described in §6.2. In particular, we use the 8:1 oversubscribed topology with 30 tenants, while keeping all other configurations unchanged. We adopt Random Packet Spraying (RPS) [20], which randomly distributes individual packets across all available paths between a sender and a receiver. In our simulation, the maximum number of paths for packet spraying is set to 4, and packet reordering or specific hardware implementation details are not considered. We compare the following four configurations: (1) Fair Share with single-path ECMP, (2) Fair Share with RPS, (3) LEVELLER with single-path ECMP, and (4) LEVELLER with RPS. To integrate RPS, LEVELLER profiles T^{ideal} with the highest priority across all available paths and dynamically updates execution priorities along each path.

Figure 22 compares the minimum and average *normalized progress rates* across the four configurations. Packet spraying reduces network contention and significantly improves individual flow performance. However, DLT jobs heavily rely on collective communication, which includes a large number of flows (e.g., a 128-GPU ring AllReduce comprises 128/8 inter-host flows). Because the overall job is bottlenecked by its slowest link, a single congested flow can slow down the entire workload. Therefore, in our high-load scenario (30 tenants in an 8:1 topology), the gains from packet spraying for DLT jobs are less substantial than those observed for individual flows, yet still meaningful. As illustrated in the figure, combining Fair Share with packet spraying improves the minimum progress rate by 24% and the average progress rate by 3% over standard Fair Share. A deeper analysis reveals that standard Fair Share results in an average of 7.8 conflicted jobs on a single link, and introducing packet spraying increases the number to 19.4. However, because packet spraying distributes traffic across four paths, the flow volume is reduced to 1/4. Consequently, the effective average conflict number drops to approximately 4.85 (19.4/4), which explains the benefit of packet spraying.

Adopting LEVELLER does not change the underlying contention level (averaging 7.8 conflicted jobs without packet spraying and 19.4 with it), but provides a more optimized schedule that enhances both minimum and average progress rates. Specifically, LEVELLER alone improves the minimum and average progress rates by 37% and 1%, respectively, over standard Fair Share. When combined with packet spraying (LEVELLER + RPS), it yields a 13% improvement in the minimum progress rate and a 3% improvement in the average progress rate over Fair Share + RPS. The results demonstrate that the performance gain of LEVELLER persists in multi-path environments, but is smaller due to reduced network contention and narrower optimization headroom.

In conclusion, LEVELLER is complementary to existing load balancing mechanisms. Specifically, LEVELLER does not directly mitigate network contention itself; instead, it achieves a more optimized schedule that significantly improves fairness while also enhancing overall efficiency at the same contention level.