Efficient and responsive job-resource co-adaptivity for deep learning workloads in large heterogeneous GPU clusters

Suhas Jayaram Subramanya

CMU-CS-25-136

August 2025

Computer Science Department
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Thesis Committee:
Greg Ganger, Chair
Zhihao Jia
Virginia Smith
Amar Phanishayee (Meta Platforms, Inc.)

Submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy.

This work is supported in part by NSF Award #2211882 and by the U.S. Army Research Office and the U.S. Army Futures Command under Contract No. W911NF-20-D-0002. We thank the members and companies of the PDL Consortium (Amazon, Bloomberg, Datadog, Google, Hitachi, Honda, IBM, Intel, Jane Street, LayerZero Research, Meta, Microsoft, Oracle, Pure Storage, Salesforce, Samsung, Two Sigma, and Western Digital) and VMware for their interest, insights, feedback, and support. The content of the information does not necessarily reflect the position or the policy of the government and no official endorsement should be inferred.

Keywords: ML Cluster Scheduling, Large-scale Resource Allocation, Mixed-Integer Linear Programs, Job-Resource Co-adaptivity, Continual Optimization

To Mom and Dad.
I did something other than play games on the computer!

Abstract

Existing cluster schedulers face many limitations in scheduling adaptive deep learning training jobs on large heterogeneous GPU clusters – many are not heterogeneity-aware, few are adaptivity-aware, and none scale to large clusters without sacrificing allocation fidelity or cluster efficiency. Emerging clusters further complicate this problem — they will be larger, more heterogeneous, run more increasingly diverse jobs, and require optimizing more dimensions of adaptivity. It is very desirable to have a cluster scheduler that quickly and efficiently co-optimizes job allocations and execution parameters during runtime to maximize efficient use of expensive GPU resources. This dissertation develops new scheduling approaches and algorithms that can (1) scale to emerging clusters with hundreds of thousands of GPUs and many GPU types, (2) quickly optimize high-fidelity allocations for adaptive DL training jobs with low scheduler overhead, and (3) efficiently adapt to changing cluster conditions to improve goodput on the limited GPU resources.

We first introduce Sia—a round-based scheduler that efficiently optimizes adaptive jobs in a heterogeneous cluster with many GPU types. Sia uses GPU resources judiciously to gather information on job-GPU fit-levels using a mix of online and offline profiling, and continuously co-optimizes the GPU resources allocated to jobs and their execution parameters at runtime to maximize cluster-wide training progress. Using job traces derived from real-world datacenters, we find that Sia's allocations are fair and efficient, and are quickly computed using an efficient formulation, even for 1000-GPU clusters.

Second, we observe that schedulers whose policies are formulated as constrained optimization problems must sacrifice cluster efficiency for responsive scheduling at larger scales. This arises due to a disconnect between the per-round modeling of scheduling problems as independent optimization problems and the slow evolution of these problems at large-scale as a result of job and resource changes, leading to solvers bottlenecking schedulers at scale. We introduce *continual optimization* — a new paradigm that explicitly models the slow evolution of resource-allocation problems at scale to reduce solver runtime for quick responses to changes in jobs or resources. We then introduce `COpter`, our approach to continual optimization that (a) efficiently updates the optimization problems for job and resource changes using a differential interface, (b) implements a factorization-free warm-started LP solver to benefit from the slowly-evolving nature of the allocations, and (c) implements lightweight heuristics to recover feasible integral solutions with negligible quality loss. In our evaluations, `COpter` speeds up Sia scheduler policy by a few orders of magnitude on clusters with tens of thousands of GPUs without sacrificing job completion times and makespan.

Third, `COpter` is easily applied to resource-allocation problems in other domains (e.g. shard load-balancing, WAN traffic engineering) and we see $57 - 83\times$ reductions in solver runtimes. Compared to problem partitioning approaches (POP), `COpter` simultaneously improves allocation quality and reduces end-to-end allocator runtimes by $1.5 - 30\times$.

Acknowledgements

First and foremost, I want to thank my advisor, Greg Ganger. His wisdom, patience, and kindness have meant so much to me, and I feel truly fortunate to have learned from him. I always looked forward to our meetings, even when I had little to report, because they were full of insight and encouragement. Greg's support in letting me explore bold ideas shaped much of this dissertation, and his attention to detail has deeply influenced how I approach research. I am especially grateful for the way he pushed me out of my comfort zone, with frequent presentations at PDL meetings and retreats that built my confidence in presenting research. As I graduate and move into industry, I look back on his mentorship with gratitude and hope to be the kind of mentor Greg has been to me.

I would also like to thank the other members of my thesis committee — Zhihao Jia, Virginia Smith, and Amar Phanishayee — for their patience, guidance, and feedback on my research. Zhihao, your insights were invaluable in helping position our `Sia` work more effectively. Ginger, your thoughtful feedback and masterful writing greatly strengthened our `COpter` work, and Amar, your wisdom and perspective helped ground my research in the needs of industry.

I am deeply grateful to my collaborators — Aurick Qiao, Saurabh Kadekodi, Daiyaan Arfeen, and Shouxu Lin — for their dedication and hard work in producing high-quality research together. I especially want to thank Aurick and Saurabh for their patience and for sharing much of their 'senior PhD' wisdom with me in my early years.

I feel very fortunate to have met such wonderful people at CMU who made my time here truly memorable through office banter, events, hangouts, and above all, great company: Praneeth Kacham, Lucio Dery, Asher Trockman, Sarah McAllister, Timothy Kim, Sanjit Athlur, Nirav Atre, Hojin Park, Nj Mukherjee, Wan Shen Lim, Laxman Dhulipala, and Hugo Sadok. I owe special thanks to the amazing staff at the Parallel Data Laboratory (PDL) — Karen Lindenfelser, Joan Digney, Jason Boles, Chad Dougherty, and Bill Courtright — for their constant support. I have been fortunate to rely on your invaluable help with both logistics and technical matters throughout the years. I would also like to thank Debra Reich and Danny Sharara for helping me navigate the difficult moments along the way.

I am thankful to my friends, near and far, whose presence has made these years so memorable and full of warmth: Aditya Pratapa, Akshay Balaji, Ankush Jain, Anup Agarwal, Austin Shaw, Chirag Gupta, Chirag Pabbaraju, Jovina, Manu Halvagal, Mazhar Shaikh, Nikita Yadav, Pablo Bhowmik, Purva Bhumkar, Sai Praveen, Sanjith Athlur, Shreyas Harish, Sooraj Reddy, Suvadip Paul, and Vivek Tejamurthy. I also want to thank the *bois* — Aditya Philip, Aditya Raut, Anish Sevakari, Boby Robert, Don Dennis, and Nithin Ramachandrappa — for the innumerable hours of gaming, hangouts and banter that remain a highlight of my time in Pittsburgh. Though life has taken us to different cities, I will always treasure our time together and look forward to hanging out in VCs on Discord every day.

My six years at CMU have had their ups and downs, but my family's support has always been unwavering. Ma, I am so grateful for our daily breakfast chats—no matter how slow research progress was or how many dead ends I hit, talking to you always made me feel that things would turn out okay. Dad, I may not have been able to explain my research clearly,

but thank you for always encouraging me and pushing me to do better. Suman and Likhitha, thank you for always being there and letting me vent about research and life, even from afar. Finally, I want to thank the love of my life — and the undisclosed *fifth author* on all my research — Sushma, for being my rock through every joy and challenge. Even 8,000 miles apart, our daily calls made the distance feel small, and I always knew I could see you the next day. I look forward to the next chapter of our lives as we finally close that distance together.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Deep Learning (DL) training has grown to become a staple datacenter workload in recent years. Deep learning models are trained on sizeable clusters with powerful accelerators (GPUs, TPUs, etc) that are often shared among multiple users for cost-efficiency reasons, thus necessitating the use of a *cluster scheduler* to allocate resources to training jobs. In recent years, DL clusters have grown to contain tens of thousands of accelerators[35], each with unique trade-offs that affect completion time of DL training jobs in distinct ways[79]. The problem of assigning the *right* resources to DL training jobs is further complicated by an increasing state-space of *adaptivity* choices — the number, type and arrangement of GPU resources and training parameters for a DL training job (e.g., training batch size, learning rate, etc.). We categorize these adaptivity choices into two classes — *resource-adaptivity* where the count and type of GPU resources allocated to jobs are adapted in response to cluster conditions (i.e., job arrivals, completions, and phase changes), and *job-adaptivity* where the execution parameters for a training job are adapted at runtime to maximize training progress on allocated resources (i.e., choosing best batch-size and learning rate for the allocated GPUs).

A DL cluster scheduler must choose the right adaptivity choices for each DL training job to minimize the average job completion time for a given job mix. However, the choice of adaptivity for any given job is also influenced by the amount of training progress already made[79] by the training job and the adaptivity choices made for all other jobs in the cluster. As a result, the cluster scheduler must optimize adaptivity for all jobs together in the cluster to use limited expensive GPU resources efficiently.

The optimal choice of adaptivity for each job is also affected by changes to the cluster composition (i.e., the set of jobs and resources). Job arrivals and resource failures increase competition for the limited GPU resources, and similarly, job completions and new resource deployments reduce contention. Since training progress also affects the choice of adaptivity [79], the optimal choice of adaptivity for each job also changes over time. As a result, the cluster scheduler must frequently re-optimize each job's adaptivity to maintain high cluster efficiency and reduce job runtimes.

An efficient and responsive cluster scheduler for modern DL cluster schedulers must address the following challenges:

- **Heterogeneity-awareness**: New GPU resources are deployed incrementally into

existing GPU clusters, often resulting in newer GPU types being added over time. A GPU cluster scheduler must be heterogeneity-aware and take into account the various trade-offs that result from differing capabilities across GPU types when deciding job allocations.

- **Adaptivity-awareness**: Users may request the scheduler to optimize one or more adaptivity choices during job submission (i.e., some may want job but not resource adaptivity), and the GPU cluster scheduler must support such requests.

- **Low scheduler overhead**: Scheduler overheads arise from multiple sources — (a) setup and teardown costs from preemption, migration and/or adaptation actions, and (b) profiling overheads incurred in learning performance characteristics of each job on all possible allocation choices. An efficient GPU cluster scheduler must minimize scheduler overheads so it can use as many GPU resources as possible towards completing jobs.

- **Quick response to changes in cluster**: As discussed before, clusters undergo changes frequently from job arrivals/ departures, resource failures/deployments and job phase changes. The GPU cluster scheduler must respond to these change events quickly by (a) re-optimizing job allocations in response to job arrivals/departures, and (b) recovering from software and hardware failures without losing training progress. Round-based schedulers can achieve quick response times by invoking the scheduler once every few minutes.

- **Scalable scheduling**: DL clusters now span over 100,000 GPUs (up from a few thousand in 2019 [43]) with a corresponding increase in users and jobs requesting these resources. Cluster schedulers must operate efficiently at these scales without any tradeoffs in resource utilization or job completion times.

Existing state-of-the-art GPU cluster schedulers for DL training workloads fall short of addressing some or all of the above challenges: for example, adaptivity-aware schedulers are not heterogeneity-aware [79] and heterogeneity-aware schedulers are not adaptivity-aware [68]. Complex scheduler policies that address one or many adaptivity dimensions can be formulated as optimization problems [111, 68, 79] solvable by off-the-shelf numerical solvers, but are bottlenecked by the solver runtime for clusters with thousands of GPUs. This dissertation identifies two crucial bottlenecks that preclude efficient solutions from addressing the above challenges all at once:

- *State space explosion*: It is impractical to explore every possible choice of adaptivity for each job: the number of possible allocations and their mappings to physical resources is exponential in the number of GPUs, further compounded by the choice of GPU type and job execution parameters for each job.

- *Solver time explosion*: Scheduler policies are formulated as optimization problems using one variable for each allocation choice for each job. For responsive scheduling in large clusters, this means solvers must find a feasible solution to problems with tens of millions of variables within a few minutes.

This dissertation introduces techniques to manage the state-space and solver time explosions by discovering and exploiting structure and redundancy in (a) state-space of adaptivity choices and (b) scheduling problems across time.

Thesis statement. *It is possible to develop powerful cluster schedulers for DL training jobs running on large heterogeneous GPU clusters that can run frequently and quickly, and adapt training jobs and their resource allocations efficiently with low overheads.*

This dissertation consists of three parts – first, we resolve the bottleneck of state-space explosion by developing a scheduler (Sia) to efficiently adapt DL training jobs on heterogeneous GPU clusters with low per-job overhead, and second, we resolve the bottleneck of solver-time explosion by introducing continual optimization to exploit the *slowly-evolving* nature of the scheduling problems (`COpter`) and their optimal allocations at scale, and third, we observe that many other resource-allocation problems of interest and their optimal solutions are also slowly-evolving and can be solved quicker using continual optimization as implemented in `COpter`.

The first part of this dissertation proposes a design and implementation for a scheduler capable of co-optimizing multiple dimensions of adaptivity for each training job to maximize cluster-wide training progress in heterogeneous GPU clusters. We show that Sia— our scheduler — can learn DL training jobs' performance on many GPU types to quickly and efficiently optimize job-resource co-adaptivity in heterogeneous GPU clusters with a few thousand GPUs. Through extensive evaluations, in a physical cluster and in simulation, we show that Sia significantly improves training performance (job runtime, GPU efficiency, fairness) in heterogeneous GPU clusters with minimal scheduler overheads. Sia policy formulated as an Integer Linear Program (ILP) scales to clusters with a few thousand GPUs with a runtime of just a few seconds. However, since emerging GPU clusters for DL workloads are expected to span hundreds of thousands of GPUs, Sia runs into the same bottlenecks as existing optimization-based schedulers – the time taken to optimize the scheduler policy bottlenecks the efficacy of the scheduler.

The solver runtime bottleneck affects many state-of-the-art round-based schedulers formulated as Linear/Mixed-Integer Linear Programs (MILPs). We find that this is because traditional approaches to round-based scheduling solve the scheduling problem in each round independently from scratch, thereby discarding all computational effort spent in solving prior scheduling problems to obtain optimal allocations.

The second part of this dissertation introduces *continual optimization*, a new paradigm for round-based scheduling. Continual optimization exploits the observation that scheduling problems at larger scales often evolve slowly between rounds — relative to the size of the cluster and current set of jobs, few jobs enter and exit the system, and few resources fail or are added back. Additionally, as a result of these changes, the scheduler changes allocations for a few jobs and retains allocations for most jobs between rounds. Continual optimization aims to exploit this *slow evolution* in the structure and solution present in scheduling problems to resolve the solver-runtime explosion bottleneck that precludes efficient and responsive scheduling in clusters with tens of thousands of GPUs.

This dissertation develops `COpter`— an approach to continual optimization of cluster scheduling policies formulated as LP/MILPs. `COpter` addresses scalability challenges in optimization of scheduling policies through three key innovations. First, it uses a differential interface to efficiently update the mathematical representation of the scheduling problems as

they slowly evolve over time from changes in the set of GPU resources and jobs requesting them. Second, it uses a factorization-free, warm-started LP solver that provably benefits from solution proximity — by starting the solving process for the current round's scheduling problem from the previous round's optimal allocations, it benefits from most jobs' allocations persisting across rounds. Third, it employs lightweight heuristics to quickly find high-quality integer solutions for MILPs, completely bypassing the expensive combinatorial search phase employed in state-of-the-art MILP solvers [29, 60]. `COpter` finds high-quality allocations within a minute for the Sia scheduler policy in clusters with tens of thousands of GPUs and many GPU types, while commercial state-of-the-art MILP solvers fail to scale beyond a few thousand GPUs. `COpter` improves over POP, a state-of-the-art problem partitioning approach, on both solution quality (i.e., lower average job completion times and makespan) and solver times.

The third part of this dissertation finds that many resource-allocation problems from other domains, like shard-load-balancing in elastic database systems [85] and traffic engineering in Wide-Area-Networks (WAN) [102, 2], are formulated as LP/MILPs, evolve slowly over time, and finding the optimal solution to these problems faces solver bottlenecks for large numbers of resources and requests. Similar to Sia, scaling up these problems sacrifices allocation quality for manageable solver runtime. We find that applying continual optimization to these problems using `COpter` provides us similar quality solutions as those found by state-of-the-art numerical solvers, but with $57 - 83\times$ faster runtimes.

Chapter 2

Background

In this chapter, we will go over some necessary background required to understand our work in scheduling and optimization of scheduling policies.

2.1 Deep Learning training

Deep learning. Deep learning is a class of machine learning methods that learn a function $f(x;\theta)$, where $f(x;\theta)$ is a deep neural network (DNN) comprising of many *layers*, and where θ is the set of parameters for each layer. Each layer is a function that operates on the output of a previous layer, to produce the input for the next layer. The function for the i^{th} layer can be written as $f_i(x; A_i, b_i, \sigma_i) = \sigma_i(A_i x + b_i)$ where (A_i, b_i) are the parameters and σ_i is the *activation* function for the i^{th} layer. Then, the output of the DNN can be computed as a composition of the layer-wise functions: $f = f_1 \circ f_2 \ldots \circ f_n$. The parameters of f is defined as the list of parameters for each layer: $\theta = [(A_1, b_1), (A_2, b_2), \ldots (A_n, b_n)]$. For example, in image classification, x can be an input image, and $\hat{y}$ could be the probability that the image contains an object belonging to some class (e.g. a bird, or an animal).
Deep learning training. Deep learning training is the process of learning the parameters of the function $f(x;\theta)$, using a *dataset* – examples of the form (x, y) where x is the input to the model and y is the expected output. ML practitioners specify a *loss* function – a measure of error between the expected output y and the output of the model for the input x, $\hat{y} = f(x;\theta)$, denoted by $\mathcal{L}(x, y, \theta)$ for an example (x, y) in the training dataset. Gradient-descent is then used to obtain the parameters θ that minimize the average loss on the training dataset using the gradient of the loss function w.r.t. the model parameters $\nabla_\theta \mathcal{L}(x, y, \theta)$. Gradient-descent uses the following simple update rule:

$$\theta_{k+1} = \theta_k - \alpha * \sum_{i=1}^{i=N} \nabla_\theta \mathcal{L}(x_i, y_i, \theta_i) \tag{2.1}$$

where α is the learning rate, θ_k, θ_{k+1} are the model parameters after k and $k+1$ iterations of gradient-descent, respectively, (x_i, y_i) is the i^{th} example, and N is the number of examples in the training dataset.
Stochastic gradient descent (SGD). Deep learning models are typically trained using datasets with millions to billions of examples, and using all examples to compute the gradient

in Equation (2.1) is inefficient for a single update to θ. Stochastic gradient descent (SGD) uses a random subset of the training dataset to approximate the gradient. The size of this subset is termed the *minibatch* size (represented by B), and the update rule is accordingly modified as follows:

$$\theta_{k+1} = \theta_k - \alpha \sum_{i=1}^{i=B} \nabla_\theta \mathcal{L}(x_i, y_i, \theta_i) \tag{2.2}$$

Parallelism in DL training. DL training can be parallelized across multiple GPUs in many ways. Data-parallelism replicates the model parameters on each GPU and splits the gradient-computation work on the mini-batch (i.e. data) axis. Data-parallelism (DP) consists of two distinct phases:

- *Compute* – Each GPU computes gradients for the model on the subset of the minibatch assigned to it
- *Communication* – All GPUs synchronously exchange their computed gradients using *all-reduce* to realize the average minibatch gradient on each GPU

After the communication phase, every GPU has a local copy of the average gradient on the minibatch and a local copy of the model parameters. Equation (2.2) is then used to update the local model parameters and this completes one iteration of SGD.

Another form of parallelism partitions the model parameters and gradient computation along the parameter axis. This style of parallelism is more attractive when a model is too large to fit into the limited memory of a single GPU, and as a result, data-parallelism (which requires a copy of the model on each GPU) cannot be used to scale-out training. Pipeline model parallelism partitions the layers and their associated parameters, with each GPU performing gradient computation only on the subset of the layers assigned to it. Yet another form of parallelism (tensor-parallelism) partitions the individual layer computations across multiple GPUs by exploiting the massive parallelism available in the matrix-multiplications for each layer. Large-scale training often requires using more than one style of parallelism (*hybrid-parallel*) to scale training to thousands of GPUs. The optimal combination of parallelism techniques depends on the model size (number of parameters $\times$ the byte-size of a parameter), GPU memory capacity, inter-GPU communication bandwidth and topology, and inter-node communication bandwidth and topology. In this document, we will focus primarily on data-parallelism for models that fit into the memory of a single GPU, and hybrid data + pipeline model parallelism for models that do not.

Performance metrics. Performance of a DL training job is typically measured using *throughput* – number of examples processed per second. For a fixed number of GPUs, increasing the minibatch-size B (up-to GPU memory limits) also increases the training throughput as parallelism within and across GPUs is exploited more efficiently, resulting in higher GPU utilization (i.e. the fraction of time the GPU is active and performing compute). For a fixed minibatch size B, increasing the number of GPUs (*strong-scaling*) gives us diminishing returns: while the added GPUs parallelize the compute phase better, they also increase the communication time. Instead, if we increase the minibatch-size proportionally with the number of GPUs (*weak-scaling*), throughput increases without sacrificing GPU

utilization. However, increasing the minibatch size also reduces the *statistical efficiency*[79] – the amount of training progress made per sample processed (i.e. minibatch training progress / minibatch-size). Say we scale out training to 8x GPUs and also scale the minibatch size by 8x. Then, the training throughput goes up ≈8x, but the model does not converge to the same quality in 8x fewer SGD iterations, resulting in a sub-linear scaling of training-time w.r.t. number of GPUs used. We adopt the definition of statistical efficiency as defined by Pollux[79] (based on *simple* noise-scale[61]). Statistical efficiency at a batch-size B relative to a baseline batch-size B_0 is defined as follows:

$$\text{statistical-efficiency}(B, B_0) = \frac{\phi + B_0}{\phi + B} \tag{2.3}$$

where ϕ is the preconditioned gradient noise scale (PGNS) as defined in Pollux [79].Equation (2.3) implies that using a batch-size of B makes about statistical-efficiency(B, B_0) fraction of the progress made with a single batch of size B_0. For a fixed batch-size B, statistical efficiency is also a function of time t [79].

Goodput. Pollux [79] introduces the notion of *goodput* to capture the trade-off between batch-size B and training throughput. Goodput is defined as the rate of progress per example processed, and is computed using the following expression:

$$\text{GOODPUT}(B, N) = \text{statistical-efficiency}(B) \times \text{throughput}(B, N) \tag{2.4}$$

where B is the batch-size and N is the number of GPUs. For a fixed set of resources, the batch-size B that maximizes Equation (2.4) makes the most training progress per unit time. This *optimal* batch-size, however, is not constant throughout the lifetime of a training job because the statistical efficiency for a given batch-size changes as the model training progresses. As a result, the batch-size that produces the highest goodput for a fixed set of resources also changes over time.

2.2 Cluster scheduling for deep learning training jobs

Large clusters for DL training are typically shared between many users and/or organizations for cost-efficiency and utilization reasons, and a cluster scheduler allocates GPU resources to the DL training jobs in the cluster. Cluster schedulers typically aim to reduce the average job completion time (JCT) – the delay between submitting a training job to the scheduler and completion of the model training. The cluster scheduler runs at discrete time intervals – called *scheduling rounds* – and in each scheduling round, it uses a *scheduler policy* to determine the allocations for each job in the scheduler queue.

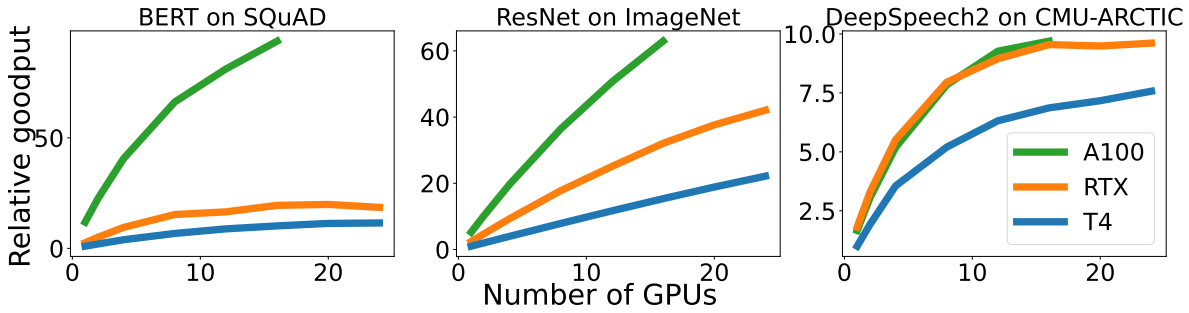


Figure 2.1: Scaling of goodput with number of GPUs for different GPU type and training job combinations. For each job type, goodput is shown relative to goodput obtained on one NVIDIA T4 GPU. See [42] for more details.

Characteristics of DL training jobs. DL training jobs have the following unique characteristics that combine with DL cluster expense to necessitate the use of custom schedulers:

- **Predictable performance**: SGD iterations using a fixed batch-size complete in approximately the same time without any dependency on the content of the batches for the iterations. So, the runtime of a single iteration with a batch-size B is predictable at any point in the job's lifetime if it has been measured at least once.
- **Preemptible computation**: DL training jobs are preemptible – they can be interrupted once they start running and can be resumed at a later point on (potentially) different resources. Preemption is often performed at SGD iteration boundaries, and a checkpoint consisting of the model parameters, and optimizer and dataset states is sufficient to resume training without any loss in progress[79].
- **Elastic computation**: DL training jobs are also elastic – they can be scaled up/down using many forms of parallelism (e.g. data and pipeline model parallelism as discussed in the previous section).
- **Fungible resources**: DL training requires accelerating tensor, matrix and vector operations and are easily executed on many GPU types with minimal code changes. As a result, a job that runs on one GPU type (e.g. NVIDIA A100) can also (typically) run on another GPU type (e.g. NVIDIA H100).
- **Heterogeneity in training performance**: Performance of a DL training does not scale in a linear fashion with added GPUs, and the scaling behaviour depends on many

factors – GPU type, model type, networking bandwidth and topology. Figure 2.1 shows the scaling of goodput (i.e. rate of training progress) for different GPU types and jobs. Jobs like BERT training ([Left] in Figure 2.1) scale better on one GPU type whereas jobs like DeepSpeech2 training ([Right] in Figure 2.1) scale similarly on different GPU types.

Heterogeneity in GPU clusters. Large GPU clusters are typically composed of GPUs from many generations as newer GPUs are deployed incrementally into existing clusters for improved capacity, performance and/or efficiency. GPUs exhibit heterogeneity along many dimensions:

- **Compute** – hardened accelerator blocks (e.g. transformer units, tensor cores, etc), support for specific data-types (e.g. FP16, FP8, INT8, FP4, etc) and core clock frequencies (NVIDIA's SXM GPUs clock higher than their PCIe variants)
- **Memory** – memory type, capacity (e.g. NVIDIA A100 has 40GB whereas a NVIDIA H100 has 80GB of GPU memory), and bandwidth (A100 has 2TB/s and H100 has 3.4TB/s of memory bandwidth)
- **Network** – intra-node GPU P2P interconnect bandwidth and topology(NVIDIA NVLink is 10 times faster than PCIe), inter-node bandwidth and topology (Infiniband vs Ethernet)

Considerations in DL cluster scheduling. A DL cluster scheduler must consider all dimensions to GPU heterogeneity when deciding allocation of heterogeneous GPU resources to DL training jobs. Additionally, the placement for a job also impacts its performance since data-parallel training is sensitive to the problem of *straggler* GPUs – where a single slow GPU can stall all GPUs waiting on a distributed synchronization barrier. Thus, it is important to choose the best placement for distributed training jobs to maintain high average GPU utilization and low job runtimes in the cluster.

It is also important to frequently re-optimize allocations in a cluster as the optimal allocation for each job depends on the cluster state that is in a constant flux due to (a) changes in statistical efficiencies for running jobs, (b) new jobs arriving into the cluster (necessitating scale-in decisions), and (c) running jobs completing and freeing up their allocated GPUs (necessitating scale-out decisions). We use the term *resource-adaptivity* to describe adaptation of the GPU resources allocated to a job, and *job-adaptivity* to describe the adaptation of the job training parameters (for example, its training batch-size). So, an ideal scheduler should continuously optimize both resource and job adaptivity to maintain high efficiency in the cluster.

2.3 Constrained optimization

Many scheduling policies can be formulated as constrained optimization problems and can be solved using off-the-shelf numerical solvers like GLPK [60], Gurobi [29], and CPLEX [39]. Constrained optimization is fundamental to optimizing scheduling policies that are formulated as optimization problems. A constrained optimization problem finds an assignment of values to variables x that minimizes an objective function $f(x)$ (or maximizes $-f(x)$) subject to

some constraints on the variables $\mathcal{C}$. For the rest of this document, we restrict ourselves to constrained optimization problems where both f and $\mathcal{C}$ are linear in the variables x (i.e.*linear programs*). We are interested in solving problems with the following formulations:

$$\min_x \; f(x) = c^T x \quad \text{subject to} \quad Ax \leq b \tag{2.5}$$

where $x \in \mathbb{R}^n$ is a vector of variables being optimized, $c \in \mathbb{R}^n$ is a vector of *costs*, $A \in \mathbb{R}^{m \times n}$ is the constraint matrix, and $b \in \mathbb{R}^m$ is the constraint vector. Additionally, for problems called *Integer Linear Programs*, x could be constrained to take on only integral values (i.e. $x \in \{\{0\} \cup \mathbb{N}\}^n$), or just binary values (i.e. $x \in \{0, 1\}^n$).

Equation (2.5) is often reformulated to express the inequality-constraint as an equality-constraint using *slack* variables $s \in \mathbb{R}^m$. This leads to the following equality-constrained formulation:

$$\min_x \; f(x) = c^T x \quad \text{subject to} \quad Ax + s = b \tag{2.6}$$

We can redefine Equation (2.6) to contain only one set of variables $z = [x \quad s]$ in the following manner:

$$\min_z \; \hat{f}(z) = \hat{c}^T z \quad \text{subject to} \quad \hat{A}z = b \tag{2.7}$$

where $\hat{c} = [c \quad 0]$, $\hat{A} = [A \quad \mathbb{I}_m]$ and $\mathbb{I}_m$ is the square identity-matrix with m rows.

2.3.1 Optimizing linear programs

There exist many methods to optimize linear programs formulated in Equation (2.7). We briefly summarize a few methods below:

- **Simplex methods**. Simplex methods solve Equation (2.7) in an iterative manner. They operate on the feasible region defined by the constraints, moving along the edges of the polytope (defined by the constraints) to find the optimal solution. The method starts from an initial feasible solution and iteratively improves the objective value by transitioning to adjacent polytope vertices with higher objective values until the optimum is reached. Simplex methods are efficient in practice and widely used in commercial solvers like CPLEX[15] and GUROBI[29].
- **Penalty Methods**. Penalty methods convert Equation (2.7) into a sequence of unconstrained optimization problems with the following objective:

  $$\min_x f(x) + \gamma \cdot g(Ax - b) \tag{2.8}$$

  where g is a penalty function and γ is the strength of the applied penalty. The solutions to these problems eventually converge to the optimal solution of the original constrained problem. They are, however, not commonly used in large-scale optimization.
- **Interior-Point (barrier) methods (IPM)**. Similar to penalty methods, IPM algorithms solve a series of unconstrained optimization problems, each of which produce a feasible solution that is *strictly* in the interior of the constraint set. IPM algorithms are widely implemented in common optimization packages like GUROBI[29] and MOSEK[5].

- **Augmented Lagrangian methods (ALM)** Like Penalty methods and IPM algorithms, ALM algorithms[78] also solve a sequence of unconstrained problems whose solutions eventually converge to the optimal solution for the original problem. ALM algorithms *augment* the Langrangian function for Equation (2.7) using a squared L2 norm penalty, resulting in the following unconstrained objective:

$$\min_x f(x) + \lambda^T(Ax - b) + \frac{1}{2\mu} \|Ax - b\|_2^2 \tag{2.9}$$

  where $\lambda \in \mathbb{R}^m$ is *dual* variable associated with the constraint $Ax = b$, μ controls the penalty for constraint violation, and $\|\cdot\|_2^2$ is the squared L2 norm operator. Practical large-scale ALM solvers include LANCELOT[14] and QPALM[32].
- **Alternating Direction Method of Multipliers (ADMM)**. ADMM[9] is a recent class of algorithms that follows the ALM framework with a lower per-iteration cost. ADMM algorithms are well suited for distributed optimization[103], but are typically slower to converge to a high-quality quality solutions quickly[91, 84].

Chapter 3

Sia: Heterogeneity-aware goodput-optimized ML cluster scheduling

In this chapter, we describe Sia– a powerful cluster scheduler for DL training jobs – capable of co-adapting a training job's hyper-parameters with the resources allocated to maximize concurrent job progress in a shared heterogeneous GPU cluster. Sia efficiently explores the large state-space of heterogeneous allocations and matches jobs to GPU counts and types using minimal profiling. Our results indicate Sia's decision making scales to thousands of GPUs and many GPU types, and its scheduling policy improves job runtimes through fair and efficient allocations in a heterogeneous cluster. Sia was published in 2023 at the ACM Symposium on Operating Systems Principles (SOSP)[42]. In the following sections, we will briefly go over the motivation for a scheduler (and the accompanying runtime) like Sia, its design and scheduler policy, and finally, conclude with a summary of evaluation of Sia in various settings.

3.1 Introduction

Sizable deep learning (DL) clusters, often shared by multiple users training deep learning models for different problems, have become data center staples. A scheduler is used to assign cluster resources to submitted jobs. Increasingly, DL clusters consist of a mix of GPU types[1], due to incremental deployment over time and advances in GPU design.

Recent work provides powerful schedulers for DL clusters, but none utilize heterogeneous DL clusters well. To help explain why, we partition existing schedulers into two categories. *Heterogeneity-aware* schedulers [68, 100, 56] explicitly consider differences among GPU types in the cluster, with Gavel [68] as a state-of-the-art example, but existing options only accommodate what we term *rigid* jobs. ("Rigid" jobs must run with a user-specified number of GPUs, do not allow elastic scaling, and do not adapt to resource assignments.)

[1]For conciseness, we will use "GPU" to refer to any accelerators used for DL model processing generally, including both traditional GPUs and various others like TPUs [48], FPGAs, and other ML accelerators [55, 44, 7].

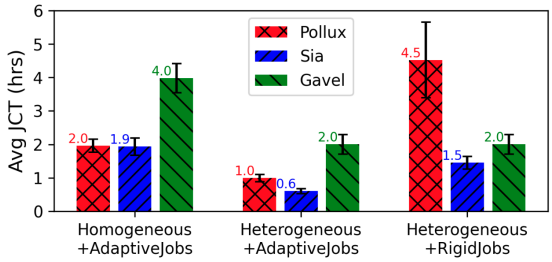


Figure 3.1: Scheduler comparison for three scenarios. [Left] For resource-adaptive (non-rigid) jobs on a homogeneous cluster, the left-most bars show that Pollux and Sia yield lower average job completion times (JCTs) than Gavel. [Right] For rigid jobs on a 3-GPU-type heterogeneous cluster, on the right, Gavel and Sia outperform Pollux. [Center] For non-rigid jobs *and* heterogeneous resources, in the middle, Sia outperforms both state-of-the-art schedulers built for only one of the two complexities. (The trace and cluster configurations are detailed in Section 3.4; the heterogeneous cluster includes some faster GPUs, causing JCTs to decrease for all schedulers.)

Adaptivity-aware schedulers [75, 79, 88] explicitly consider how non-rigid jobs would adapt to (e.g., batchsize adjustments) and perform with different numbers of GPUs, with Pollux [79] being a state-of-the-art example, but existing options assume that the cluster's GPUs are all the same type.

Figure 3.1 illustrates the resulting problem. When only one degree of freedom (heterogeneous GPUs or adaptive jobs) is present, a state-of-the-art scheduler for addressing it provides good performance. But when both are present, much opportunity is lost (see 40–70% lower average JCTs in the middle trio of bars) because existing schedulers do not consider both. Worse, for more intense workloads the gaps grow larger (e.g., see Figures 3.6 and 3.10), because these schedulers scale poorly with contention (Gavel) and cluster size (Pollux).

Sia is a new scheduler designed for *resource-adaptive* DL training jobs **and** *heterogeneous* resources, matching each state-of-the-art for their category but outperforming them when both degrees of freedom are present. Conceptually, in each scheduling round, Sia considers every possible assignment of GPUs (number and type) to current jobs, estimates their aggregate "goodput"[2] (including any job resizing costs), and selects the best cluster resource assignment for the next period of time. This is challenging for two fundamental reasons: (1) the search space is huge, for a sizable cluster, and much worse when there are multiple GPU types and each job can use and adapt to any number of GPUs of any type; (2) different DL jobs experience different performance changes when comparing one GPU type to another, when increasing the number of GPUs (i.e., one may scale better than another), and when comparing scaling with one GPU type to scaling with another (e.g., different GPU types can have distinct compute-to-network-bandwidth ratios), and yet profiling each DL job for all possible resource allocations is prohibitively expensive.

Sia addresses these challenges with a new solver formulation to deal with scale and a new approach to online learning of per-job per-GPU-type throughput models. Sia's new ILP formulation, together with pragmatic search space reductions, allows it to efficiently find assignments of GPU types, GPU counts, and batchsizes for all pending jobs even as load and cluster size grow. Sia's new approach to throughput modeling (as a function of GPU type,

[2]"Goodput" [79] is a DL efficiency metric that combines sample-processing throughput and statistical efficiency to reflect *rate* of training progress.

GPU count and batchsize) avoids extensive profiling, which could override scheduling benefits. Instead, Sia bootstraps each new job's throughput model with profiles of just one minimum-sized[3] configuration per GPU type, initially assumes simple scaling/projection across as-yet-unknown configurations, and dynamically refines the model as different configurations are used for the job. Experiments confirm that Sia's approach yields good decisions with low profiling overhead.

Extensive evaluations with workloads derived from three real cluster environments show Sia's effectiveness, scalability, and superiority to three state-of-the-art schedulers (Pollux, Gavel, and Shockwave [111]), as well as others. Sia is implemented as a plugin-compatible scheduler replacement in the open-source AdaptDL framework [40], allowing us to perform head-to-head comparisons with the public Pollux implementation. Experiments with Sia, Pollux, and Gavel on a 44-GPU 3-GPU-type cluster show that Sia provides 35% and 50% lower average JCTs (avgJCT) than Pollux and Gavel, respectively. Importantly, these experiments also re-validate the simulator from [79], which we use for broader explorations, including larger clusters than we can obtain and more intense workloads. Indeed, we find that Sia's advantages grow with cluster load/contention, especially compared to Gavel (up to 95% lower avgJCT) and Shockwave (up to 47% lower avgJCT), which treat all jobs as rigid.

Overall, the results show that, for adaptive jobs on a heterogenenous cluster, dynamically adapting job resource assignments (GPU type and count) is crucial and results in Sia outperforming all three state-of-the-art schedulers on all performance metrics considered: 30–93% lower average JCT, 28–95% lower p99 JCT, 38-65% lower makespan, 12–60% lower GPU hours used. Sia also outdoes the other schedulers on fairness metrics [59, 111], including 64% lower worst-case finish-time fairness and 99% lower unfair job fraction, even though Shockwave was designed to provide fairness. Additional results confirm Sia's (1) ability to improve cluster efficiency even when many jobs disallow changing of batchsize or GPU count, (2) ability to schedule and elastically scale Megatron[69]-style pipeline-model-parallel [66, 36] jobs (scale-out using data-parallelism), (3) scheduler-runtime scalability to sizable clusters (up to 2000 GPUs), (4) robustness to scheduler-parameter defaults, and (5) minor penalty for initially-crude bootstrapped throughput-models.

3.2 DL cluster scheduling and related work

A deep learning (DL) training job trains a deep neural network (DNN) model on a dataset in an iterative manner over multiple *epochs*. In each *epoch*, and for each *minibatch* in an epoch, an optimizer updates model parameters by minimizing a loss function over the minibatch of samples. Since the minibatch size is usually fixed for extended periods of training (if not the entirety of training), most DL jobs take a consistent and predictable [90] amount of time to complete a minibatch. These jobs are also generally *pre-emptible*, as one can checkpoint the state of the job (including the model and optimizer states) after any minibatch and resume the job from a checkpoint without losing much job progress. They are also amenable to

[3]For traditional data-parallel jobs, the minimum size is 1 GPU. For forms of model-parallel (e.g., pipeline parallel [66, 36]), which we term "hybrid parallel", the submitter-specified number of GPUs will be the minimum.

scaling as gradient computation can be parallelized across multiple GPUs on a single node and across multiple nodes[86, 16, 1, 12, 74, 45].

Although various parallelization strategies exist [66, 36, 45, 87, 69], most training jobs use *synchronous data parallelism* (DP)—given a set of GPUs, each GPU receives a *replica* of the model and computes gradients on a partition of the minibatch, whose size is termed *local batch size*. After the gradients from all the GPUs are reduced to a minibatch gradient, such as by a collective *all-reduce* [86, 74], an optimizer (e.g., SGD or Adam [50]) applies the gradient to generate the updated model parameters on each GPU. How well a given DL job scales depends on characteristics of the job (e.g., compute intensity and number of model parameters), the GPUs, and the inter-GPU network: for each minibatch, the gradient computation phase is divided among the GPUs, while the reduce phase synchronizes them. Prior work has shown that job scalability can be modeled effectively with relatively few measurements [79, 75].

Some DL jobs use forms of *model parallelism*, such as pipeline model parallelism (PMP[66, 36]) or Tensor Model Parallelism (TMP[69]), when the model being trained is too large to fit in a single GPU's memory. Powerful optimizers [93, 110, 66] exist for modeling performance of different configurations and partitioning a model across GPUs to maximize performance. Recently, some increase scale by mixing multiple parallelism types—e.g., Megatron-LM[69] mixes PMP and TMP at moderate scales and then employs synchronous data-parallelism to scale out to 100s of nodes.

Elastic and resource-adaptive DL jobs. Data-parallel DL jobs can be elastically re-sized over time, by checkpointing and then restarting on a different number of GPUs (with a different division of each minibatch's samples). Moreover, aspects of how the job does its work can be adapted to assigned resources, if the job is designed to do so [79, 107]. For example, the minibatch size can be adapted, such as by increasing its size when using more GPUs in order to increase the per-GPU compute for each minibatch and thereby increase scalability. Different minibatch sizes do have different statistical efficiency impacts, and the differences depend on job characteristics, but this effect can also be measured and modeled [61, 79].

Other DL jobs are usually submitted to a scheduler with a predetermined configuration, and changing it usually requires re-running the hybrid parallel optimizer. As discussed above, however, they can also be scaled using a data-parallel style by replicating the original configuration: for example, a PMP job that requires 4 GPUs for model and selected minibatch-size could use 8 GPUs and a doubled minibatch, one for each 4-GPU instance of the original configuration [69].

Resource heterogeneity. There are many *GPU types*, representing different product lines and generations produced by different vendors, and they naturally differ in GPU memory size and in compute and communication performance. It is common for a DL cluster to contain multiple GPU types. In part, this occurs because many clusters are deployed and grown over time, and the most cost-effective option can be selected each time new hardware is purchased and added. Looking ahead, the rapid development of new DL accelerators [7, 48, 55, 44], including some targeting specific DL models [106], will make having multiple "GPU types" a design feature rather than a deployment consequence. Unsurprisingly, a DL job may perform differently on different GPU types, and it can also scale differently for different GPU types (e.g., because the compute-to-network ratio changes). In addition, as illustrated

in Figure 2.1, different DL jobs can experience different speedups and different scalability.

DL Cluster Schedulers. In practice, DLT jobs are submitted as requests to a shared cluster, and the scheduler assigns resources to achieve cluster-wide goals. Many schedulers only accommodate requests that specify a fixed number of GPUs, ignoring opportunities presented by elasticity, resource-adaptivity, and heterogeneity. Others do address some of these opportunities. Sia seeks to address them all.

3.2.1 Related work in DL cluster scheduling

To our knowledge, no prior scheduler optimizes assignments for resource-adaptive jobs on a heterogeneous DL cluster. This section groups prior schedulers by unaddressed aspects.

Scheduling for heterogenenous DL clusters (no resource-adaptive jobs). Among DL schedulers that are designed to handle heterogeneity within a cluster [11, 56, 68], none adaptively tune the number of GPUs assigned nor account for other potential adaptations made by DL jobs. Instead, the user specifies a number of GPUs for each job submitted.

Gavel[68] is the best-performing state-of-the-art heterogeneous DL cluster scheduler, using a fast linear-program formulation that scales to large cluster sizes. However, Gavel does not support job adaptivity, and only optimizes the assigned GPU type given the minibatch size and GPU count specified by job submitter. This approach may lead to under-utilization of newer, more powerful GPUs because of too-small batch sizes. Also, when the cluster is congested, Gavel time-shares resources between jobs, wasting GPU time on checkpoint-restore operations.

Most importantly, extending Gavel to handle job adaptivity is non-trivial: Gavel expresses scheduling options using a throughput matrix populated with (job_id, GPU_type) pairs. If one simply expands the throughput matrix to contain entries for each adaptivity choice (job_id, GPU_type, num_GPUs, minibatch_size), it leads to two problems – (1) populating a non-trivial portion of this matrix will require extensive per-job profiling, and (2) the resulting optimization program is too large to be solved quickly.

Scheduling for elastic and resource-adaptive jobs (no heterogeneity). Among DL schedulers that are designed to tune for elastic and resource-adaptive jobs [75, 101, 88, 79, 38], none consider GPU heterogeneity—they assume that all GPUs in the cluster are identical.

Pollux [79] is a state-of-the-art DL cluster scheduler for elastic resource-adaptive jobs for homogeneous clusters. Pollux uses per-job goodput models to assign both a number of GPUs and a batchsize setting to each current job, and it re-considers all assignments each scheduling cycle based on updated job behavior and job queue information. By doing so, it exploits elasticity to avoid unused or over-committed GPU resources. Each job's goodput model consists of two component models: one for statistical efficiency (a rate of training progress per sample, based on Gradient Noise Scale [61]) as a function of batchsize, and one for throughput (samples processed per second) as a function of both GPU count and batch-size. How each job scales with GPU count is learned by scaling it up, measuring each count tried, and interpolating for others. Pollux uses the per-job models with a genetic algorithm to search the space of resource allocations (and corresponding batchsizes) for all current jobs to maximize aggregate cluster-wide goodput weighted by fairness. Unfortunately, Pollux's

no-pre-profiling throughput modeling approach blocks consideration of GPU heterogeneity. Worse, Pollux's formulation of the scheduling problem as a genetic optimization problem results in very poor cluster-size scaling even for homogeneous clusters with 100s of GPUs, which would only worsen with GPU heterogeneity.

This is because Pollux considers a very large number of adaptivity choices: for each (job, GPU_count) pair, it considers every possibly way to place this job across all nodes. As a result, the number of possible solutions is exponential in the number of nodes and number of GPUs per node. For clusters with 1000+ GPUs, it is too slow to respond to changes in cluster as it takes tens of minutes for the genetic algorithm to terminate (see Figure 3.10).

Scheduling for rigid jobs on homogeneous DL clusters. Most existing DL schedulers require the submitter to specify GPU count (and job configuration) for each job [27, 59, 111]. These schedulers do not adjust the number of GPUs assigned based on current load or scalability/efficiency of current jobs. They also do not consider GPU type differences, instead assuming that all GPUs in the cluster are identical. As such, these schedulers use DL cluster resources less efficiently than schedulers from the two prior categories [79, 68]. As a recent example, Shockwave [111] improves performance and fairness relative to prior schedulers in this category, and we include it in our evaluations. Some batch-schedulers like Kubeflow and Volcano can adjust GPU count to improve GPU utilization, but do not co-adapt batch-size, GPU count and type simultaneously.

Parallelism optimizers that are not cluster schedulers. There are various optimizers [66, 93, 110, 6, 94, 45, 97] for selecting an individual job's configuration before acquiring cloud resources for it or submitting it to a cluster scheduler. Such optimizers are especially important and popular for hybrid parallelism approaches. However, they cannot be considered cluster schedulers as they consider an individual job in isolation, without considering cluster load or trade-offs in assigning a particular resource to one job rather than another. To our knowledge, no existing scheduler co-optimizes non-data-parallel job configurations and cluster resource assignments, even for homogeneous clusters.

3.3 Sia Design and Implementation

Sia is a pre-emptive, round-based scheduler that optimizes allocations for a set of jobs to maximize cluster-wide goodput. In each round, jobs receive bundles of resources (CPU, GPU and network, like VMs in cloud) and Sia uses checkpoint-restore preemption to optimize job adaptivity.

3.3.1 Sia components and job life cycle

Figure 3.2 illustrates the life-cycle for a job J under Sia. A user submits a job J to Sia (①) and declares both the maximum batchsize (`max_bsz`) and GPU count (`max_ngpus`) for execution. Sia then profiles throughput of J on a few batchsizes using one GPU of each type(②). Goodput Estimator bootstraps a throughput model for J on each GPU type using the profiles. Goodput estimates for J on various resource configurations are provided to the policy optimizer(⑧) for informed scheduling. Job J stays in the queue (④) until

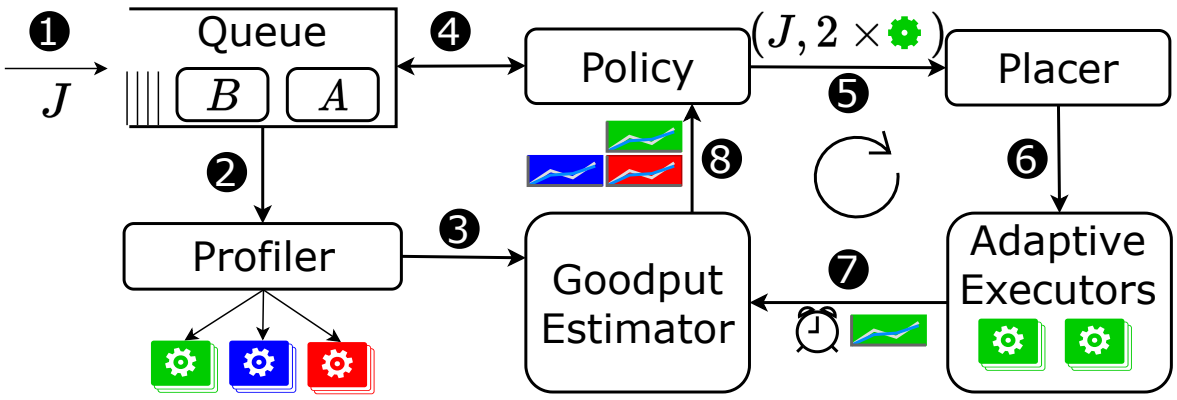


Figure 3.2: Lifecyle of a job under Sia. After a job is submitted, it is profiled once on each GPU type for a few batchsizes. Upon receiving an allocation, the job begins a cycle of continuous optimization (steps 5-8) for the remainder of its life in the cluster. *Policy* continuously optimizes allocations for the job, while *Goodput Estimator* provides up-to-date performance and gradient statistics to *Policy* to aid in decision making.

Sia allocates some GPUs to it and then enters a cycle where its adaptivity is continuously optimized by Sia as follows.

Continuously optimized job adaptivity. Sia Policy uses goodput estimates from each job's Goodput Estimator and finds an optimal partitioning of cluster resources among the jobs in the cluster, giving job J, say, 2 GPUs of type GREEN (⑤ in Figure 3.2). (The Goodput Estimator combines Sia's throughput model with a statistical efficiency model borrowed from Pollux [79].) Placer then determines the 2 GPUs to assign to job J (⑥) given the current assignment of GPUs to jobs and attempts to reduce unnecessary job migrations due to resource de-fragmentation. Sia runs jobs on Adaptive Executors that support (1) transparent checkpoint-restore for low-overhead job pre-emption and resource scaling, (2) batchsize adaptivity to maximize statistical efficiency, and (3) frequent reporting of gradient and throughput statistics for current allocation (default = 30 seconds). After J starts running on Adaptive Executors, Goodput Estimator uses J's gradient and throughput statistics (reported by Adaptive Executors) to update the goodput model for J on GPU type GREEN(⑦). In the next scheduling round, Sia Policy queries the updated goodput estimates for J on all GPU types (⑧) and completes the loop in the Sia architecture (⑤→⑥→⑦→⑧...), allowing us to continuously optimize J's goodput until its termination/completion.

Heterogeneous Execution. Sia transparently handles GPU heterogeneity in number and capabilities – GPU memory capacity, interconnect speeds, throughput are modeled in the goodput estimator, and Adaptive Executors optimize for goodput given a fixed set of resources. Gradient accumulation is used if statistical efficiency dictates higher batchsize than supported by GPU memory limits, with goodput optimized over a larger range of per-GPU batchsizes for GPUs with larger memories, fully exploiting whichever GPU type for optimal job progress.

Job Scaling policy. Sia uses a simple scale-up policy – start each job with exactly 1 GPU, and scale the job up by a maximum of 2× in each scheduling round. If a job requires a minimum of `min_ngpus` to start execution, Sia will respect this minimum and ignores all

allocations *smaller* than `min_ngpus` for this job. Jobs may also be scaled down to a minimum of `min_ngpus` to accommodate more jobs in the cluster (determined by the scheduling objective).

Decoupled allocation and placement. Given a set of heterogeneous resources to be partitioned among a set of jobs, Sia decomposes the problem into two stages – (a) an *Allocation* stage (⑤) that determines the number and type of resources to assign to each job, and (b) a *Placement* stage (⑥) that determines the exact physical resources (and the network topology) to satisfy allocations for all jobs. This decoupling allows us to restrict the space of placements for an allocation (there exist many placements for a given allocation[89]). Sia uses three rules to obtain placement in Placer: (a) partial node (fewer GPUs than max GPUs per node requested) allocations must not be split across two nodes, (b) whole node allocations must take whole nodes, and (c) if there exists no placement satisfying (a) and (b) (resource fragmentation), evict some jobs and try again. Evictions resulting from fragmentation are quite rare and often result in fewer than 3 evictions at once. As we will see in Section 3.3.3, restricting allocations to a particular set allows us to guarantee a placement for all valid allocations output by Sia.

3.3.2 Bootstrapping of throughput models

A naive approach to constructing each job's throughput model (as a function of GPU count and batchsize) for every GPU type would require profiling a variety of multi-GPU allocations for each GPU type to collect compute and communication times. This profiling overhead grows linearly in both the number of GPU types and the number of nodes of each GPU type. Sia takes a different approach, starting with minimal profiling information and refining based on observed allocations.

For each job, Sia learns one throughput model for each GPU type and one statistical efficiency model for the job. Consider a job J submitted to Sia running on a cluster with two GPU types A and B. Let's assume that J needs `min_GPU_count`$=> 1$ GPUs per data-parallel worker. Sia first profiles J on one GPU of each type (corresponding to ②in Figure 3.2). Starting from a minimum batchsize, Sia profiles increasingly larger batch sizes till it hits GPU memory limits (typically 10 profiled batchsizes per GPU type); altogether, the average per-job profiling cost is < 20 GPU seconds per GPU type. This gives us two crucial pieces of information: (1) compute times for various combinations of GPU type and batchsizes, and (2) comparison of compute times across GPU types. Importantly, compute time is independent of GPU count increases (since we scale via data-parallelism with all-reduce), this leaves only the communication time to be predicted.

Sia initializes J's throughput models for each GPU type using their 1-GPU profiles. These throughput models are used by Sia to place J on 1-GPU of some type, say A. Once J starts running on a single A GPU, online profiling is used to (a) learn a statistical efficiency model for J as a function of batch size, and (b) refine throughput model for J on 1-GPU of A type. These throughput models, however, cannot estimate communication time, so Sia makes a **one-time** simplifying assumption to estimate J's throughput on 2-GPUs of A: *throughput of two data-parallel replicas is twice the throughput of a single replica* (i.e. perfect scaling with zero communication time). Say Sia then assigns 2-GPUs of type A to J. Using online profiling, Sia refines J's throughput model for A GPUs using the measured communication

times on a multi-GPU allocation. Sia can now use the refined throughput model to estimate J's throughput on multi-GPU allocations on A GPUs as it accurately models both compute and communication time. However, since J has not yet run on a multi-GPU allocation on B GPUs, the throughput model for B GPUs does not model communication time on B GPUs as it was learned from initial profiling and cannot be used to estimate J's throughput on, say, 4-GPUs of B type. To overcome this problem, Sia combines J's learned throughput model for A GPUs with the initially profiled single-GPU throughputs for both A and B to obtain a crude *bootstrapped* throughput model for B GPUs. In our example, J's throughput on N GPUs of B type is estimated with a bootstrapped throughput model, $\texttt{est-xput}_B$, given by:

$$\texttt{est-xput}_B(N) = \frac{\texttt{xput}_B(1)}{\texttt{xput}_A(1)} * \texttt{xput}_A(N) \tag{3.1}$$

where $\frac{\texttt{xput}_B(1)}{\texttt{xput}_A(1)}$ is the ratio of 1-GPU throughputs and $\texttt{xput}_A(N)$ is the throughput for N GPUs of A type. This simple estimator assumes that if we do not know the communication time for B, the scaling of compute:communication ratio for B is the same as A (which is known). In Section 3.5.8, we show that bootstrapped throughput models are accurate enough to guide Sia towards taking useful explorative steps.

We use $\texttt{est-xput}_B$ to estimate goodput for multi-GPU allocations on B GPUs and if J runs with a multi-GPU allocation on B GPUs, we can safely discard the bootstrapped throughput model (from Equation (3.1)). This is because using online profiling, Sia can refine $\texttt{xput}_B$ to accurately predict communication time on B GPUs (which is now known), eliminating the need for the crude bootstrapped model $\texttt{est-xput}_B$.

3.3.3 Configurations

A configuration represents a bundle of resources (CPU, GPU, Network, etc) and is similar to virtual machine sizes in the cloud. Configurations can be represented as a 3-tuple – (n, r, t) where n is the number of nodes containing a total of r resources of type t. For example, $(2, 16, \texttt{T4})$ represents a configuration with 2 nodes containing 16 `T4` GPUs in total.

Sia's Policy supports efficient job adaptivity by optimizing for allocations over a small *valid* set of configurations designed to simplify placement logic in Placer. This set can be decomposed into two sets: a *single-node* allocation set which contains allocations that do not cross a node boundary (i.e. $n = 1$), and a *multi-node* set that contains allocations that span node boundaries (i.e. $n > 1$). We provide a construction of these sets below.

Consider a cluster with N physical nodes, containing R GPUs of type X per node, the configuration set C is given by a union of the *single-node* and *multi-node* sets –

$$\begin{aligned} C = &\{(1, 2^0, X), (1, 2^1, X), \dots (1, R, X)\} \cup && \leftarrow \texttt{single-node} \\ &\{(2, 2R, X), \dots, (N, N \cdot R, X), n \in \mathbb{N}\} && \leftarrow \texttt{multi-node} \end{aligned}$$

The *single-node* set constrains allocations to be powers of 2 within a node, and at most R, the number of GPUs within a node. If R is not a power of 2, one can decompose R as a sum of powers of 2, and model each physical node with R GPUs as multiple virtual nodes with different GPU counts. The *multi-node* set constrains all allocations to use all available

GPUs in a node (i.e. GPU count is a multiple of R). Using these allocation and resource sets, we can rely on existing literature (*Submesh Shape Covering theorem* [110]) to guarantee a placement for all valid allocations where no two distributed jobs share any nodes. This is especially desirable because it eliminates resource contention on the NICs which can cause significant slowdown to all contending jobs [79, 43].

In a homogeneous cluster, Sia matches Pollux's performance (Table 3.4), despite optimizing over a smaller configuration set. Pollux optimizes over the full space of (GPU count x placement) choices for each job ($O(N^R)$), while Sia restricts the configuration set to a size of $(N + \log_2 R)$ for a cluster with N nodes and R GPUs each. This suggests that our restrictions do not significantly impact job runtimes. This reduction in problem complexity allows Sia's optimization to scale to clusters with thousands of GPUs (see Section 3.5.7) with practical runtimes.

3.3.4 Scheduler objective

This section describes the Sia scheduler objective. We use a running example where a heterogeneous cluster has 2 GPU types - (a) one node with 2 GPUs of type A and (b) one node with 4 GPUs of type B. Let $J = \{J_1, J_2\}$ be the set of jobs in the scheduler queue, both of which require a minimum of 1 GPU to run.

Valid configurations. Using rules described in Section 3.3.3, we construct the set of valid Sia configurations C. For the example cluster, $C = \{(1, 1, A), (1, 2, A), (1, 1, B), (1, 2, B), (1, 4, B)\}$. Recall that if Sia assigns a configuration $c = (n, m, X) \in C$ to a job, the job runs with m GPUs of type X split across n nodes. A job receives either no resources in a scheduling round, or a set of resources identified by a valid configuration.

Goodput estimation. Sia uses one throughput model for each (job, GPU_type) combination to model job and hardware heterogeneity effectively. Sia optimizes for goodput cluster-wide, so we use the per-job statistical efficiency models to derive goodput estimators, one for each (job, GPU_type) combination. Let (f_A, f_B) and (g_A, g_B) be the goodput estimators for jobs J_1 and J_2 and GPU types A, B, respectively. We define a goodput matrix G of size $|J| \times |C|$, where G_{ij} is the estimated goodput for job $J_i \in J$ using resources defined by configuration $c_j \in C$. For a given job J_i, all values in that row are comparable: $G_{ij} > G_{ik}$ means configuration c_j is better than c_k for the job J_i. However, for a given configuration c_j, $G_{mj} > G_{nj}$ does not derive that J_m deserves to run in configuration c_j over job J_n. We apply a simple row-normalization technique to make values in G comparable across jobs for each configuration.

Normalized goodput matrix. For each job J_i with minimum required GPU count N_i^{min},

$$G_{ij} \leftarrow N_i^{min} \cdot \frac{G_{ij}}{\min_j G_{ij}}$$

where $\min_j G_{ij}$ is the minimum of goodput values for the job J_i across all configurations in C. The result matrix G is called a normalized goodput matrix.

Using the row-minimum values to normalize each row in G provides two benefits. First, we can interpret G as a utility-matrix for jobs J, with G_{ij} capturing the utility of configuration C_j to job J_i. Second, we can compare utilities for a given configuration across job types. Choosing the configuration with the highest value along a job's row in G makes the most

progress for that job, and a configuration is best used by the job with the highest value along the configuration's column in G.

Each new job adds a row to G, and the completion of a job delete its respective row, which keeps G up to date with goodputs only for active jobs. If a job's statistical efficiency changes, or its throughput model gets more refined, G is updated to track the most recent values. For our running example, Table 3.1 shows the normalized goodput matrix G.

	$(1,1,A)$	$(1,2,A)$	$(1,1,B)$	$(1,2,B)$	$(1,4,B)$
J_1	1	1	2	3	**4**
J_2	2	**4**	1	2	3

Table 3.1: Normalized goodput matrix G. Boxed entries show the allocation that maximizes sum of goodput for jobs J_1, J_2.

Scheduler objective. G represents the utility of the set of configurations C to the set of jobs in J, and Sia selects the (job, configuration) pairs that maximize the sum of normalized goodputs for jobs in the chosen configurations. Each configuration maps to a unique allocation and by constraining the number of allocated resources, a valid schedule can be determined.

We define a binary matrix A with the same shape as G where $A_{ij} = 1$ if configuration c_j is chosen for job J_i and 0 otherwise. We formulate the problem of choosing the best pairs (as outlined above) as the following optimization problem over A:

$$\max_{A} \sum_{i=1}^{|J|} \Big(\sum_{j=1}^{|C|} A_{ij} \cdot G_{ij} \;+\; \lambda(1 - ||A_i||_1) \Big) \tag{3.2}$$

where $||v||_1$ denotes the ℓ_1 norm of a vector v. This objective is composed of two terms: a sum of normalized goodputs of jobs in all chosen configurations, and a scheduler penalty for not choosing any configuration for each job—no penalty if some configuration is chosen for job J_i ($A_{ij} = 1$ for some j, so $||A_i||_1 = 1$), and a constant penalty $-\lambda$ otherwise. The penalty λ can also be thought of as an incentive to reduce scheduler queue occupancy: if λ is large, then Sia will allocate at-least one GPU to each job in the cluster, if available.

We formulate the Sia scheduler objective as a (binary) Integer Linear Program task with the binary matrix A as an optimization variable and the following added constraints:

wide Each job chooses at-most one configuration: $||A_i||_1 \leq 1$

wiide Allocated number of GPUs does not exceed available GPUs for each GPU type

Solving the optimization problem gives us a binary solution matrix A that contains allocations for the next scheduling round: a job J_i receives no resources if $||A_i||_1 = 0$, otherwise (there must exist an $A_{ij} = 1$) it runs under configuration c_j for the next scheduling round. For the normalized goodput matrix G shown in Table 3.1, optimizing Equation (3.2) gives us the allocations : J_1 gets configuration $(1, 4, B)$ and J_2 gets $(1, 2, A)$. The corresponding entries in G are each highlighted with a box in Table 3.1.

Restart Factor. To prevent frequent job restarts which can harm performance, a re-allocation factor, r_i, is used to adjust the utilities in G for configurations that differ from

the currently allocated configuration for each job J_i. This multiplicative factor models the expected goodput for such configurations by projecting the historical rate of restarts into the future, and is necessary because the restarting cost for deep learning jobs can be high (e.g., 25-250 seconds for the models listed in Table 3.2). Consider a job J_i with age T_i, wasting S_i GPU seconds per *restart* operation and having restarted N_i times previously. The re-allocation factor r_i for the job J_i is computed as follows:

$$r_i = \frac{T_i - N_i \cdot S_i}{T_i + S_i} \tag{3.3}$$

If job J_i is currently running under a configuration c_k, we discount goodput values for all other configurations $c_j \neq c_k$ that require a restart by the restart factor: $G_{ij} \leftarrow r_i \cdot G_{ij}$. Without a restart factor, each tiny changes in G would result in altering some jobs' resources and additional checkpoint-restore overheads. By applying a restart factor to only those utility values in G that require restarting the job, Sia only restarts jobs if not doing so results in a big reduction in optimal value for its scheduling objective.

Balancing goodput with fairness. We provide a simple knob to tune *fairness* of allocations in Sia – a parameter p that can be used to manipulate the *scale-free* matrix G by raising the elements to the power of p. If $p < 0$, we flip the sign of the objective (i.e., minimize the original objective instead of maximizing) to preserve its semantics. We investigate the effect of p on Sia's scheduler metrics in Section 3.5.8, showing that it provides robust fairness with minimal negative impact on efficiency metrics across a range of settings between -1.0 and 1.0. We use a default of -0.5. Sia's full scheduler objective, for $p > 0$, is as follows:

$$\max_{A} \sum_{i=1}^{|J|} \Big(\sum_{j=1}^{|C|} A_{ij} \cdot \big(r_i \cdot G_{ij}\big)^p \; + \; \lambda(1 - ||A_i||_1) \Big) \tag{3.4}$$

Support for limited adaptivity. Sia supports executing jobs with some adaptations disabled (batch size, GPU count and/or type). Large batch sizes result in high throughput and GPU utilization, but may result in a generalization gap for the trained model[61, 49]: a phenomena where the final model performs poorly on unseen samples. Sia supports different types of jobs with varying degrees of adaptivity to accommodate diverse reasons for limited adaptivity: *strong-scaling* jobs run with a fixed batch size, but allow the GPU count and type to be optimized, while *rigid* jobs run with a fixed batch size and GPU count, only leaving GPU type to be optimized. Both strong-scaling and rigid jobs preserve model quality and training semantics by keeping batch size fixed, but allow Sia to optimize job execution in a limited manner. Given a fixed batch size, goodput is directly proportional to throughput; so for strong-scaling jobs, Sia directly uses throughput in place of goodput in Equation (3.4). For rigid jobs, we add the following objective to Equation (3.4):

$$\max_{B} \sum_{i=1}^{|J_R|} \Big(\sum_{g=1}^{N_g} B_{ig} \cdot \big(r_i \cdot T_{ig}\big)^p \; + \; \lambda(1 - ||B_{ig}||_1) \Big) \tag{3.5}$$

where J_R is the set of rigid jobs, N_g is the number of GPU types, T_{ig} is the goodput of job J_i on GPU type g and r_i is the job's restart count. We then update constraints for the ILP to constrain total GPUs allocated for each GPU type across all active jobs.

In a similar manner, with few changes to Equation (3.4) and optimization program constraints, Sia's flexible scheduling formulation can support scheduling custom resource requests and jobs with user-defined parallelism tuned to a specific GPU count, type and/or batch size.

Preemption and reservation. Sia assumes all jobs are preemptive, but can also support a small number of non-preemptive jobs in the cluster (as long as their aggregate demand can be satisfied): for each non-preemptive job, we add a constraint to Equation (3.4) to force the requested resources to be allocated. This constraint ensures that the non-preemptive jobs get allocated first, guaranteeing non-preemption in each scheduling round. Reservations are implemented in a similar manner.

Support for other parallelization techniques. In general, Sia only requires that a job provide a goodput estimator that can be evaluated on valid configurations.

This design allows Sia to support jobs with more advanced parallelization strategies [66, 110, 97]. We extend Sia's throughput models to support jobs that use a combination of pipeline [36, 66] and data parallelism, allowing Sia to schedule jobs with multi-billion-parameter models.

These jobs employ a mix of data and pipeline parallelism [69] where a pipeline parallel strategy partitions a large model onto many GPUs, and data parallelism is used to scale up training (see Section 3.5.3). Each model partition is mapped to one or more GPUs, say P GPUs across all partitions. A job with N data-parallel replicas uses exactly $N \times P$ GPUs. Given a mini-batch size of M and micro-batch size of m, each replica computes gradients locally using $\frac{M}{mN}$ micro-batches of size m each across P GPUs. Then, N replicas of these pipelines synchronize using a gradient all-reduce, thus finishing one training iteration. The distinct compute and communication phases [69] allow us to leverage Sia's throughput models for goodput estimation at various batch sizes. Since these jobs scale as units of P GPUs each, we add additional terms to our scheduling objective with the appropriate constraints (similar to Equation (3.5)). We discuss adaptation for one such hybrid-parallel model in more detail in Section 3.5.3.

Existing hybrid-parallel optimizers are time-consuming[110, 63], so we leave the problem of efficient elastic scaling without fixing non-data-parallel degrees as future work.

Scheduling other workload types. Sia exploits characteristics unique to deep learning training, but we believe it could also handle other batch-processing workloads by using a goodput estimator customized to each workload type. For example, one can use Sia to schedule batch deep learning inference jobs that run inference on a large dataset. Here, throughput can be used as a proxy for goodput, yielding a simple goodput estimator. For latency sensitive inference jobs, one could use Sia to pick the right set of resources: goodput=1 if a configuration can support inference within the promised latency constraints and 0 otherwise.

3.3.5 Implementation

We implement Sia using the open-source AdaptDL framework, replacing its scheduler and data-loader implementations with our own, as the PyTorch-based framework provides native support for dynamically adjusting batch-size and number of GPUs for DL training jobs on Kubernetes-managed GPU clusters. For a data-parallel DL training job, we use AdaptDL data-loaders to vary batch-size during training, and use all-reduce to synchronize gradients

across workers. Sia Adaptive Executor continually profiles minibatch runtimes and gradient statistics, periodically (default 30s), optimizes goodput model parameters using these profiles and communicates the new goodput model parameters to Sia Policy. It also selects the batch-size that maximizes goodput given allocated resources and scales the learning rate in accordance with the selected batch size using a configurable learning rate scaling rule. For models listed in Table 3.2, we use the square-root learning-rate scaling rule[52] for models using the AdamW [58] optimizer and AdaScale[47] scaling rule for models with SGD optimizers.

Sia Policy runs as a Kubernetes service, and at the start of each scheduling round, uses the latest goodput model parameters for each job to optimize resource allocation using (Equation (3.4)). We formulate Equation (3.4) as a Mixed-Integer Linear Program using the `GLPK_MI` [60] solver from the CVXPY package [21] and use the output solution to determine job allocations.

Preemption with checkpoint-restore. If a DL training job's allocation changes, Sia preempts the job only after the current minibatch has finished processing so there is no communication in flight. First, Sia checkpoints the latest model weights, data-loader (e.g., sampler and iterator states) and optimizer states (e.g. gradient statistics for Adam[50]) to shared persistent storage and releases all GPUs allocated to the job. Then, on the new resources, Sia launches one Adaptive Executor per GPU, restores training state from the checkpoint on disk, and resumes model training.

Sia also uses the checkpoint-restore mechanism to recover from worker failures. After every epoch, Sia checkpoints model weights and optimizer states to disk, so if some workers fail in the next epoch, model training can be resumed from the last saved checkpoint on different resources.

3.4 Experimental Setup

We compare Sia with state-of-the-art schedulers in both homogeneous and heterogeneous clusters using workloads derived from real-world environments. This section describes the workloads, configurations and the schedulers used.

Table 3.2: Models used in our evaluations.

Size	Task	Model	Dataset	Target Metric	Batch Sizes	Optimizer
S	Image Classification	ResNet18 [31]	CIFAR-10 [53]	94% Top-1 acc	[128 - 4096]	SGD
M	Question-Answering	BERT [20]	SQuAD [81]	0.88 F1 score	[12 - 384]	AdamW [58]
	Speech Recognition	DeepSpeech2 [3]	CMU-ARCTIC [51]	25% word err	[20 - 640]	SGD
L	Object Detection	YOLOv3 [83]	PASCAL-VOC [22]	85% mAP	[8 - 512]	SGD
XL	Image Classification	ResNet50 [31]	ImageNet-1k [19]	75% Top-1 acc	[200, 12800]	SGD
XXL	LLM Finetuning	2.8B GPT [80]	SQuAD	0.88 F1 score	[48, 384]	AdamW

3.4.1 Workloads and Traces

We use traces derived from three production DL clusters, using a common approach from recent work [111, 79, 68]. We categorize each job in a trace based on its total GPU time: Small (0-1 hrs), Medium (1-10 hrs), Large (10-100 hrs) and Extra-large (XL, ¿100 hrs). We

map each category into one or more representative jobs as listed in Table 3.2. XXL models are only used for hybrid-parallel experiments in Section 3.5.3.

Philly is from 100k jobs executed over two months in a multi-tenant cluster with multiple GPU types at Microsoft [43].

Helios is from the Saturn cluster in the Helios cluster traces [35]. The original traces contain 3.3M jobs recorded over a six-month period in a heterogeneous cluster with over 6k GPUs. Compared to **Philly**, **Helios** jobs request more GPUs and run for longer, resulting in a higher cluster load.

We derive ten traces for each workload by randomly sampling the 8 busiest hours in the respective real-world trace using an average job arrival rate of 20 jobs/hr, resulting in a total of 160 jobs submitted over the 8-hour window.

newTrace is a more recent trace from a production system for deep learning jobs that spans multiple clusters with thousands of GPUs. Similar to the Microsoft Philly traces [43], this production system allocates Virtual Machines (VMs) to DL training jobs where each VM instance is provisioned a pre-configured amount of CPU, GPU and memory resources. Similar to other production environments, we observe a wide range of resource requests exhibiting diurnal patterns with bursts of resource requests coming by virtue of job submission scripts (e.g., hyper-parameter tuning). We sample 10 traces over a 48 hour period at an average arrival rate of 20 jobs/hr (total 960 jobs submitted in each trace).

The longer 48-hour **newTrace** traces are used to evaluate a more *realistic* setting where congestion slowly builds up in a cluster from long-running jobs over a long duration. **newTrace** sees a significant variance in job arrival rates from 5 to 100 jobs/hr over the 48-hour job submission window and gives us valuable insights into how schedulers can deal with congestion and variance in cluster loads.

3.4.2 Hardware measurements and simulator

Most of our experiments use the discrete-time simulator open-sourced [40] by authors of Pollux whose fidelity is verified by prior work [79] and our own measurements. We added a Gavel implementation and the open-source Shockwave [111] to the simulator, as well as extended the original version of Pollux to support heterogeneous clusters. The simulator allows us to experiment on a range of cluster sizes and hardware configurations.

We use four different types of GPUs in our experiments: a cluster of 16 `t4`instances [79] and three on-premise node types (3x `rtx`, 2x `a100`, and 1x `quad`):

`t4` – [Cloud] `g4dn.12xlarge` AWS EC2 instance with 4 NVIDIA T4 (16GB VRAM) GPUs.

`rtx` – [On-prem] *commodity* node with 8 NVIDIA RTX 2080Ti (11GB VRAM) GPUs and 50Gb/s Ethernet.

`a100` – [On-prem] *high-performance* NVIDIA DGX-A100 node with 8 NVIDIA A100 (40GB VRAM) GPUs and 1.6Tb/s Infiniband.

`quad` – [On-prem] *workstation* node with 4 NVIDIA Quadro RTX6000 (24GB VRAM) GPUs and 200 Gb/s Infiniband.

We were able to get a limited amount of dedicated time with the on-prem nodes, which allowed for direct experiments on a 44-GPU, 3-GPU-type cluster. The results (Section 3.5.1) confirm Sia's efficacy and the simulator's fidelity.

The original simulator from [79] simulates checkpoint-restore with the same constant delay for all jobs, which we replaced with model-specific checkpoint-restore delays.

3.4.3 Evaluated settings

We compare schedulers in the following three settings:

- *Physical*: Physical cluster with 3 `rtx`, 2 `a100`, and 1 `quad` nodes for a total of 44 GPUs. In Sec. 3.5.1, we compare Sia with Pollux and Gavel.

- *Homogeneous*: Simulated cluster with 16 `t4` nodes(64 GPUs). In Sec. 3.5.2, we compare Sia to Pollux, a state-of-the-art job-autoscaling scheduler for homogeneous clusters, and inelastic schedulers Shockwave [111], Themis [59], and Gavel [68].

- *Heterogeneous*: Simulated cluster with 6 `t4`, 3 `rtx`, and 2 `a100` nodes (64 total GPUs). In Sec. 3.5.2, we compare Sia with Pollux and Gavel, a state-of-the-art heterogeneity-aware scheduler.

Tuning job hyper-parameters. Gavel lacks support to auto-tune job parameters, so we manually tune the batch size and requested number of GPUs for each job in our sampled traces to ensure optimal performance. We follow the approach used in [79] and optimize each job's batch size and GPU count: we search over (batch size, GPU count) combinations (GPU count $\leq$ 64 GPUs for *Homogeneous*, and $\leq$ 16 GPUs for *Physical* and *Heterogeneous* settings) and randomly choose a combination (`bsz, GPU_count`) such that the simulated runtime using (`bsz, GPU_count`) is 50-80% of ideal speedup over the runtime of a 1-GPU baseline with the optimal batch size. We refer to these optimized job configurations as *TunedJobs* (TJ) in our evaluations, even though real-world jobs may be submitted with worse performing job parameters.

Fixing mixed-GPU allocations from Pollux: To make Pollux work on our heterogeneous clusters, we present 8-GPU nodes as 2 virtual 4-GPU nodes to eliminate heterogeneity in node capacities. However, Pollux may still schedule a job on more than one GPU type (not allowed in our setup). So, we apply a simple heuristic: the GPU type with the most GPUs is selected, and in case of a tie, the more powerful GPU type is chosen (`a100` ¿ `quad` ¿ `rtx` ¿ `t4`). Although not perfect, this heuristic enables fair comparisons to Pollux in heterogeneous settings (Section 3.5). Our paper's focus is not on designing the perfect heuristic.

Default parameters. Unless explicitly stated otherwise, all experiments use the following parameters: $p = -0.5, \lambda = 1.1$ for Sia, $p = -1$ for Pollux (same as [79]), (10, 1e-1) for Shockwave (same as [111]. We choose a scheduler round duration of 60s for Sia and Pollux, and choose 360s for Gavel, Themis and Shockwave. We choose the *max-sum-throughput* scheduling policy [68] for Gavel as it results in the lowest average JCT on **Philly** traces among the policies listed in [68]. We investigate sensitivity of Sia to its parameters in Section 3.5.8.

3.5 Evaluation

This section evaluates Sia, showing that it outperforms state-of-the-art cluster schedulers for both resource-adaptive and rigid jobs running on both *homogeneous* and *heterogeneous* resources. Results also show that Sia provides better finish-time fairness, scales to large cluster sizes, and is not overly sensitive to our default parameter settings.

3.5.1 Physical cluster experiments

We compare Sia with Pollux and Gavel in the 44-GPU *physical* cluster setting (Sec. 3.4.1) that consists of 3 `rtx` + 1 `quad` + 2 `a100` nodes. We sample a smaller, single 3-hour trace with 30 jobs with a mix of all the models listed in Table 3.2 and run all schedulers on the physical cluster. Owing to resource availability constraints,[4] we run Sia, Gavel and Pollux four times to account for any randomness in their schedules.

Figure 3.3 shows the results of our physical cluster experiment side-by-side with those predicted by the simulator. On the physical cluster, Sia provided lower average JCT than Gavel or Pollux by 50% and 35–50%, respectively.

Figure 3.4 shows resource allocations for three jobs over 45 minutes, illustrating how Sia dynamically adjusts GPU count and type. Rising congestion triggers Sia to scale down and then move the `ImageNet` job (top) to `rtx` GPUs, leaving the fastest (`a100`) GPUs for incoming `CIFAR-10` jobs. Over time, Sia scales up and refines throughput models for each job (e.g. `DeepSpeech2` job in Figure 3.4) while adapting to GPU type and count changes. When congestion decreases sufficiently, Sia shifts `ImageNet` back to `a100` and scales out `DeepSpeech2` on `rtx` GPUs for better throughput.

Simulator fidelity. The simulator was found to have less than 5% error in average JCT and Makespan for both Sia and Gavel, validating its accuracy yet again. However, Pollux performed significantly worse on the physical cluster than predicted by the simulator, due in part to the modifications we made to the simulator giving Pollux an advantage when scheduling a single job over heterogeneous resources (Section 3.4.3). The schedules produced by Pollux can have large variations due to randomness in its optimization and its potentially *misguided* job adaptivity (due to noisy throughput estimators), resulting in bad worst-case scenarios. Additionally, the heterogeneity of the underlying hardware mapped to the virtual nodes that Pollux assumes are homogeneous can also contribute to the variation.

3.5.2 Simulator experiments

Table 3.3 shows key performance metrics for Sia, Pollux, and Gavel running on the heterogeneous cluster with traces described in Section 3.4.1. Across all traces and evaluated metrics, Sia outperforms heterogeneity-aware schedulers like Gavel and job auto-scaling schedulers like Pollux. Sia reduces average JCT by 30–93% and 99th-percentile JCT (p99 JCT) by 28–95%, compared to Pollux and Gavel. In doing so, Sia is also more resource efficient– it allocates 12-60% fewer GPU hours per job compared to Pollux and Gavel.

[4]Unlike profiling runs, our scheduler experiments require extended isolated control over the entire collection of machines, blocking out all users for which the machines were acquired.

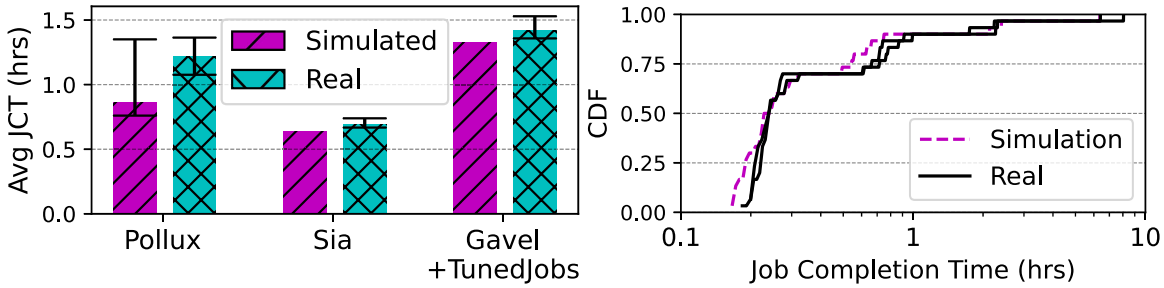


Figure 3.3: (Left) AvgJCTs on the *Physical* testbed, and (Right) CDF of job completion times for Sia predicted by the simulator (*Simulated*) compared to a run on *Physical* testbed (*Real*). Error bars represent the extreme values seen across 200 simulator and 4 physical cluster runs.

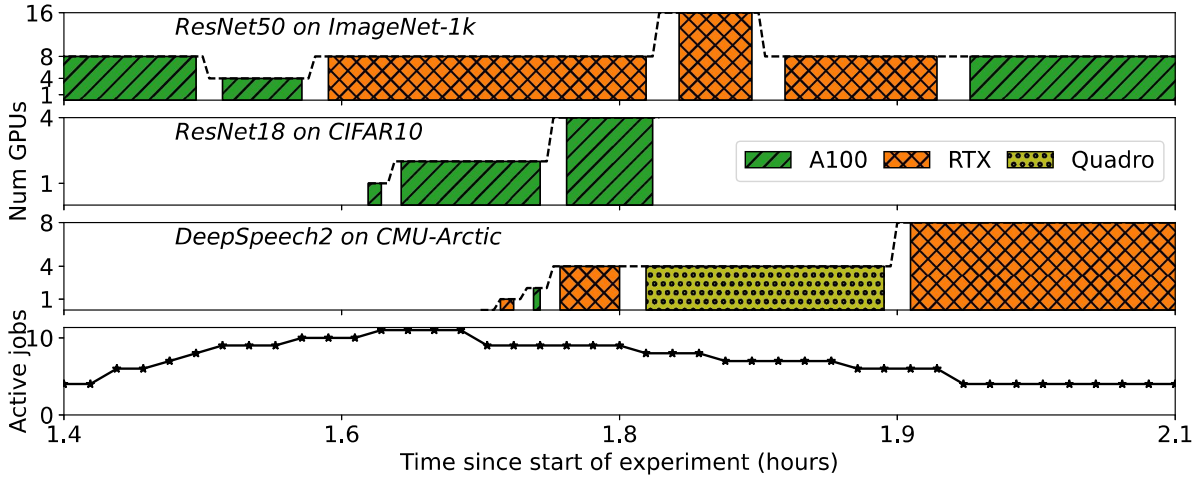


Figure 3.4: Resource allocations for three jobs in the Sia physical cluster experiment, along with number of active jobs in cluster. Colors indicate GPU type and whitespaces represent checkpoint-restore delays caused by Sia's scheduling decisions.

Trace	Policy	JCT		Makespan	Avg. GPU-hours/job	Contention		Avg. job restarts
		Avg.	p99			Avg.	Max.	
Philly	Sia	**0.6h ± 0.1**	**9.5h**	**14.2 ± 1.9h**	**4.0 ± 0.7**	**6.9**	**31**	**2.9**
	Pollux	1.0 ± 0.1h	14.9h	24.5 ± 7.9h	5.6 ± 1.1	7.2	42	5.8
	Gavel+TJ	1.9 ± 0.3h	30.0h	33.8 ± 8.6h	9.0 ± 6.3	9.9	56	5.7
Helios	Sia	**0.7 ± 0.1h**	**10.9h**	**14.9 ± 1.7h**	**4.8 ± 0.7**	**7.4**	**32**	**3.4**
	Pollux	1.0 ± 0.2h	15.0h	25.5 ± 8.0h	5.9 ± 0.7	6.9	47	5.3
	Gavel+TJ	2.5 ± 0.9h	38.7h	43.0 ± 10.9h	12.1 ± 3.7	9.2	48	7.5
new-Trace	Sia	**0.7 ± 0.1h**	**4.6h**	**52.2 ± 1.3h**	**3.0 ± 0.1**	**13**	**69**	5.0
	Pollux	1.5 ± 0.2 h	10.3h	62.3 ± 4.6h	3.4 ± 0.2	22	85	5.4
	Gavel+TJ	11.3 ± 3.0h	98.1h	110 ± 21.5h	6.4 ± 1.1	96	243	**4.5**

Table 3.3: Comparison of Sia, Gavel, and Pollux in the *Heterogeneous* setting. TJ is short for TunedJobs, Contention is the number of jobs contending for resources in the cluster.

Gavel+TunedJobs performs poorly compared to Sia for two reasons: (1) time-sharing overheads reduce the useful GPU time spent on training progress in a given round, and (2) using a batch size that fits the *smallest* (in memory) GPU leads to under-utilization of more

powerful GPUs. Pollux outperforms Gavel due to job adaptivity, but falls behind Sia for two reasons: (1) it treats heterogeneous hardware as homogeneous, failing to exploit performance heterogeneity, and (2) it can output placements spanning more than one GPU type; fixing them so they only span one GPU type forces some GPUs into idling, but it is better than using a mix of GPU types and running at speed of the slowest.

From Table 3.3, we see that Pollux restarts jobs twice as often as Sia for moderately congested clusters (**Philly** and **Helios**). This is because it optimizes job allocations in steps of 1 GPU, while Sia only allocates configurations with steps as large as an entire node (as defined in Section 3.3.3).

Congestion in newTrace. Compared to **Philly** and **Helios** traces, **newTrace** contains bursts of up to 100 jobs/hr during the busiest hour. Gavel struggles to handle these bursts, and as congestion worsens, this problem compounds creating a positive feedback loop: rising contention forces Gavel to swap jobs in/out more frequently, wasting significant GPU capacity on executing checkpoint-restore operations when GPUs are already scarce. As a result, the average and p99 JCTs for Gavel degrade far worse compared to Sia and Pollux. During peak congestion, Sia and Pollux both scale jobs down to just 1-GPU per job, resulting in a smaller gap between them. However, Sia's heterogeneity-aware scheduling better matches jobs to GPUs, improving cluster goodput over Pollux even during congestion.

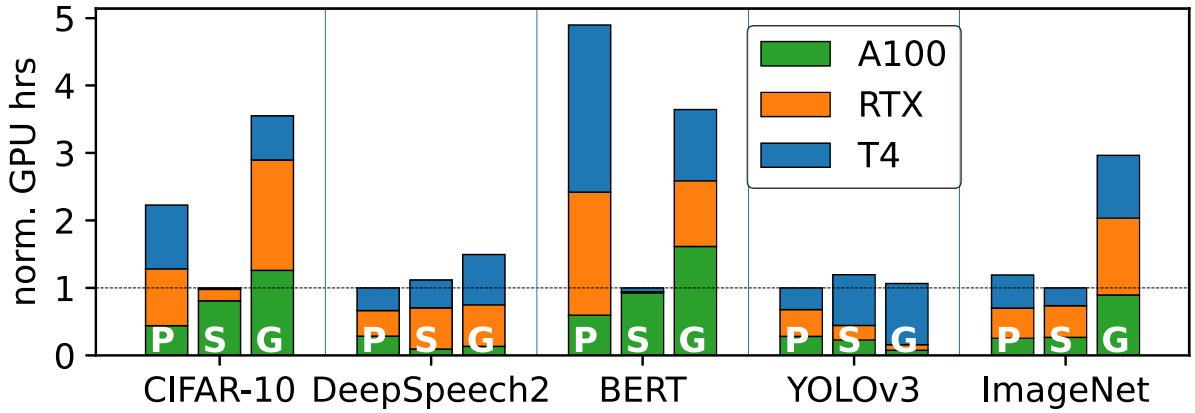


Figure 3.5: (Min-normalized) GPU hours consumed per model for Sia (S), Pollux (P), and Gavel (G) using **Helios** traces.

Matching jobs to GPU types. Figure 3.5 shows the average GPU hours used to train each model(Table 3.2) using **Helios** traces. Pollux is heterogeneity-unaware and has no distinct (job, GPU type) preferences, whereas Gavel and Sia are heterogeneity-aware and strongly prefer certain GPU types for particular models. Figure 3.5 shows that Sia allocates `BERT` models almost exclusively to `a100` GPUs, aggressively exploiting the heterogeneity in model goodputs across GPU types. Gavel's time-sharing approach, however, forces `BERT` jobs to rotate between `a100`, `rtx`, and `t4`GPUs, resulting in less-efficient execution. Similarly, Sia prefers to use `rtx` GPUs for `DeepSpeech2` and leaves the `a100` GPUs free for `BERT` models, achieving significant reduction in GPU hours consumed per job compared to Gavel's approach. On average, `YOLOv3` and `DeepSpeech2` models consume about 5% more GPU hours under Sia compared to heterogeneity-unaware Pollux, as Pollux gives them more GPU time to jobs on faster GPUs (out of randomness).

Workload Intensity. Figure 3.6 shows the average JCT as a function of average arrival rate for each of our evaluated schedulers. We sample jobs from **Helios** traces at various job arrival rates and evaluate them in the *Heterogeneous* setting with a fixed cluster of 64GPUs.

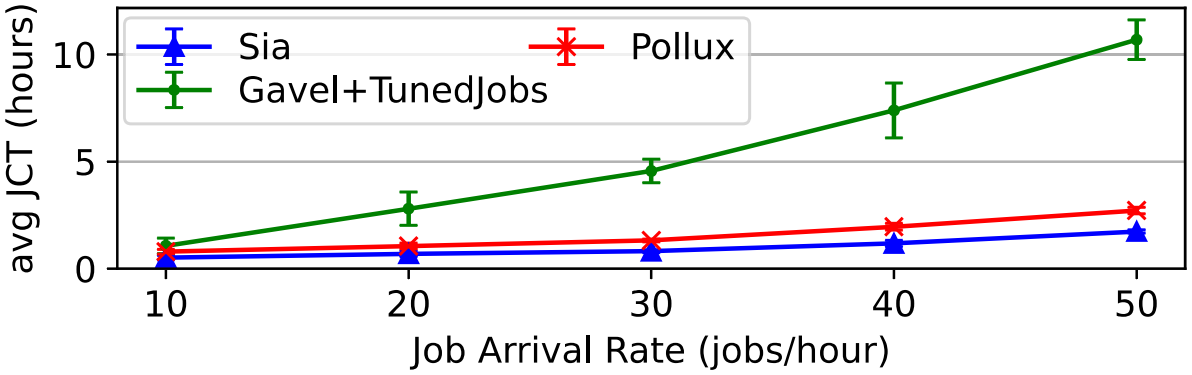
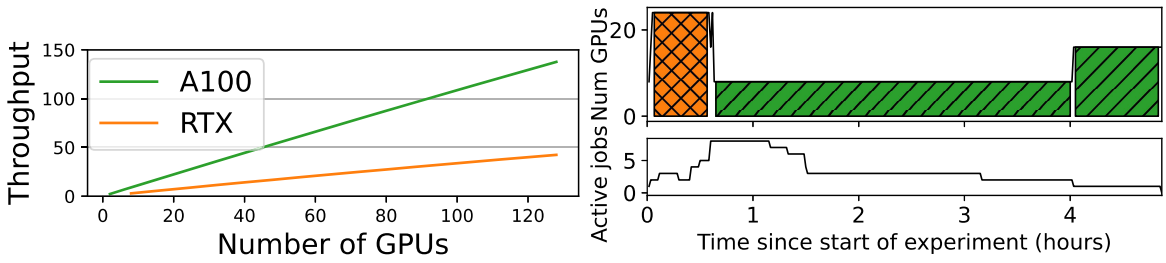


Figure 3.6: Avg. JCT for Sia, Pollux, and Gavel for various job arrival rates sampled using **Helios** traces.

Pollux and Sia outperform Gavel at larger arrival rates because they can scale down running jobs to use fewer GPUs rather than having to time-share GPUs. Sia consistently outperforms Pollux by 50–65% by aggressively matching jobs to preferred GPU types. As jobs arrival rates increase, jobs wait longer for resources, a problem that worsens with increasing congestion. However, an 8-hour job submission window is too short to observe these effects; on the 48-hour **newTrace** (Table 3.3), Gavel sees about 7× more contention compared to Sia (< 2× on the 8-hour traces), adding evidence to our claim.

3.5.3 Adapting hybrid parallel jobs

We simulate training of a 2.8B GPT model that uses pipeline model parallelism to scale to a few GPUs, and data-parallelism with gradient all-reduce to scale to multiple nodes. We borrow statistical efficiency profiles from `BERT` (closest match) to simulate a DL job finetuning the GPT model, and profile compute times for micro-batches and all-reduce times for different placements on `a100` and `rtx` GPUs to seed the simulator. Finally, we assume this job uses the commonly used Gpipe schedule [36] internally for PMP. We use 2 and 8 stages (1 per GPU) for `a100` and `rtx` GPUs, respectively, to account for the larger memories on `a100` GPUs. Each data-parallel replica runs 48 microbatches of size 1 each.



(Left) shows the hybrid-parallel GPT model's throughput scaling linearly with GPU count as computation dominates communication for this model. (Right) shows Sia adaptation

decisions for this model in response to changing cluster conditions. As expected, Sia scales the GPT model in response to congestion: scaling it down around the 1hr mark and back up around the 4-hr mark. Sia is the first cluster scheduler to support elastically scaling hybrid-parallel jobs on heterogeneous resources; supporting additional adaptation dimensions for PMP jobs is left to future work.

3.5.4 Attribution of primary benefits

We show the importance of having the scheduler directly address each key aspect (resource heterogeneity and job adaptability) by evaluating scenarios where only one is present.

Job adaptability, but not resource heterogeneity. We compare Sia against Pollux[79], Shockwave[111], Themis[59], and Gavel[68] using the **Philly** traces in a *Homogeneous* setting. We use TunedJobs for Shockwave, Themis and Gavel and re-tuned the job hyper-parameters to fully exploit the 64-GPU cluster. Table 3.4 (and the left-most bars in Figure 3.1) shows average and 99th percentile (p99) job completion times, average job makespan and average number of GPU hours consumed to train a job.

Policy	**JCT**		**Makespan**	**GPU hrs/job**
	Avg.	**p99**		
Sia	**1.9h**	**18.1h**	**21.4h**	**8.4h**
Pollux[79]	2.0h	19.3h	21.7h	8.6h
Shockwave[111]+TJ	3.6h	32.8h	35.0h	12.5h
Themis[59]+TJ	5.4h	44.7h	49.7h	17.2h
Gavel[68]+TJ	4.3h	37.1h	44.3h	15.3h

Table 3.4: Comparison of Sia against state-of-the-art in the *Homogeneous* setting. TJ is short for TunedJobs.

Pollux was designed for this scenario, and Sia matches it on all metrics, even outperforming Pollux as its ILP formulation can guarantee a global optimum (Pollux's genetic algorithm does not). Sia had fewer restarts compared to Pollux – 2.6 vs 5.1 restarts per job, so Sia wasted fewer GPU hours on checkpoint-restore operations. Shockwave [111] is the best inelastic scheduler as its objective optimizes for job progress and finish-time-fairness while penalizing schedules that result in large makespan. Themis (optimizing FTF) and Gavel (optimizing cluster throughput) fall behind Shockwave on all metrics. Sia and Pollux both exploit adaptivity and show a 50-70% improvement over the state-of-the-art inelastic baselines in all metrics.

Resource heterogeneity, but not job adaptability. The right-most bars in Figure 3.1 show average JCTs for the three schedulers, but with every job being treated as *rigid* – it must be run with the batch size and GPU count specified in the trace. Said differently, auto-scaling and co-adaptive batch size tuning is disabled for Sia and Pollux, evening the playing field with Gavel that cannot exploit job adaptivity. Even though Gavel was designed for this scenario, Sia outperforms it by about 25%. This can be atttributed to the fact that Sia explicitly optimizes for goodput (aka a max-sum-*goodput* policy) while Gavel optimizes for cluster throughput (max-sum-*throughput* policy). So, with inelastic jobs, Sia will always provides higher per-GPU goodput, resulting in better performance over Gavel. Pollux also

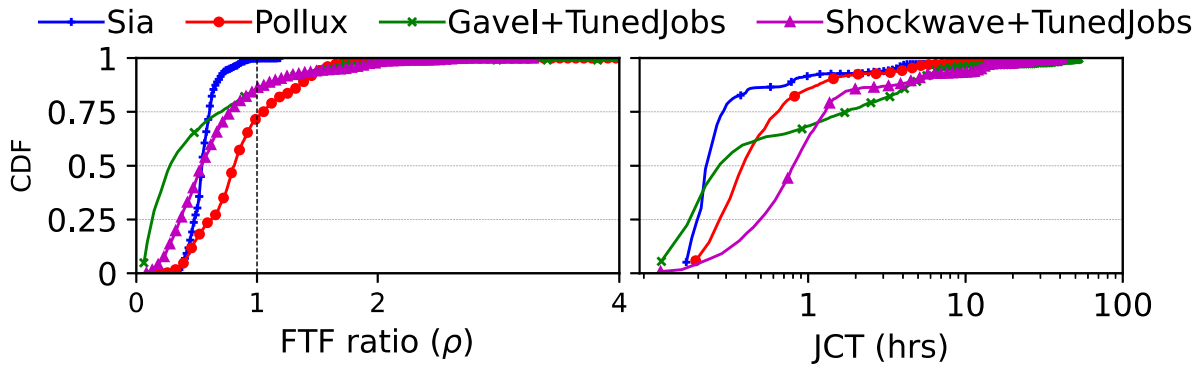


Figure 3.7: CDF of (left) Finish-Time Fairness ratio ρ[59], and (right) job completion times for Sia, Pollux, Gavel and Shockwave using **Helios** traces in the *heterogeneous* setting.

optimizes for sum of goodput, but produces worse JCTs as it is blind to and cannot exploit the GPU heterogeneity in the cluster.

3.5.5 Finish Time Fairness

Mahajan et al. [59] propose *finish-time fairness* (FTF [59]) as a metric that captures fairness of allocations to a job over its lifetime in a cluster. Assume job J sees an average contention (total number of jobs requesting resources) of N_{avg} and takes T_s to complete. Finish-time fairness (FTF) ratio ρ for the job J is defined as the ratio of a job's completion time in a shared cluster (T_s) to its JCT in an *isolated* and fair-sized cluster (T_f), where the isolated cluster contains $\frac{N_{gpus}}{N_{avg}}$, and N_{gpus} is the cluster size. We extend finish-time-fairness, defined originally for homogeneous clusters [59], to heterogeneous clusters as follows –

$$\rho = \sum_{G} P(G = g) \cdot \rho_g \tag{3.6}$$

where ρ_g is the FTF ratio for GPU type g.$P(G = g)$ is the probability that a random GPU in the cluster is of type g, given by $\frac{N_g}{N_{total}}$ where N_g is the number of GPUs of type g, and N_{total} is sum of N_g across GPU types. ρ_g is computed using the homogeneous-cluster definition [59] and only for the N_g GPUs of type g. If there exists only one GPU type, Equation (3.6) reduces to the homogeneous-cluster definition, preserving the metric's semantics. For heterogeneous clusters, ρ can be interpreted as the *expectation* of the FTF ratio taken over multiple GPU types.

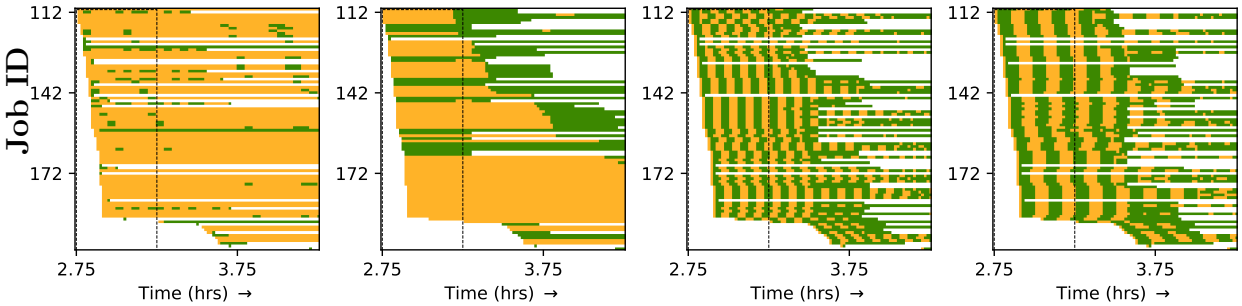
$\rho > 1$ means unfair executions: job finishes faster in isolation than using scheduler's policy, while $\rho < 1$ means sharing resources can improve job runtimes using idle GPUs. A vertical CDF for ρ with all jobs having $\rho \leq 1$ means a perfectly finish-time-fair scheduler. We are interested in three metrics: (a) worst FTF ratio [111, 59] across all jobs, (b) *unfair* job fraction [111](fraction of jobs with $\rho > 1$), and (c) CDF(ρ).

Figure 3.7 (left) shows the CDF of finish-time-fairness ratios for Sia, Pollux, Gavel and Shockwave using the **Helios** traces in a heterogeneous-setting. From Figure 3.7, we see visually that Sia is more fair (more vertical and <1) than Gavel, Pollux or Shockwave. Indeed, Sia provides a worst FTF ratio of 1.2 and unfair fraction of ¡0.3%.

The worst FTF ratio for the other schedulers in Figure 3.7 are: Pollux=4.6, Gavel=27.8, Shockwave=3.3. Their unfair job fractions are 28%, 15% and 14%, respectively. Shockwave does better than Gavel and Pollux, since it penalizes jobs with high FTF ratios, trading worst FTF ratio for unfair job fraction when compared to other schedulers. Sia achieves by far the lowest unfair job fraction and worst FTF ratio.

Figure 3.7 shows job completion times for Gavel, Shockwave, and Sia. Gavel and Shockwave prioritize either makespan or long jobs, resulting in worse outcomes for short jobs during periods of congestion. Sia adapts to prioritizing minimizing average JCT or makespan based on congestion levels: scale down long jobs during congestion to prioritize incoming short jobs (reduces congestion) and scaling out long jobs during reduced congestion to minimize makespan.

3.5.6 Congestion



(a) Gavel(5.6h) (b) Sia(2.65h) (c) Sia,$\tau = 180$s(2.29h) (d) Sia,$\tau = 300$s(2.32h)

Figure 3.8: Visualization of GPU activity for Gavel, and Sia with and without timeshare penalty for a 1.5hr period starting from $t = 2.75$hrs in the **Spike-240** trace. The numbers in parentheses represent the average JCT of the jobs submitted during the 1.5hr window.

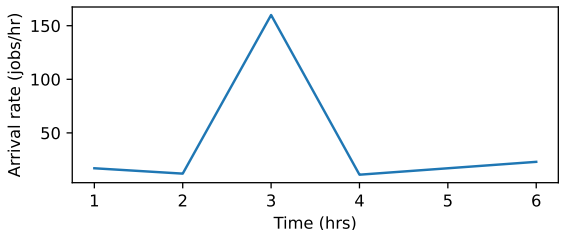


Figure 3.9: Job arrival rate for the 6-hour **Spike-240** trace.

In a production deployment, congestion in the cluster could worsen any minute as a result of the bursty nature of job arrivals. Further, as long jobs settle into the cluster, fewer GPUs are available to autoscale any new jobs and reduce congestion. To evaluate our scheduler in a congested setting, we sample a 6-hour trace (**Spike-240**) from the busiest period in our internal production traces with *double* the job arrival rate of **newTrace** at 40 jobs/hr for a total of 240 jobs submitted over the 6-hour window. We visualize the average job arrival rate for the **Spike-240** trace in Figure 3.9.

We visualize the resource allocation for a small subset of jobs (Job IDs 60-200) in the trace for a 1.5-hour period (hours $2.75 - 4.25$) in Figure 3.8 – a job J_i is colored *green* at time t if the job J_i received any service in the scheduling interval starting at time t. Note that *yellow* bands correspond to queueing delays for J_i i.e. scheduling intervals where a job did not receive any allocation and *white* bands correspond to periods before job submission and after the job completion. For the rest of this subsection, we focus only on the jobs submitted in the 30-minute window starting at hour 2.5 corresponding to a burst of submissions in the **Spike-240** trace (see Figure 3.9). For visualizations presented in Figure 3.8, this window corresponds to the region inside the dotted rectangles.

We run both Gavel and Sia schedulers on the **Spike-240** trace and visualize the GPU activity under the schedulers in Figures 3.8a and 3.8b, respectively. During the congested period, Gavel either runs a job J_i till the end of the scheduling interval (or completion, whichever is earlier) or idles it – job J_i sits in the scheduler queue with no allocation for the entire interval. For each job in Figure 3.8a, we can observe the sparse GPU activity resulting from the repeated execution and eviction (and thereby queueing) using Gavel. Unlike Gavel, Sia can respond quickly to changing loads by rapidly scaling down jobs to free up GPUs that are then allocated to incoming jobs. Indeed, we observe this behaviour for Sia in Figure 3.8b where some incoming jobs receive allocations as soon as they enter the system (e.g. Job ID 165). However, as the number of jobs vastly exceeds the number of GPUs in the cluster (around hour 3 in Figure 3.8b), Sia simply picks a subset of jobs to execute and recomputes allocations only if one of two conditions is met: (1) a new job arrives into or an existing job departs from the cluster, or (2) a running job changes its learning rate and this produces a stark change in statistical efficiency for the job, requiring re-optimization of resources.

As a result, Sia tends to *maintain* the current allocations (as seen in Figure 3.8b). So, a fraction of the new *Small* and *Medium* jobs submitted during this period receive allocation and finish execution quickly while the remainder wait in the scheduler queue for their turn at execution. We observe that this queueing delay could be significant and could span multiple hours. For example, consider the job corresponding to `jobID` = 180 in Figure 3.8b. This job waits at least 1.5 hours before starting execution, increasing the average JCT due to the queueing delay.

We propose amending that scheduler objective such that it provides each job with an opportunity to execute (preferably uninterrupted) on some GPU type in an attempt to reduce cluster load. To reduce the cluster load, *ideally*, we would like to execute jobs that need the fewest GPU hours to complete execution (aka Shortest Remaining Processor Time (SRPT)). However, since we do not have an estimate of the *remaining GPU time* for our jobs, we instead opt to time-share the GPUs between jobs in a clever manner. Given a large number of jobs and very few GPUs to allocate, Sia allocates exactly one GPU to each job in an attempt to minimize the penalty for not allocating any resources to a given job. In doing so, if multiple jobs prefer a single GPU type to the same extent, Sia simply picks one to execute till completion.

To time-share GPUs between these jobs efficiently, we vary the *no-alloc* penalty for jobs over time. Let $\lambda_n(i,t)$ be the no-alloc penalty for a job J_i. Let $s_i(\tau)$ denote the number of seconds for which the job J_i was running on some GPU in the last τ seconds. If a job has received service throughout the window (i.e. $s_i(\tau) = \tau$), we set $\lambda_n(i,t) = 0$ for the next τ

seconds. Otherwise, $\lambda_n(i,t) = \left(\frac{\tau - s_i(\tau)}{\tau}\right)$ where δ is the length of each scheduling interval (default $\delta = 60s$).Intuitively, this time-varying penalty attempts to provide τ seconds of GPU time to each job during periods of intense congestion. In doing so, all the new *Small* and *Medium* jobs should receive service as soon as they enter the system, albeit at the expense of some long-running *old* job. Figures 3.8c and 3.8d visualize the allocations under Sia using time-sharing penalties with $\tau = 180s$ and $\tau = 300s$ respectively. Visually, we see a significant reduction of queuing delay for *new* jobs before they begin execution at the expense of queuing some *older* long-running jobs. We see that the time-varying penalty produces *bands* of execution for each job indicating that the jobs run uninterrupted for as long as τ seconds before being pre-empted for time-sharing purposes.

Incorporating the time-sharing penalties reduces queueing delays significantly and reduces the average JCT for the ***Spike-240*** *trace by 14% when running Sia with* $\tau = 180s$ *over baseline Sia.* However, using $\tau = 180s$ might not be very efficient as each job spends an average of 30 seconds in container setup and checkpoint-restore, giving us about 150s of GPU time per job per 180 second window (30 seconds or 17% GPU time wasted per job). Setting $\tau = 300$ reduces the overhead to just 30 seconds per 300 seconds window (10% GPU cycles wasted), at the expense of increasing the average JCT by just 1.5% over $\tau = 180$. We conclude that $\tau = 300s$ provides the best compromise between wasted GPU cycles and average JCT for experiments using the time-varying *timeshare* penalty.

3.5.7 Policy overhead and scalability

In the 64-GPU *Heterogeneous* setting with **Helios** traces, Sia's policy optimization has a median and 95th percentile runtime of 96ms and 426ms, respectively(insignificant overheads for 60s scheduling rounds). Pollux takes longer with median and p95 times at 2.2s and 4.8s, respectively, indicating it may not scale to larger cluster sizes. Gavel is significantly faster with a median and p95 policy runtime of 13ms and 28ms, respectively.

Figure 3.10 shows scheduling policy runtime as a function of cluster size. Experiments are conducted using the *Heterogeneous* setting running **Helios** traces, scaled up to 2048 GPUs (traces scaled accordingly). Sia scales well, with a single-second runtime for policy optimization, enabling management of large clusters with thousands of GPUs and many GPU types. Pollux's genetic algorithm runs significantly slower (100x slower than Sia's ILP formulation) and struggles to find optimal solutions for large clusters due to an explosion of search space complexity (even without considering the extra complexity induced by heterogeneity). Gavel is much quicker, because it does not consider job-adaptation.

3.5.8 Sia Parameter Sensitivity

Fairness parameter (p). Figure 3.11 shows scheduler metrics for Sia, as a function of p computed using the **Helios** traces.

The impact of p on 99th percentile JCT is very evident as Sia allocates more GPUs to jobs that can take advantage of both scale and newer GPU types better (particularly `BERT` and `ImageNet` training jobs). Since these jobs also tend to run for a long duration, this drastically reduces their JCTs, bringing down the p99 JCTs the expense of average JCT. This

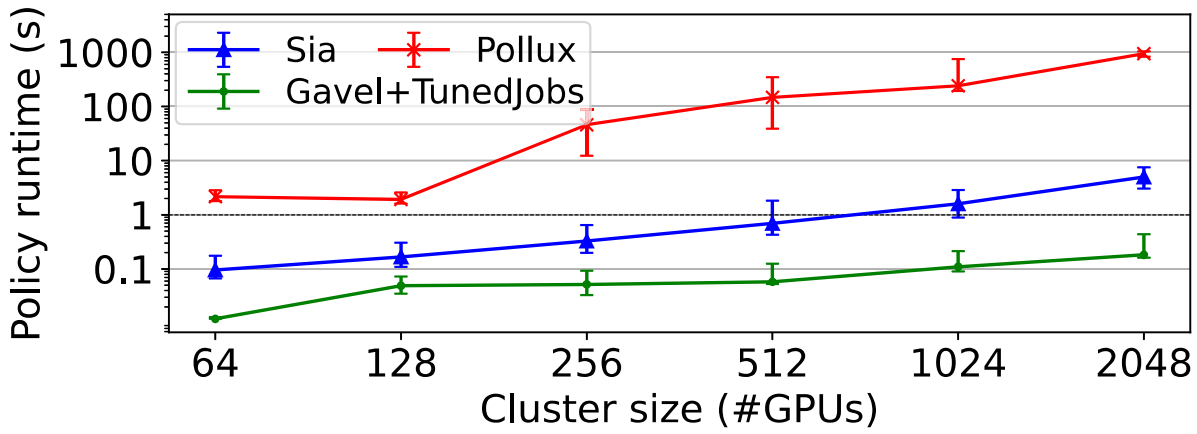


Figure 3.10: Median policy runtime for Sia, Pollux, and Gavel for various cluster sizes using proportionally-sized **Helios** traces. Error bars represent 25th and 75th percentiles.

also affects fairness of allocations and we notice that $p = 1.0$ has higher unfair job fraction (not shown here) compared $p = -0.5$. We choose $p = -0.5$ since it performs the best among all the p values we tested along the average JCT, average makespan and finish-time-fairness metrics.

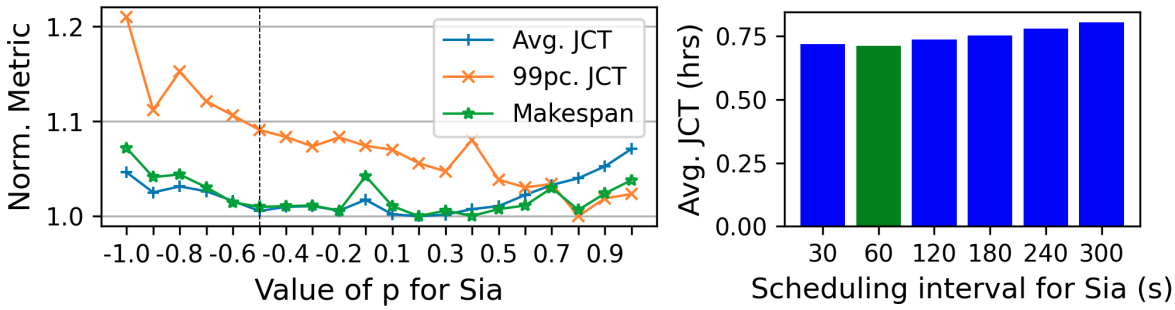


Figure 3.11: (Left) Trend for average and 99th percentile JCTs, and makespan for various values of p for Sia, and (Right) Average JCT for Sia for different scheduling round durations.

Scheduling round duration. We use a 60-second scheduling round duration for all our experiments. Increasing round duration from 60s to 300s increased average JCT for Sia by 333s (12%), while a shorter duration (30s) resulted in a higher rate of re-allocations and worsened average JCT. Sia's policy optimization takes less than 1 second, even for moderately sized clusters, and we choose a round duration of 60s since it performed the best. There was no significant change in p99 JCT or makespan observed.

Fraction of jobs supporting adaptivity. Sia supports adapting batch size, GPU count, and GPU type. Figure 3.12 shows the average JCT and makespan for Sia, normalized to all AdaptiveJobs (0% constrainted), as we vary the percentage of jobs with restrictions on which dimensions are adapted. *Strong-scaling* adaptivity constrains a job to use a fixed (user-supplied) batch size, but allows Sia to optimize the number and type of GPUs allocated to this job. *Rigid* jobs constrain a job to use a fixed batch size *and* fixed GPU count, but allow Sia to optimize the GPU type allocated. From Figure 3.12, we can conclude the following: (1) optimizing number of GPUs in addition to the GPU type improves avg JCT by 56%, and

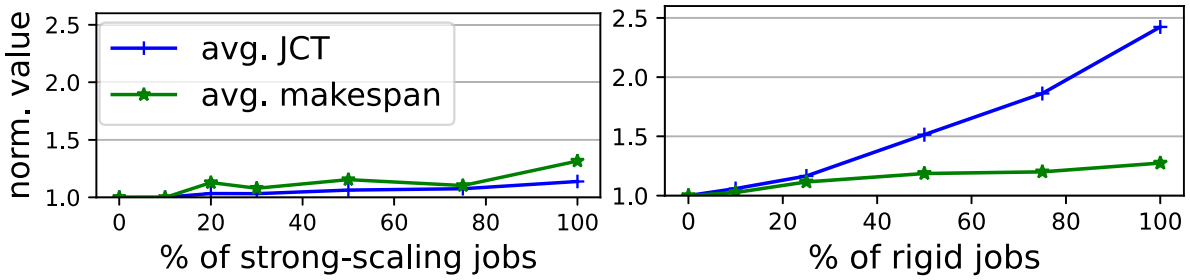


Figure 3.12: Average JCT and makespan for Sia on **Philly** traces as a function of the % of jobs that support (Left) only *strong-scaling* adaptivity, and (Right) no adaptivity (*Rigid*)

(2) additionally optimizing batch size (with number of GPUs and GPU type) improves avg JCT by another 13%.

Profiling Overheads. Profiling jobs incurs overheads, and profiling every possible configuration is impractical. Too little profiling and Sia might produce sub-optimal schedules, while too much profiling can waste cluster resources for marginal improvement in cluster efficiencies. To understand this trade-off, we evaluate Sia on **Helios** traces in three settings: (a) *Oracle* is an ideal setting where Sia knows a job's throughput on any set of resources (a best-case scenario for Sia) (b) *No Prof* does not profile initially and adopts a *profile-as-you-go* approach, resulting in zero profiling overhead (but no initial info for Sia); and (c) *Bootstrap* uses min-GPU profiles and extrapolates throughput for yet-to-run configurations (see Section 3.3), requiring ≈0.1 GPU hrs of profiling for each job—a middle ground between the extremes. Note that *Oracle* only serves as a baseline to quantify the effectiveness of Sia's bootstrap approach; it is impractical in most clusters, as it would need to profile 100s-1000s of placements across GPU types (1-10 GPU hrs/job).

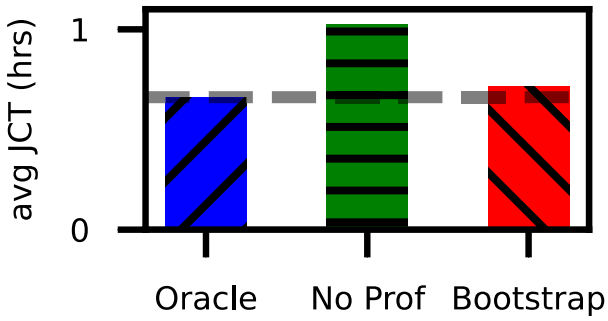


Figure 3.13: Average JCT for Sia on **Helios** traces with varying profiling overheads.

Sia with *Bootstrap* performs 30% better than *No Prof* and only 8% worse than *Oracle*, demonstrating the effectiveness of its bootstrapping mechanisms for heterogeneity-aware job adaptivity with minimal profiling overhead. We also found that profiling two GPU counts per GPU type performed worse that Sia's minimized approach. Sia's bootstrappping also scales well: for a cluster with 20 GPU types, bootstrapping adds ¡5% overhead to a job's execution.

3.6 Conclusion

Sia efficiently schedules adaptive DL jobs on heterogeneous resources, co-adapting each job's assignment of GPU count, GPU type, and batch size, resulting in increased DL cluster performance. Experiments show 30–93% reductions in average JCT, 28–95% reductions in p99 JCT and makespan, and 22–31% reductions in the unfair job fraction, when Sia is compared to existing schedulers. As such, Sia provides a critical component for emerging heterogeneous DL clusters.

Chapter 4

COpter-Sia: Scaling Sia scheduler to large GPU clusters efficiently using continual optimization

4.1 Introduction

In deep learning clusters, schedulers must allocate thousands of GPUs to jobs with diverse requirements while balancing throughput and fairness [67, 42]. Like the Sia scheduler, many scheduling policies are formulated as optimization problems (e.g., linear/mixed-integer linear programs) that are then solved by numerical solvers like GLPK [60](open-source) or GUROBI [29](commercial).

It is practical to run these policies at small-to-medium scales, but they can falter quickly as cluster sizes sizes grow—a reality driven by the recent explosive growth of computational infrastructure. Deep learning clusters now deploy over 100,000 GPUs (up from a few thousand in 2019 [43]), with a corresponding increase in users and jobs requesting these resources. Optimizing the scheduling policies at this scale with off-the-shelf solvers often exceeds practical time limits. For example, scaling up the Sia scheduling policy [42] from a 10k GPU cluster to 25k GPUs leads to an $\approx 100\times$ increase in solver run-times. As a result, the fraction of problems solved within the allocated round duration of 1 minute decreases from 85% to only 15%. Increasing round duration is not sustainable: GPUs released by completed jobs sit idle for longer while newly arrived jobs wait for GPU allocation in the scheduler queue, leading to reduced GPU utilization and increased job completion times. Practitioners have developed various strategies to scale optimization-based scheduling to large problem sizes: Warm-starting can be beneficial, but is not universally applicable and/or supported by existing commercial solvers [39, 29]; solver computation is not efficiently parallelized [102]; and GPU acceleration is not always applicable and/or beneficial [84] owing to the sparse nature of the solver compute. Forcibly terminating solvers early by setting a compute/time budget can return sub-optimal and/or infeasible solutions that are often meaningless. Partitioning problems by exploiting problem-specific properties can help scale many scheduling policies to larger cluster sizes with minimal effort, albeit at the cost of solution quality. POP [67] is a state-of-the-art problem-partitioning approach that partitions a large problem by randomly

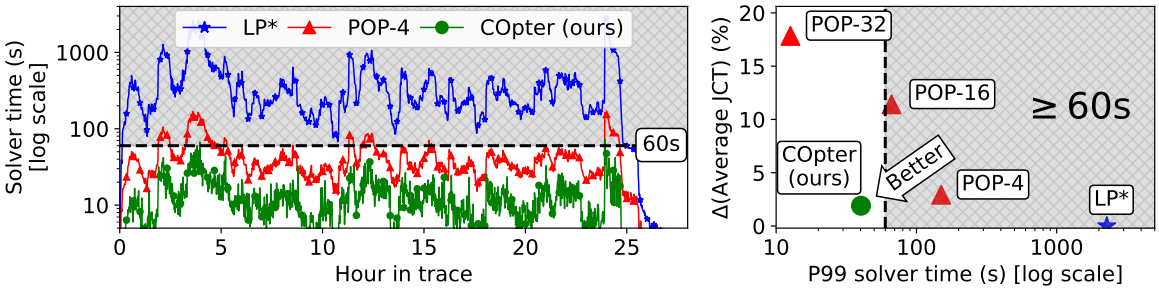


Figure 4.1: [Left] Solver runtime for three approaches of solving the Sia scheduler policy MILP [42] and their [Right] quality-runtime trade-offs. The approaches are evaluated on a trace with 100k jobs submitted to a 25k GPU cluster with 7 GPU types over 24-hours. *LP** solves the *LP-relaxation* of the MILP in each round independently using a commercial LP solver [39]. *POP-k* partitions the set of GPUs and jobs into k equal sizes (whenever possible) and solves the MILP for each partition using an MILP solver[24] in parallel. `COpter` uses continual optimization (see Section 4.3).

partitioning the set of resources and requests, solves subproblems on the partitions in parallel, and combines their allocations to exploit parallelism for large-scale scheduling under practical time constraints. However, there exists a quality-runtime tradeoff from applying POP, as can be seen in Figure 4.1[Right]: 32-way partitioning (*POP-32*) consistently meets scheduler deadlines but with poor quality allocations, whereas 4-way (*POP-4*) recovers higher quality allocations but frequently misses the 1-minute scheduler deadline.

All the approaches discussed thus far employ a commercial or open-source solver in a black-box manner. This means that the scheduling problem in each round is treated as an independent problem instance that is to be passed on to the solver. However, this leads to a fundamental mismatch between the modeling of round-based scheduling problems and the dynamics of the clusters to which these problems are applied: many large-scale clusters *evolve slowly* across rounds, exhibiting a certain structure across rounds. For example, in a *small enough* round duration (2-5 minutes) in GPU clusters, few GPUs fail, most jobs from the previous round continue executing without much change to their allocations, and few new jobs arrive into the cluster. By ignoring this structure and treating each round as an independent problem, existing methods forfeit an opportunity to amortize solving effort over time.

Adapting existing solvers to reuse computational effort from prior rounds is not straightforward. First, a small change to the input problem (e.g., the addition of just one new job) would require reconstructing solver-specific problem representations (e.g., constraint matrices), inducing significant setup cost for even the smallest change. Second, solvers need to reconstruct key internal transformations, such as matrix (Barrier methods [29, 60]) or basis (Simplex methods [39, 71, 60, 37]) factorizations, discarding all prior computational effort. Third, for MILP scheduling problems [42, 111], since converting non-integer valued variables to their integer counterparts dominates solver time, this further exacerbates the scalability challenge for large problems.

To address these limitations, we propose *continual optimization* — a paradigm that formulates scheduling problems in round-based scheduling as a sequence of interconnected

optimization programs with the goal of amortizing solver work/overhead over multiple rounds. We propose `COpter` as a system for performing continual optimization and show that it is possible to (a) reflect small changes in the scheduling problem to its optimization formulation using an efficient *differential* update interface to the problem; (b) exploit the slow evolution of the optimal solution to efficiently (and provably) warm-start subsequent problem solves; and (c) use lightweight heuristics to recover high quality integer solutions for MILPs that bypass expensive combinatorial searches entirely.

In this chapter, we demonstrate `COpter`'s effectiveness by applying to the Sia [42] scheduler for heterogeneous GPU clusters, discussed in the previous chapter, and defer discussion of `COpter` for other resource-allocation problems to the next chapter. In our evaluations, we find that `COpter` recovers high quality solutions, solves problems with millions of variables using just a few CPU threads, and reduces solver times by orders of magnitude compared to traditional use of state-of-the-art commercial solvers (Section 4.5.1 in this chapter, and Sections 5.2.1 and 5.2.2 in the next), with minimal quality loss (Sections 4.5.1 and 5.2.1). Compared to using POP, `COpter` can simultaneously improve allocation quality and reduce end-to-end allocator runtimes by 1.5–30×.

A paper describing continual optimization and `COpter` for many resource allocation problems was accepted at the ACM Symposium on Operating Systems Principles (SOSP) 2025.

4.2 Background and Motivation

In this section, we describe formulating scheduling problems as linear programs, the computation involved in solving linear programs using modern numerical solvers, and discuss challenges in scaling optimization-based scheduling to large problem sizes. We discuss other related work in Section 4.6.

4.2.1 Scheduling Problems as Linear Programs

Many scheduling policies can be formulated as *linear programs (LPs)* or *mixed integer linear programs (MILPs)* [68]. Given a set of resources (e.g., GPUs, CPUs) and resource requests (e.g., jobs, tasks), the goal with these programs is to find an allocation of resources to requests in a way that optimizes the given objective (e.g., maximizing throughput, minimizing job completion time, or ensuring fairness). Additionally, constraints are added to ensure that the resulting allocations are valid, realizable, and satisfy any policy requirements (e.g., priority requests must be satisfied before other requests).

Standard form linear program. Without loss of generality, let us consider linear programs in their *standard form*:

$$\begin{aligned} \min_{x} \quad & f(x) = c^T x \\ \textbf{subject to:} \quad & Ax \leq b,\ x \geq 0 \end{aligned} \tag{4.1}$$

where: $x \in \mathbb{R}^n$ is a vector of decision variables, $f(x) = c^T x$ is the *linear* objective function on the decision variables, $A \in \mathbb{R}^{m \times n}$ is the constraint matrix, $b \in \mathbb{R}^m$ is the right-hand side vector. Additionally, in *mixed-integer* LPs (MILPs), some or all variables are constrained to

integral values. MILPs are particularly useful in resource allocation as the decision variables in many problems are inherently discrete. For example, a resource is either allocated to some job or none, and a job is allocated some resource or none.

Example: GPU Cluster Scheduling. Consider a simplified GPU cluster scheduler that allocates a fixed number of GPUs to non resource-adaptive jobs. Let the cluster have N GPUs and M jobs. Let g_j be the number of GPUs requested by job j, and c_j be its throughput when using g_j GPUs. We wish to allocate resources to maximize the throughput of jobs in the cluster under the GPU capacity constraints. This can be stated as the following linear program:

$$\min_{x} \; -c^T x$$
$$\textbf{subject to:} \quad G \cdot x \leq N,\, x_j \in \{0,1\} \quad \forall j$$

where $c = [c_1, c_2, \ldots, c_M]$ is the throughput vector, $G = [g_1, g_2, \ldots, g_M]$ is a $1 \times M$ matrix, and $x = [x_1, x_2, \ldots x_M]$ is the vector of variables. The variable $x_j = 1$ iff g_j GPUs are allocated to job j (0 otherwise).

4.2.2 Solving linear programs

The LP/MILP formulation for a scheduling problem is solved in two stages, with an additional third stage for MILPs.

1. Program compilation. This stage transforms the scheduling problem into a form that existing solvers can consume. For example, in the example GPU scheduling problem in Section 4.2.1, this involves (a) ordering the M active jobs and assigning them an index in $\{1 \ldots M\}$, (b) building the c vector and (c) building the G matrix.

2. Program solving. The solver then uses one or more optimization algorithms to find an optimal solution to the *compiled* linear program. If the input program constrains some variables to take only integral values (MILPs), this stage ignores such constraints and finds an optimal solution to the *LP relaxation* of the MILP instead.

3. (Optional) Integerization. This stage only applies to MILPs and converts any possibly non-integer values for variables with integer constraints to an integer value. Techniques like *branch-and-cut* often require alternating between the *solving* and *integerization* stages as it progressively assigns integral values to variables with integer constraints [60].

High-level modeling frameworks, such as CVXPY and Google OR-Tools [21, 76], provide powerful abstractions to specify optimization problems common in systems, like resource allocation. Developers define problems programmatically using symbolic variables and functions mirroring the mathematical formulation that are then automatically compiled into solver-compatible representations, shielding users from low-level implementation details.

4.2.3 Scalability challenges in solving linear programs

Traditional approaches to round-based scheduling reformulates and solves one scheduling problem *from scratch* in each round. This means that, in each round, we recompile the current linear program from scratch, pass this to an LP solver, and use a procedure like

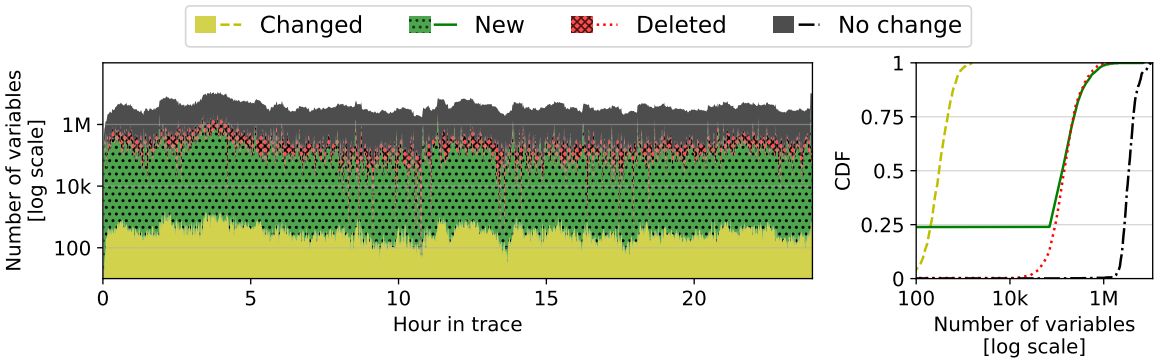


Figure 4.2: Changes to the solution between consecutive scheduling rounds over a 24-hour period in a 25k GPU cluster running Sia policy [42]. Most variables remain unchanged: fewer than 0.01% of variables change values and at-most a few % are added/removed between consecutive rounds.

branch-and-bound [29, 60] to recover integer-valued solutions to the problem as needed. As problem sizes grow (e.g., clusters with tens of thousands of GPUs/jobs), each stage in solving an LP faces scalability challenges, which we discuss below.

Program re-compilation discards prior computational effort. For any scheduling problem, the bulk of the compilation time is spent in constructing the constraint matrix. Constraint matrices are often quite sparse— each row in the matrix corresponds to a constraint on resource or request, and applies to a small subset of variables at once. For large scheduling problems, even the sparse representation can contain tens of millions of non-zero values and can take tens of seconds to populate in each round. Further, since current approaches trigger full recompilation for small changes to the problem (e.g., add/remove resources/requests), they discard all prior computational effort spent in creating the constraint matrices for previous rounds. We show in Section 4.3 that it is possible to reduce this overhead of recompilation every round by adopting a problem-level *differential* interface, significantly reducing end-to-end runtimes.

Independently solving LPs ignores problem evolution. Many large scale scheduling problems *evolve slowly*, i.e., between two consecutive rounds t and $t+1$, few resources are added/removed and few jobs are added/removed (relative to the current sizes of resource and request sets). While, in general, small changes in problems can still induce large changes in the optimum solution, we observe that for scheduling problems, the optimal solutions in rounds t and $t+1$ only differ by a small amount (see Figure 4.2).

State-of-the-art LP and MILP solvers [29, 39] use either an Interior-Point (Barrier) or Simplex algorithm, both of which maintain internal state that cannot be easily updated to benefit from the slow evolution of the problems. Interior-Point methods maintain matrix factorizations and adding new variables and/or constraints to the problem requires a refactorization. Similarly, the Simplex algorithm maintains a *basis* factorization. While theoretically amenable to warm-started updates when adding variables and/or constraints, in practice this process necessitates expensive updates to the factorization and triggers numerous pivot operations to restore feasibility. Solvers using first-order algorithms like ADMM [9] or Proximal Point Algorithm [73] often maintain very little internal state, so they do not suffer

from the same issues, and form the basis for our approach.

Warm-start not well exploited by existing approaches. Consecutive scheduling problems often exhibit a significant overlap in the set of resources and jobs. Often, even the optimal solutions from successive rounds, x^*_{t-1}, x^*_t are also close to each other in Euclidean distance (ℓ_2 norm). Intuitively, one might expect that initializing a solver with a point already close to optima would lead to faster convergence. Conventional solvers, however, face challenges in effectively exploiting this ℓ_2 proximity. The runtime of Simplex methods depends on traversing the vertices of the constraint polytope (defined by the constraint matrix A and vector b). A small $||x^*_t - x^*_{t-1}||_2$ does not guarantee a short path between the corresponding vertices. Furthermore, even minor modifications, such as removing a single constraint, can theoretically lead to worst-case exponential runtime for deterministic Simplex. Interior-Point methods, while capable of warm-starting, measure the quality of an initial point relative to the problem-specific *central path*. A small $||x^*_t - x^*_{t-1}||_2$ does not necessarily imply proximity to this central path, making the practical benefit of such warm-starts difficult to predict or control [105, 46]. First-order methods [9, 4, 91, 72] can benefit from a small $||x^*_t - x^*_{t-1}||_2$, but may suffer from slow practical convergence to acceptable tolerances.

Combinatorial explosion for MILPs. Mixed Integer formulations of scheduling problems are typically solved using a two-stage process. First, a traditional, often a Simplex-based, LP solver is used to find an optimal solution to the *LP-relaxation* of the MILP, where integer constraints are temporarily ignored. However, this solution could assign non-integral values to variables that require integers, leading to integer infeasibilities, that are resolved in the second stage using an algorithm like *branch-and-cut* [60, 29, 39]. Branch-and-cut systematically explores a tree of refined LP problems: it selects an integer-infeasible variable, say x_j, creates branches by adding constraints that push its value towards adjacent integers (e.g., $x_j \leq \lfloor x^*_j \rfloor$ and $x_j \geq \lceil x^*_j \rceil$), and solves the resulting LPs recursively by choosing another x_k for $k \neq j$. This continues until we find a solution with zero integer infeasibilities and whose objective value is close (i.e., within a *MIP gap*) to the LP relaxation's optimum. However, the number of LPs to solve can scale exponentially, making this stage computationally prohibitive for large problems. We show in Section 4.5 that problem-specific heuristics are effective in circumventing this bottleneck by efficiently finding feasible integer solutions with negligible quality impact.

4.3 Continual Optimization

In this section we first introduce a notion of *slowly evolving* linear programs and then describe the *continual optimization* paradigm along with its goals. We then describe `COpter`— our proposed approach for continual optimization of large-scale round-based scheduling problems.

4.3.1 *Slowly Evolving* LPs and Continual Optimization

As discussed in Section 4.2.3, in many real-world systems, scheduling problems and their associated optimal solutions evolve *slowly* over time (see Figure 4.2). Naturally, such a *slowly evolving* nature suggests the idea of reusing prior allocations or computation across rounds

to improve run-times: if the scheduling problem in each round largely remains unchanged, then the solve times should remain small.

Let us describe some notation needed to formalize the notion of slow evolution. We use t to index successive rounds in a cluster scheduling process. With a minor overloading of the notation, let $\min_x \text{LP}_t(x)$ denote the LP/MILP solved in round t. In the standard form (Equation (4.1)), $\text{LP}_t(x)$ can be fully defined using a 3-tuple (c_t, b_t, A_t), where c_t is the cost vector, and A_t, b_t are the constraint matrix and vector, respectively. Let x_t^* be an optimal solution to the problem in round t, i.e. $x_t^* \in \arg\min_x \text{LP}_t(x)$. The size of these LPs is determined by the number of variables n and constraints m (also the dimensions of $A \in \mathbb{R}^{m \times n}$). We define problem dimension $\eta_t = \max(m_t, n_t)$ as the larger of the two for round t, and $\eta_{\max} = \max_t \eta_t$ as the largest problem dimension over all rounds $1 \leq t \leq T$.

Definition 1 (**Slowly Evolving Linear Programs**). *We consider a sequence of round-based scheduling problems* $\{\text{LP}_t(x)\}_{t=1}^T$ *as slowly evolving, if there exists some small* $\alpha, \beta \in [0, 1]$, $\alpha \ll 1$, $\beta \ll 1$ *such that,*

1. $\mathcal{P}_\eta(T) = \sum_{t=2}^T |\eta_t - \eta_{t-1}| = \mathcal{O}(T^\alpha \eta_{\max})$, *and,*
2. $\mathcal{P}_x(T) = \sum_{t=2}^T \|x_t^* - x_{t-1}^*\|^2 = \mathcal{O}(T^\beta B)$,

where $B > 0$ *is such that* $\|x_t\|^2 \leq B$ *for all feasible* x *and all* t.

The quantities $\mathcal{P}_\eta(T)$ and $\mathcal{P}_x(T)$ are often referred to as *path-lengths* as they represent the *trajectory* of a sequence of values: problem dimensions in the case of $\mathcal{P}_\eta(T)$ and the optimal solutions in the case of $\mathcal{P}_x(T)$. The quantities α and β are used to interpolate between two cases—one where consecutive programs do not change at all (both $\alpha = 0$ and $\beta = 0$), and the other where they change drastically (either $\alpha = 1$ or $\beta = 1$). For example, consider the hypothetical scenario in GPU resource scheduling where an adversary cycles between taking all the GPU nodes offline and then bringing them back online. In such a scenario, in rounds where no nodes are available, the optimal allocation x^* is all zeros and in rounds where the GPU nodes come back online, x takes on non-zero values. Assuming $x_i \in [0, 1]$, the maximum ℓ_2 distance between consecutive optima is at most $\sqrt{n}$ (i.e. $\|x_t^* - x_{t-1}^*\|^2 \leq n$), and therefore, $\mathcal{P}_x(T) = \sum_{t=2}^T \|x_t^* - x_{t-1}^*\|^2 \leq Tn$ and $\beta = 1$. Similarly, on the other extreme, consider the scenario where the resources and requests, and therefore the allocations all remain unchanged. This means that $x_t^* = x_1^* \, \forall \, t \leq T$, so $\mathcal{P}_x(T) = 0$ and $\beta = 0$.

Continual optimization. Given the model of round-based scheduling as a sequence of programs, $\{LP_t(x)\}_{t=1}^T$, and path lengths $\mathcal{P}_x(T)$ and $\mathcal{P}_\eta(T)$ as defined in Definition 1, the goal of continual optimization is to develop approaches where the total end-to-end time to allocation, say $\tau(T)$, depends on problem-specific path lengths, $\mathcal{P}_x(T)$ and $\mathcal{P}_\eta(T)$ as follows:

$$\tau(T) = \mathcal{O}\big(\mathcal{P}_x(T), \mathcal{P}_\eta(T)\big)$$

Scheduling problems have small $\mathcal{P}_x(T)$ and $\mathcal{P}_\eta(T)$ (i.e $\alpha \ll 1, \beta \ll 1$), and so we can expect continual optimization to enable significant speedups in runtime when compared to treating each problem independently (see Figure 4.3 [Right]). This notion of problem-dependent bounds have recently received extensive attention in the *online optimization* literature [109, 108, 8] in the context of *dynamic regret*, the sum of optimality gaps across all T rounds.

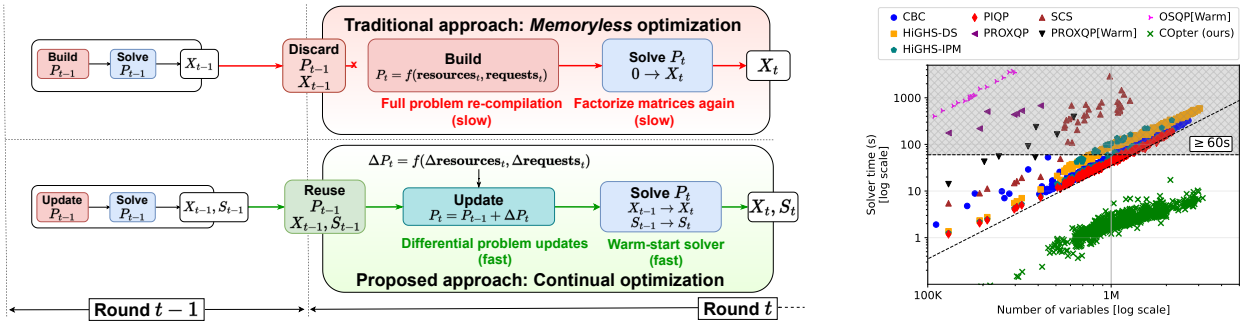


Figure 4.3: Existing approaches for optimization round-based scheduling problems cannot scale to large sizes because: (a) any changes to a problem (from adding/removing resources and/or jobs) require problem recompilation; (b) recompiling a problem discards any solver work done for previous rounds; and (c) warm-starting is either not supported or often not beneficial in reducing solver runtimes. *[Left]* We propose *continual optimization* and implement a prototype (`COpter`) with techniques designed for (a) efficient problem manipulation, (b) solver state re-use, and (c) efficient warm-starting. *[Right]* For the LP-relaxation of the Sia ILP policy [42], `COpter` using continual optimization is a few orders of magnitude faster than memoryless optimization with any open-source solver, and scales better to larger problem sizes (see Section 4.5 for comparisons with commercial solvers).

Continual optimization differs in that we additionally care about the end-to-end runtime for the scheduling problem.

Note that since programs across rounds are allowed to have different dimensions η_t, the distance $\|x_t^* - x_{t-1}^*\|^2$ is defined by padding x_t^* and x_{t-1}^* so their dimensions are exactly $\eta_{\max}$, the largest problem dimension. We also note that the definition of slowly evolving programs also admits *bursts* of resources and/or jobs as long as such events are not too frequent in the measured T rounds.

4.3.2 `COpter` for Continual Optimization

In this section, we describe `COpter`, our proposed approach for continual optimization of round-based scheduling problems. Figure 4.3[Left] shows the high-level overview of the scalability challenges in existing *memoryless* approaches and our proposed approach using continual optimization. `COpter` proposes techniques to address scalability challenges in each of the three stages involved in solving scheduling problems as described in Section 4.2. Taken together, these techniques enable `COpter` to efficiently reuse computational effort from prior rounds to reduce overall runtimes.

Differential interface for problem updates. `COpter` uses a differential interface at the problem level that reflects changes to the scheduling problem from the addition and/or removal of resources and requests between rounds.

To support such differential updates to the problem we need to efficiently support the following methods:

- `add-request(job ID, num variables)`: modifies A by (a) appending *num variables* columns to A, c, and x, (b) appropriately filling in the newly appended columns for all

constraints affected by adding this job, and (c) adding any additional constraints (one row per constraint) for the new job to A and b.

- `remove-request(job ID)`: modifies A by (a) deleting columns associated with the job, from A, c, and x, and (b) removing any associated constraints from A.
- `add/remove-resources(resource ID, count)`: modifies b to reflect the change in number of resources of type *resource ID* by $\pm$*count* units as appropriate.

To efficiently implement such an interface, we manipulate the constraint matrix A in-place. Since the matrix A is mostly sparse (i.e. few non-zeros per row), it is common to store the non-zero entries in A in a contiguous array with a majorization either along the rows or the columns of A — compressed sparse row (CSR) representation for row level access, and compressed sparse column (CSC) for columns. However, a contiguous memory layout does not permit efficient modifications to the matrix (like adding or removing rows/columns) without incurring a full copy cost of $\Theta(\min\{mn, \text{nnz}(A)\})$. So, even when existing solvers provide interfaces for incremental modifications [29, 39], the cost of recompilation from an existing in-memory matrix does not improve significantly.

`COpter` trades the cache-friendly, contiguous memory layouts for a List-of-List (LoL) representation as a compromise between the latency advantages of contiguous layout and the ease of modification of a non-contiguous representation. We use pointer manipulations to efficiently add and remove rows (i.e. add/remove constraints) in the row-major LoL format. The corresponding column operations are still expensive, but only affect a few rows because A is sparse, and are trivially parallelized over rows. We show in Section 4.5.2 that differential problem updates cut compilation times by upto 30× compared to recompilation each round.

Factorization-free, efficiently warm-started LP solvers. State-of-the-art solvers rely on internal problem transformations of some form, often a factorization of the constraint matrix A, to speed up solving times. In the context of continual optimization, changes to the A matrix from adding/removing resources and/or jobs are common operations. However, these changes make the factorizations from prior rounds invalid, forcing us to discard all prior computational work (or adopt expensive procedures to restore factorizations to a reusable state). Additionally, in a slowly evolving system, the optimal allocation for a majority of the requests remains unchanged, and so the previous solution serves as a good starting point for the solver in the current round. However, Simplex and Interior-Point(Barrier) based methods cannot be easily warm-started with guesses that are close in ℓ_2 norm to the optimal solution. A simple workaround is to adopt a solver based on first-order methods such as ADMM [9] or Proximal Point Algorithms (PPA) [73] that maintain an easily recomputable internal state.

We choose the Proximal Point Algorithm (PPA) as our workhorse for its practical convergence behavior, ability to be warm-started and its ability to recover largely integral solutions for LP-relaxations of MILPs. PPA transforms $\text{LP}_t(x)$ into a sequence of quadratic programs whose solutions converge to that of $\text{LP}_t(x)$. Successive iterations improves both feasibility and optimality of the current iterate by a multiplicative factor. We efficiently warm-start PPA by starting its first iteration for round t with x^*_{t-1} – the optimal solution to $\text{LP}_{t-1}(x)$ from round $t-1$. To solve the quadratic *subproblems*, we employ a fast *coordinate descent (CD)* [104] algorithm. This enables our solver to (1) maintain minimal internal state, as coordinate descent is factorization-free, (2) benefit from sparsity in the constraint

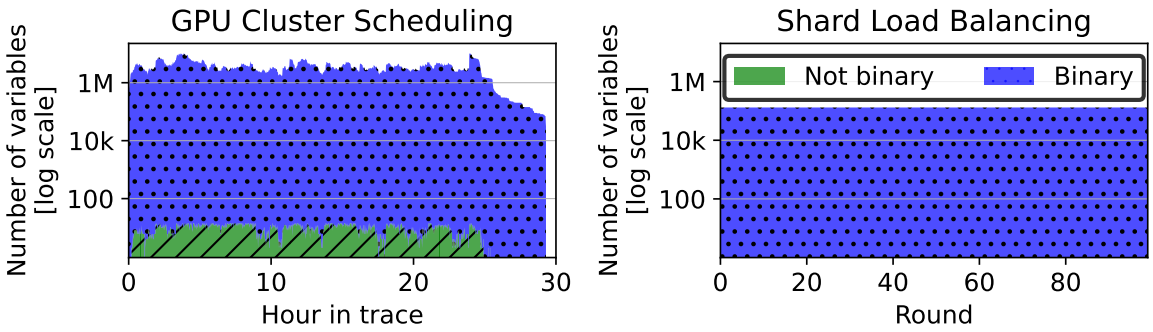


Figure 4.4: Solution composition for LP-relaxations of MILP formulations for two scheduling problems using a commercial simplex based LP solver (CPLEX [39]): Sia scheduler policy for GPU cluster scheduling [42] and Shard Load Balancing (see Section 5.1.1). In the optimal solution to the LP-relaxations, $\geq 99.99\%$ values are either 0 or 1 with $\leq 0.01\%$ values in $(0, 1)$.

matrix and solution vectors through sparse matrix-vector (*SpMv*) operations and an *active-set* strategy, and (3) provably benefit from a good initial guess. At large scales, our approach solves problems orders of magnitude faster than the alternative of solving linear systems, all while using just a few CPU threads (see Figure 4.3 and Section 4.5.1).

Lightweight integerization for MILPs. The applicability of traditional MILP solvers to large-scale optimization-based scheduling is often limited by the runtime cost of enforcing integer constraints with minimal loss of optimality. After solving the LP relaxation of the MILP, techniques like branch-and-cut [60, 39, 29] are used to fix integer infeasibilities. Since this integer resolution stage frequently dominates the overall solution time, it creates a significant scalability bottleneck.

We find that for many resource allocation MILPs (see Sections 4.4.1 and 5.1.1), solvers using Simplex[39] or `COpter`'s PPA (see Figure 4.4) often produce solutions to the LP relaxation with only a few integer infeasibilities, in contrast to ADMM-based solvers [9, 72]. While conventional MILP solvers invoke expensive branch-and-cut [60, 39, 29] to guarantee optimality of the integer-feasible solution, this is a bottleneck step and `COpter` introduces lightweight post-processing, using *shims*, to produce integer-feasible solutions from the optimal solution to the LP-relaxation of the MILP. *Shims* operate heuristically: they round fractional variables and then apply fast, local corrections to restore constraint feasibility, and are deliberately designed to prioritize problem feasibility over optimality.

The crucial result is that `COpter` delivers high-quality, feasible solutions for large-scale MILPs (millions of variables) in milliseconds, effectively circumventing the traditional bottleneck with minimal impact on quality (Sections 4.5.1 and 5.2.1).

4.4 COpter-Sia

We will first recap the MILP formulation of Sia policy and then describe how COpter realizes continual optimization for the Sia policy. We defer evaluations of COpter-Sia to Section 4.5.

4.4.1 Sia policy — a quick recap

The MILP formulation for Sia policy starts with the goodput matrix G where G_{ij} is the normalized goodput for job i estimated on configuration j. X is an allocation matrix (same dimension as G) and job i is allocated configuration j if $X_{ij} = 1$, or no resources if $X_{ij} = 0$. Let $\texttt{NGPU}(C_j, k)$ be the number of GPUs of type k in configuration C_j, and N_k be the total number of GPUs of type k in the cluster. The MILP formulation for the Sia policy is as follows:

$$\max_X \sum_{i=1}^{N_J} \sum_{j=1}^{N_C} X_{ij} \cdot G_{ij} + \lambda \sum_i \left(1 - \sum_j X_{ij}\right) \qquad (a)$$

$$\textbf{subject to:} \quad \sum_{j=1}^{N_C} X_{ij} \leq 1 \quad \forall i \qquad (b)$$

$$\sum_{i=1}^{N_J} \sum_{j=1}^{N_C} \texttt{NGPU}(C_j, k) \cdot X_{ij} \leq N_k \quad \forall k \qquad (c)$$

$$X_{ij} \in \{0, 1\} \quad \forall\, i, j \qquad (d)$$

where (a) is the objective that maximizes cluster-wide goodput with an incentive $\lambda \geq 0$ to reduce scheduler queue occupancy[42], (b) ensures each job is allocated at-most one configuration, (c) ensures that each GPU type k is not over-subscribed (i.e. allocations do not exceed capacity), and (d) ensures that for any job, choice of a configuration is binary.

Applying COpter. We first relax the binary constraints on X_{ij} to $0 \leq X_{ij} \leq 1\ \forall i, j$. Then, we solve the relaxed LP and apply a shim to its optimum solution to recover binary and feasible allocations in each round. For each job, the shim simply picks the configuration with highest weight in the solution to the LP-relaxation without violating GPU constraints. If it finds no such configuration for the job, it receives no allocation in this round.

The resulting allocations are both binary and feasible, but possibly sub-optimal. We show in Section 4.5.1 that COpter's allocations result in negligible increases to average job completion times and makespan in clusters with 10,000+ GPUs.

4.5 Experiments

This section evaluates the effectiveness of applying *continual optimization* via COpter to the Sia scheduler policy formulated as an MILP. In the next chapter, we will describe and evaluate COpter when applied to resource-allocation problems from other domains like elastic database and WAN traffic engineering systems.

We run the Sia policy [42], in simulation, with 60 second round on two clusters with 7 GPU types, and 10k and 25k GPUs, respectively. We sample a job traces with 40k and 100k jobs submitted over 24-hours from a real GPU datacenter trace [99], and map each sampled job to one of 10 representative jobs (same as prior work [42, 79]).

Cluster size: 10k GPUs			
	Avg JCT (hours)	**Makespan** (hours)	**Solve time** (p99, seconds)
LP*	**0.35**	**29.3**	233.4
POP	0.41	31.0	**2.9**
`COpter`	0.36	29.4	6.5
Cluster size: 25k GPUs			
LP*	**0.35**	**29.3**	2277
POP	0.39	30.9	66.5
`COpter`	0.36	29.4	**40.3**

Table 4.1: Summary of experiments for the GPU cluster scheduling case study using the Sia policy [42] for 10k and 25k GPU clusters using `COpter`, POP and a commercial LP solver[39].

Testbed. We run our experiments on a 2-socket AMD EPYC 7742 64-core processor with 1TB of DDR4 memory, but limit solvers to 8 threads (no significant speedup is observed beyond 8 threads [102]).

4.5.1 Effectiveness of applying `COpter` to Sia

Evaluated approaches. We compare `COpter` against two approaches: (a) *POP* [67] splits the ILP formulation of the Sia policy into 16 sub-problems with equal resources (and jobs whenever possible), solves subproblems in parallel (using `ray` [64]+`cvxpy` [21]) with a Simplex-based solver[24], and (b) *LP** solves the LP-relaxation of the Sia ILP using a commercial solver [39] and obtains binary allocations each round using the *shim* described in Section 4.4.1.

Metrics. We are interested in the following metrics: (1) *average job completion times* (avg. JCTs) is the average time taken to execute a job from submission to completion, averaged over all jobs in the trace (lower is better), (2) *makespan* is the time taken for all jobs in a trace to complete (i.e. time between first job submission and last job completion, lower is better), and (3) *solver runtime* measures the end-to-end runtime for the scheduler policy (including the time spent in *shim* for `COpter` and *LP**, lower is better).

We summarize the results of our experiments in Table 4.1 and the CDF of the solver runtimes in Figure 4.5. From Table 4.1, we see that for both cluster sizes, `COpter` is $30-40\times$ faster than *LP** and achieves a similar average JCT and makespan. POP's 99th percentile solve time is faster than `COpter` for the 10k GPU cluster, but exceeds the 60-second threshold for the 25k GPU cluster owing to the combinatorial explosion for the MILP solved in each subproblem. Despite solving the ILP formulation directly, POP's solutions result in $11-17\%$ and 6% increase in average JCT and makespan, respectively. This also highlights the efficacy of our *shims* – despite using a heuristic to obtain integer solutions to the MILP, solutions for both `COpter` and *LP** result in lower average job completion times and makespan. This indicates that the combinatorial explosion from the *integerization* stage can be bypassed for large resource-allocation problems without much penalty.

Figure 4.5 also highlights another characteristic of MILP solvers: the number of variables in the ILP does not always dictate the solver runtime due to the unpredictability of the runtimes for the *integerization* stage. Since POP solves the ILP formulation directly in each

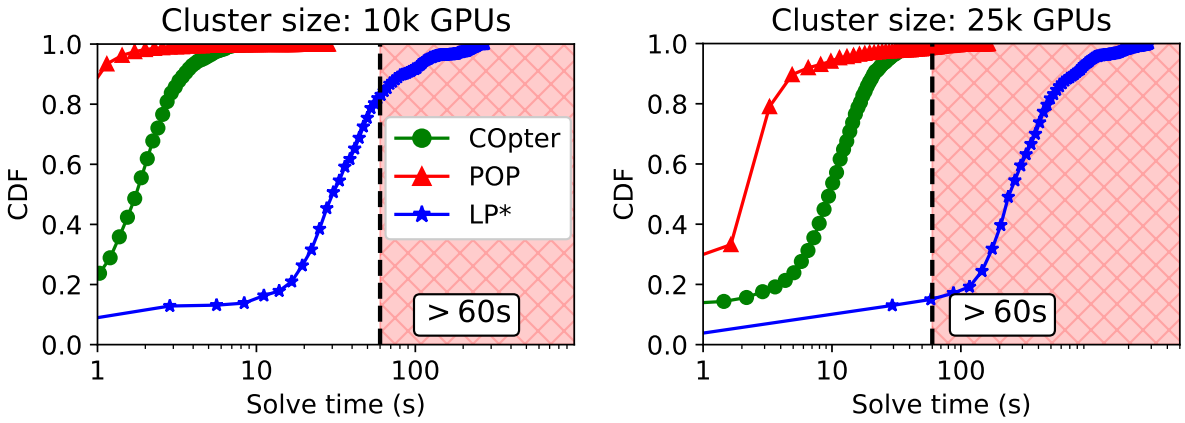


Figure 4.5: CDFs of end-to-end solver runtimes for Sia policy [42] in two clusters with 10k and 25k GPUs respectively. We compare `COpter` to (1) POP, which partitions the ILP 16-way and solves the sub-problems in parallel, and (2) *LP**, which solves the LP-Relaxation with a commercial solver [39] and recovers binary solutions using a shim.

subproblem, we see a large spread of runtimes – over a 30x difference between the 90th and 100th percentile solver times. `COpter`, on the other hand, shows predictable runtimes – the slowest runtime (70.4s) corresponds to the problem with the most number of variables (over 11-million variables) with just a 2-3x difference between the 90th and 100th percentile solver times.

4.5.2 Attribution of benefits

`COpter` provides a varying amount of benefits in our benchmarks. Differential program updates help if the set of resources and requests change, so this only benefits the GPU cluster scheduling problem. While all three problems benefit from our factorization-free, sparsity-aware efficiently warm-started solver, lightweight integerization via problem-specific *shims* only benefits GPU cluster scheduling and shard load balancing problems.

Figure 4.6 illustrates the stacking benefits from `COpter` on the LP-relaxation of the Sia ILP [42] in the previously described 10k GPU cluster. We choose the LP relaxation, and not the ILP directly, as many problems are directly formulated as LPs (like WAN traffic engineering) and do not benefit from using lightweight *shims* to resolve integer infeasibilities.

Figure 4.6 shows the impact of (a) using the custom solver implementation that benefits from problem and solution sparsity, (b) using a differential interface to reflect problem updates efficiently, and (c) efficiently warm-starting the custom solver using previous round's solution as the initial guess for the current round. On average, each of these contribute a stacking $5\times$, $3.5\times$, and $1.5\times$ speedup, respectively, for a total speedup of $24\times$ over independently solving problems using existing LP solvers. Additionally, as we will see later in Section 5.2.1, solving MILPs by first solving its LP-relaxation and then applying a problem-specific shim can lead to an end-to-end speedup of $2-3\times$ (using the same solver [39] for both MILP and LP-relaxation of the load balancing problem). Thus, we can expect `COpter` to provide $24-72\times$ speedup for MILPs.

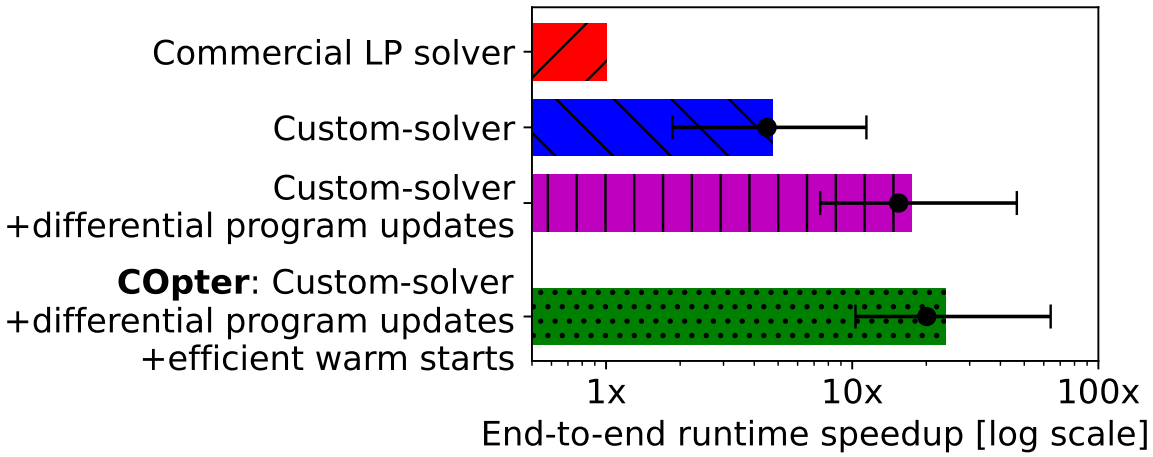


Figure 4.6: Speedups in end-to-end solver runtimes from two of the three techniques that make up `COpter` compared to a commercial LP solver (CPLEX [39]). We show mean, median, 10th and 90th percentile speedups for the LP-relaxation of the Sia ILP [42] for 1 minute rounds on a 10k GPU cluster.

4.6 Related Work and Discussion

In this section, we discuss some systems that formulate scheduling problems as optimization programs, describe existing approaches to solving these optimization programs, look at the landscape of methods used to reduce solver times in large-scale resource-allocation and contrast our approach to methods in existing literature. We also discuss some best and worst-case scenarios for `COpter`.

4.6.1 Scheduling as Optimization Problems

Plan-ahead schedulers like Tetri-Sched [96] and Shockwave [111] solve one MILP to determine allocations for multiple rounds, and as a result, are reasonably robust to missing scheduler deadlines in large clusters at the cost of increased queue times and reduced resource utilization. Meta's RAS [70] solves MILPs hourly for large-scale capacity reservation uses a two-stage decomposition (MILP for server-reservation assignment, then parallel container allocation), but requires problem-specific optimizations and heuristics to scale the MILP stage to large scales. Tetris [13] solves MILPs for load balancing ML workloads and data placement optimization across regions, and employs multi-level scheduling for scalability at the cost of global optimality. Gavel [68] supports time-slicing DNN jobs across heterogeneous GPUs using LP formulations for many scheduling objectives, but runs into solver bottlenecks if jobs request space-sharing leading to missed scheduler deadlines in large clusters. Rebalancer[54] is a framework to express and compile scheduling policies into MILPs. The MILPs are then solved using off-the-shelf solvers at small scales, and use a faster, sub-optimal local search algorithm on a graph representation at larger scales. `COpter` complements Rebalancer in two ways – (a) potentially extending the size of problems directly solvable as MILPs, and (b) using `COpter`'s outputs as initial guesses to initialize the local-search algorithm at scale for better quality solutions.

4.6.2 Solving Optimization Problems

Offline optimization. Offline optimization deals with problems that are fully specified before optimization starts. State-of-the-art LP and MILP solvers (e.g., CPLEX [39], Gurobi [29], GLPK [60]) often use simplex or interior-point (barrier) methods that rely on solving linear systems iteratively, and employ optimizations like matrix factorizations (LU, QR, LDL) [60, 39]. These methods scale poorly to large problems issues due to high per-iteration costs. Additionally, factorization often destroys sparsity patterns in the constraint matrices and must be recomputed if the problem changes, rendering them inefficient for continual optimization where problems change frequently. First-order methods (OSQP [91], ADMM [9], ALCD [104, 98], PGD [10], PDLP [4]) address these scalability challenges by trading slower convergence for lower per-iteration costs, and have proven useful in large-scale applications [18, 104, 98]. Stochastic variants like SGD reduce per-iteration costs further, sacrificing convergence guarantees and quality of optimum.

Online optimization. Online optimization[30] deals with problems that are revealed over time, often minimizing *regret* against a fixed optimum (*static* regret)[30, 82]. Static regret is less suitable for resource allocation problems whose optimal solution changes with time. Instead, *dynamic regret*, that compares against a sequence of changing optima [109, 108, 8], is more relevant. However, existing work does not focus on runtime efficiency that is important for slowly evolving resource-allocation problems. Our focus in continual optimization is different: we seek algorithms that provide both low dynamic regret and runtimes that scale favorably with the amount of change in the optimal solution over time, especially for slowly evolving resource-allocation problems that are the focus of this paper.

4.6.3 When to use `COpter`?

`COpter` is best applied to critical, large-scale resource allocation problems where the set of resources and requests change slowly over time. Many such problems have received extensive attention in recent literature, like:

- Cluster scheduling for DL workloads (e.g., Pollux [79], Sia [42], Shockwave [111]) and best-effort batch scheduling (e.g., Google Borg [95])
- Shard placement for elastic databases (e.g., Accordion [85], E-Store [92], see Section 5.1.1)
- Virtual Machine/Container/Data Placement (e.g., Meta's RAS [70] and MAST [13])
- Traffic Engineering (e.g., Google's B4 [41], Microsoft's SWAN [33]) (see Section 5.1.2)

`COpter` is not a great fit when resource needs and/or availability change dramatically from round-to-round, rather than in relatively small amounts over time, such as serverless functions or fine-grained streaming tasks.

4.7 Conclusion

We find that large-scale scheduling problems are better formulated as sequence of interconnected problems. `COpter` is our approach towards efficient continual optimization of slowly evolving LPs. It pairs a differential problem update interface with a factorization-free LP solver to seamlessly reuse computational effort across rounds, and uses lightweight problem-specific heuristics to recover high-quality feasible solutions to MILPs. In our evaluations, we find that applying `COpter` to the Sia scheduling policy finds allocations $10 - 100\times$ faster than state-of-the-art solvers with negligible loss in allocation quality. Further, `COpter` also outperforms POP in allocation quality with $1.5\times$ reduction in tail solver times, for a more predictable policy optimization time.

Chapter 5

Continual optimization for other resource-allocation problems

It is common to see resource-allocation problems from other domains formulated as linear/mixed-integer linear programs. In Wide Area Newtork (WAN) traffic engineering, operators allocate flows to links to maximize network utilization and minimize latency [102]. In elastic database systems, the query load to a fixed number of shards must be periodically re-balanced given a fixed set of servers [85]. We study these two problems thoroughly in this chapter, apply continual optimization and compare our approach using `COpter` against both heuristic and problem-partitioning approaches often used to scale these problems to large problem instances.

5.1 Problem descriptions

Similar to Sia, we describe the shard-load-balancing and traffic engineering problems in detail, and our approach to continual optimization of these problems.

5.1.1 Shard Load-Balancing

The shard load-balancing problem frequently arises in the operation of elastic database systems [17, 85, 92] where replicas of data shards are shuffled around a fixed pool of servers to ensure a consistent load across the server fleet.

Problem Formulation. Let $L = \{l_1, l_2, \ldots, l_D\}$ be the load on each of the D shards, and let N be the number of servers in the fleet. Let T denote the existing shard-server assignment where $T_{ij} = 1$ iff a replica of shard j exists on server i, $S = \{s_1, s_2, \ldots, s_D\}$ denote the size of each shard, and $M = \{m_1, m_2, \ldots m_N\}$ denote the memory capacities of each of the N servers. The optimization problem uses two related variables for each (server, shard) pair (i, j): R_{ij} and X_{ij}. $R_{ij} \geq 0$ denotes the fraction of load for shard j routed to server i, and $X_{i,j} \in \{0, 1\}$ denotes if a server i hosts a replica of shard j. So, $R_{ij} > 0 \implies X_{ij} = 1$, and

$R_{ij} = 0 \iff X_{ij} = 0$. The MILP is defined as follows:

$$\min_{R,X} \sum_i \sum_j (1 - T_{ij}) \cdot s_i \cdot X_{ij} \qquad (a)$$

$$\textbf{subject to:} \quad \sum_{i=1}^{N} R_{ij} = 1 \quad \forall\, j \qquad (b)$$

$$\mathcal{L} - \epsilon \leq \sum_j X_{ij} \cdot l_j \leq \mathcal{L} + \epsilon \quad \forall\, i \qquad (c)$$

$$\sum_{j=1}^{D} X_{ij} \cdot s_j \leq m_i \quad \forall\, i \qquad (d)$$

$$R_{ij} \leq X_{ij}, \quad X_{ij} \in \{0, 1\} \quad \forall\, i, j \qquad (e)$$

where (a) is the objective that minimizes the cost of starting up new replicas on servers without a replica for each data shard, (b) ensures all load for each shard j is accounted for, (c) ensures the load on each server is within ϵ of the average load $\mathcal{L} = \frac{\sum_j l_i}{N}$, (d) ensures a server i's memory usage does not exceed its capacity m_i, and (e) ensures that if a server i executes queries for shard j, it hosts a replica of shard j, and that shard-server assignment variables X are binary. Note that (c) allows for 2ϵ load imbalance – the difference in load between the most and least loaded servers – by design.

Applying `COpter`. We first relax the binary constraints on X_{ij} to $0 \leq X_{ij} \leq 1$. Then we solve the relaxed LP using continual optimization and apply a shim to recover a feasible binary X^{bin} and continuous R. We start the shim with an initial shard placement X^{bin} computed using R which may exceed some servers' memory capacities. We pick a server i whose memory capacity is exceeded in X^{bin} and greedily remove assigned shards j one by one, starting with those placing the least load $R_{ij} * l_j$. A shard j is only removed if it has replicas on other servers, and if removed, we proportionally shift its load share R_{ij} from server i to all servers hosting its other replicas. We repeat until server i's memory capacity is not violated, and move on to the next overloaded server. The final server-shard assignment X^{bin} and load distribution R does not violate hard-constraints like server-memory capacity, but may violate the load imbalance soft-constraint.

We show in Section 5.2.1 that (X^{bin}, R) rarely violate the soft load balancing constraints, and as a bonus, we find that `COpter` solutions are sometimes *better-than-optimal* — an artifact of the problem formulation that allows for reducing the number of replicas for shards without any penalties. We discuss this observation in more detail in Section 5.2.1.

Note that some solvers, particularly the ADMM-based SCS and OSQP [72, 91], produce *degenerate* solutions to the LP-relaxation that suggest each shard's load to be evenly split across all N servers, making it an infeasible solution due to server memory capacity constraints. However, our solver and Simplex based solvers produce mostly binary solutions to the LP-relaxation natively (see Figure 4.4).

5.1.2 WAN Traffic Engineering

In WAN traffic engineering, demands (i.e. flows) are allocated link capacities in the network to maximize total flow through the network of links and routers(*max-flow* [2, 67]) or minimize the maximum link utilization (*MLU* [67, 102, 23]) for better network performance. As the demands fluctuate, TE systems must frequently re-optimize flow allocations (often at 5-minute intervals [2, 67, 102]) to maintain optimal network performance. We describe the LP formulation for the maximize total flow, or *max-flow* for short, objective below.

Problem Formulation. Let $D = \{d_1, \ldots d_k\}$ be the set of demands such that demand j for source s_j and destination t_j requests bandwidth d_j. Let $P = \{P_1, \ldots P_k\}$ be a set of N pre-configured paths for each pair of routers in the network: $P_j = \{p_j^1, \ldots p_j^N\}$, and c_e be the capacity of the link connecting routers u and v for each edge $e = (u, v) \in E$ in the graph $G = (V, E)$ with routers as vertices V and links as edges E.

The *path formulation LP*[33, 34, 41], is defined as follows:

$$\max_X \sum_j \sum_{p \in P_j} X_j^p \qquad (a)$$

$$\textbf{subject to:} \quad \sum_{p \in P_j} X_j^p \leq d_j \quad \forall j \qquad (b)$$

$$\sum_{j, p \in P_j | e \in p} X_j^p \leq c_e \quad \forall\, e \in E \qquad (c)$$

$$X_j^p \geq 0 \quad \forall\, p \in P_j, \forall\, j \qquad (d)$$

where (a) is the objective maximizing the total allocated flow through the network, (b) ensures each demand j is allocated at-most the requested bandwidth d_j split across its N pre-configured paths, (c) ensures that the sum of all flows through a link e is limited by the link capacity c_e, and (d) ensures that all flow allocations are non-negative.

Applying `COpter`. The variables X_j^p are continuous as they represent flow values in Mb/s, so `COpter` will solve the above LP as-is using continual optimization without the use of shims. We show in Section 5.2.2 that `COpter`'s flow allocations result in optimal *max-flow* and for changes to demands across time, new and optimal allocations are quickly computed from previous allocations.

5.2 Experiments

This section evaluates the effectiveness of applying *continual optimization* via `COpter` to the resource-allocation problems previously described. We setup the experiments for each problem as follows:

- **Shard Load Balancing**. We follow the experiment setup described in [67] and simulate time-varying load distributions. We re-use the code open-sourced by authors of POP [67] wherever possible and use the same load distributions to benchmark `COpter` (implemented separately).
- **Traffic Engineering**. We optimize the *max flow* objective for two network topologies – the Kentucky Data Link network (`Kdl` [67, 102]), and an AS-level topology (`ASN` [102]). We implement `COpter` in the same evaluation framework as POP [67] to simulate identical traffic patterns.

Testbed. We use the same testbed previously used to benchmark `COpter`- Sia. For POP [67], we follow the approach used in prior work [67, 102], and solve each subproblem on a single thread and report estimated parallel runtimes computed mathematically for 64 cores.

5.2.1 Load Balancing

Evaluated approaches. We compare `COpter` against three approaches: (a) *Heuristic* is a greedy algorithm from E-Store [92], (b) *Exact* is the MILP formulation described

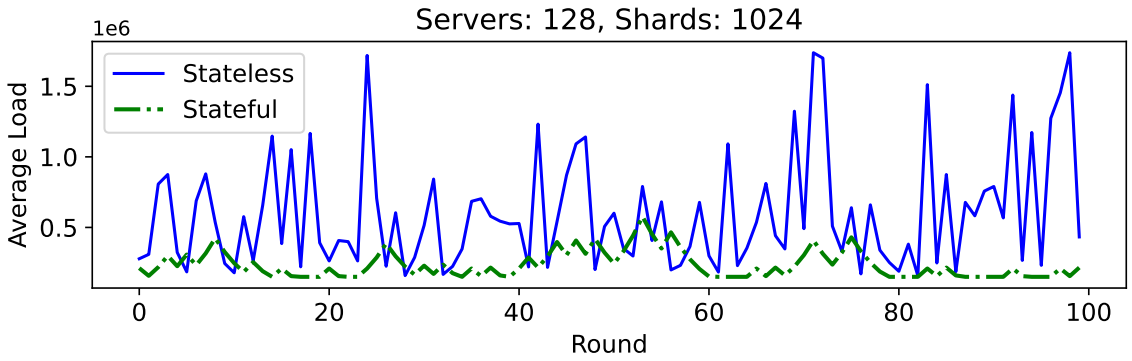


Figure 5.1: Average load per server for 1024 shards load-balanced across 128 servers.

in Section 5.1.1 and solved using a commercial MILP solver[39], (c) *LP-Relaxed* solves the LP-relaxation of the MILP formulation, and applies the *shim* described in Section 5.1.1 to resolve integer infeasibilities, and (d) *POP-k* splits the set of servers and shards into k pieces and solves the MILP formulation using a commercial MILP solver[39].

Metrics. We are interested in the three metrics: (a) *average shard movements* is the average number of shards movements per round in response to changing load distributions (lower is better), (b) *average load imbalance* is the difference in load between the most and least loaded servers (lower is better), and (c) *average load balancer time* is the time taken to compute shard-server assignments in each round and includes the time spent in the *shim* for approaches that use it (lower is better).

Load distributions. We benchmark two load distributions computed using one Zipf value z_k for each round k – (a) *Stateless* where z_k is uniformly random in $[z_{min}, z_{max}]$ and models an unpredictable and random query workload [67], and (b) *Stateful* where the $z_k = z_{k-1} \times (1 \pm 0.1)$ with uniformly random direction (i.e. 10% increase/decrease in z_{k-1}) and models smoother changes to load distributions across rounds. Figure 5.1 shows the average load per server for the two load distributions over time.

Additional details. We set $\epsilon = 5\%$ (upto 10% load imbalance allowed), run for 100 rounds and exclude the first 20 to remove any startup effects. We evaluate two problem sizes – 1024 shards, 128 servers and 2048 shards, 256 servers.

1024 Shards, 128 servers. Figure 5.2 (top) and (bottom) shows a summary of the results for the two load distributions. *Heuristic* is quick, produces many shard movements and falls short of maintaining load imbalance under the required 10%. *POP-8* using 8 subproblems runs in about a second and reduces shard movements, but also fails to control load imbalance – while all servers within a subproblem maintain low load imbalance, random partitioning of shard loads leads to load imbalance across subproblems, creating imbalance between pairs of servers across subproblems. *Exact* achieves perfect load imbalance with almost minimum shard-movements by solving the MILP to optimality. *LP-Relaxed* uses the same solver as *Exact*, but skips the integerization stage and uses a shim to speedup end-to-end solver times by $2-3\times$ without any adverse impacts to either shard movements or load imbalance. Finally, `COpter` employing continual optimization exploits the stability of server-shard assignments resulting in the lowest shard movements and load imbalances.

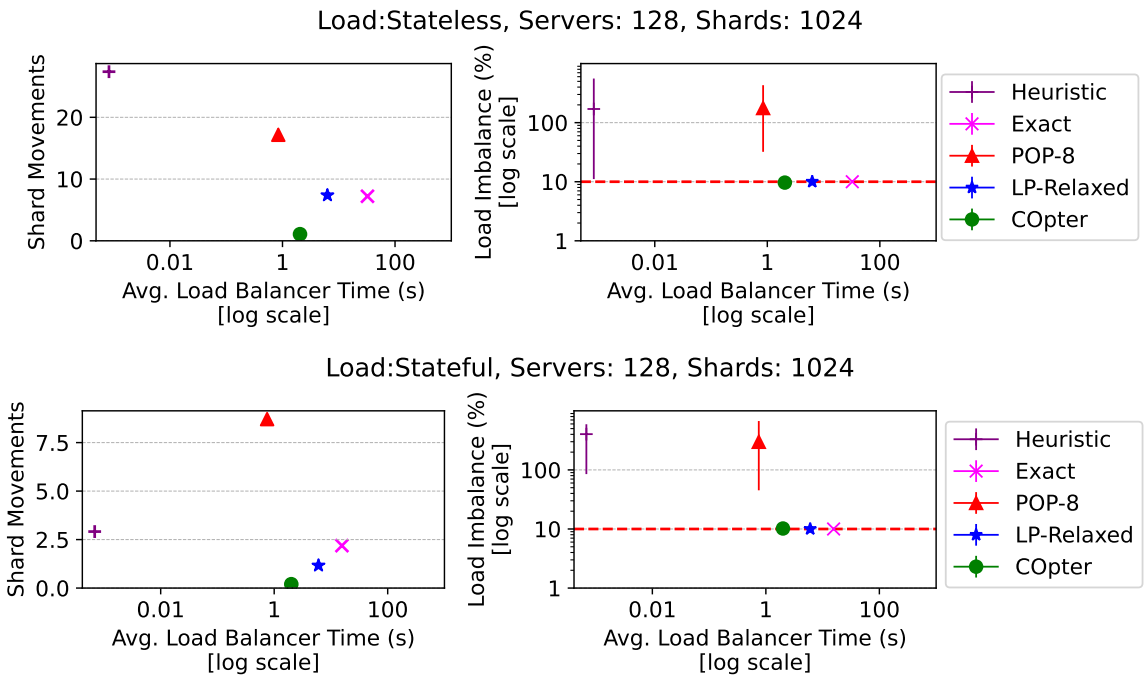


Figure 5.2: Summary of experiments with 1024 shards across 128 servers for *Stateless* (top) and *Stateful* (bottom) loads.

However, since neither the set of shards nor the servers change over time, `COpter` does not benefit from differential updates to the problem as only the cost and constraint vectors are updated in each round.

Super-optimality. Curiously, *LP-Relaxed* and `COpter` both achieve lower shard movements compared to *Exact*. Upon closer inspection, this *super-optimality* results from a mismatch between the dynamics of the load variance across rounds and the corresponding MILP formulation described in Section 5.1.1. Starting up a replica for a shard j on server i has a non-zero objective cost, whereas deleting a replica has zero cost. Consider a shard whose load increases between rounds $k-1$ and k, incurring corresponding shard movements in round k. Let the shard load decrease in the next round $k+1$ – *Exact* will find a solution that deletes a few replicas as they are no longer needed to maintain low load imbalance. If the load increases in round $k+1$, *Exact* will start up replicas again and incur shard movements. *LP-Relaxed* and `COpter` do not suffer as much from this oscillation for two reasons – (a) their solutions are not fully integral and using the *shim* to realize a binary solution leads to temporally stable solutions, and (b) `COpter` maintains solution sparsity that implicitly penalizes deletion of shards. `COpter`'s approach of continual optimization more closely matches the problem dynamics and delivers better quality solutions at a fraction of the runtime cost compared to *Exact* running with a commercial MILP solver.

2048 shards, 256 servers. Figure 5.3 (top) and (bottom) summarize our experiments with the *Stateless* and *Stateful* loads for 2048 shards hosted on 256 servers. We omit *Exact* from comparisons as each problem takes multiple hours to solve and is impractical for meaningful round durations. Similar to the 1024 shard case, *POP-16* is quick, but suffers

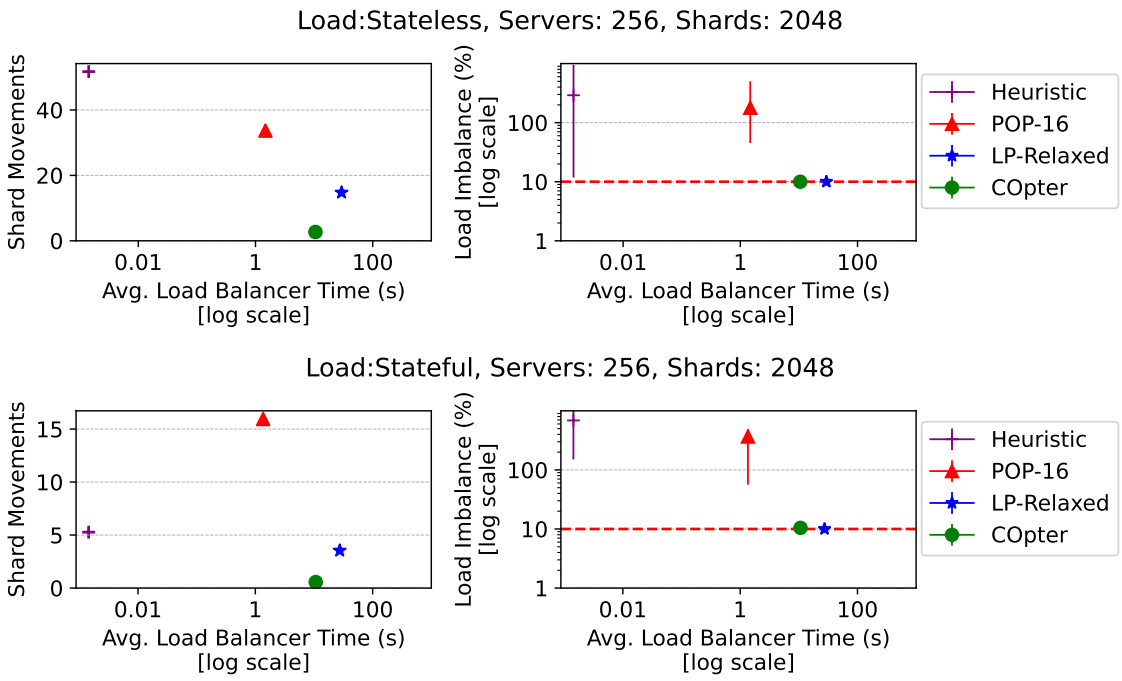


Figure 5.3: Summary of experiments with 2048 shards across 256 servers for *Stateless* (top) and *Stateful* (bottom) loads.

from load imbalance across subproblems. *LP-Relaxed* and `COpter` both meet load imbalance constraints *despite* using shims to resolve integer infeasibilities, further evidence that we can bypass the combinatorial explosion in large-scale resource allocation problems from integerization stage for MILPs.

`COpter` obtains high-quality solutions about 2.8× faster than *LP-Relaxed* using our factorization-free, sparsity-aware, and efficiently warm-started LP solver that benefits from highly stable shard-server assignments.

5.2.2 Traffic Engineering

Evaluated approaches. We compare `COpter` against three approaches: (a) *LP-All* solves the full LP on 8 threads using the Gurobi Commercial LP Solver (v11.0.3[29]), (b) *LP-Top* solves the LP using Gurobi[29] only for top 10% of demands (by volume), and allocates remaining demands to shortest paths (also called *demand pinning*[65]), and (c) *POP-k*[67] splits the LP into k subproblems. Each subproblem in POP replicates the full topology, but with $\frac{1}{k}$ of its link capacities, solves the LP for a random $\frac{1}{k}$ of the demands, and uses 4-way *client-splitting*[67] for large demands to improve allocation quality. Subproblems are solved in a single-thread and a *parallel* runtime is estimated mathematically.

Traffic matrices. We generate synthetic traffic matrices following the `Poisson` and `Bimodal` distributions (similar to prior work[2, 67]). Figure 5.4 shows the probability distribution of demands used for the `Kdl` topology. `Poisson` is dominated by few flows requesting majority of the demand by volume, whereas `Bimodal` sees roughly even split

between two *low* and *high* demand bands. In each round, we perturb demands from the previous round by upto 10% to simulate time-varying demands. We run each experiment for 20 rounds and exclude the first 5 to remove startup effects.

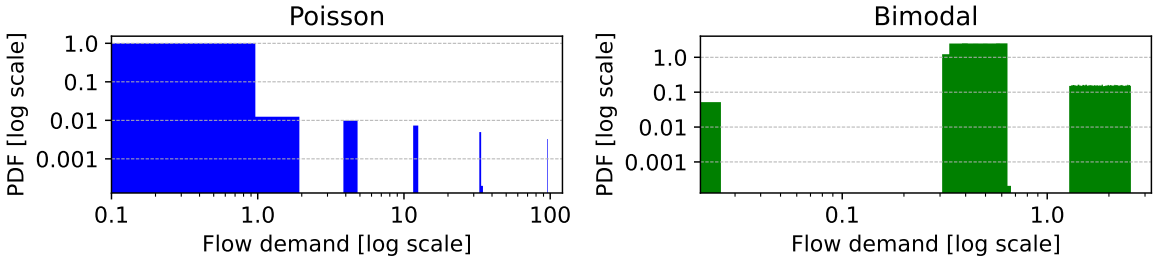


Figure 5.4: Distribution of demands in the two synthetic traffic matrices – in *Poisson*, few large flows dominate the demands, and in *Bimodal* demand is split across two distinct bands.

Metrics. We consider three metrics: (a) *Total allocated flow* is the the sum of allocated flow across all demands, averaged across rounds (higher is better), (b) *Runtime* is the average time taken to compute flow allocations across rounds (lower is better), and (c) *Optimality Gap* is the difference between the total allocated flow for a given approach and the optimal value found by *LP-All* (lower is better). We compute optimality gap against `COpter` for the `ASN` topology as *LP-All* takes tens of hours to solve for each round and `COpter` allocates the most flow among the evaluated approaches.

Figure 5.5 and Figure 5.6 summarize our experiments for `Poisson` and `Bimodal` traffic matrices, respectively. For the `Bimodal` traffic matrices, *POP-128* takes multiple days to solve all subproblems for a single round, so we present results only for the first round. The first round runtime is representative of POP-128's average runtime as it does not benefit from any warm-starts because it creates new subproblems from random partitions in each round.

Poisson demands. *Poisson* is dominated by a small set of demands, so as expected, *LP-Top* is two orders of magnitude quicker and allocates as much flow as *LP-All* for `Kdl` topology, but only reaches 90% of `COpter`'s allocation for `ASN` showcasing the limitation of heuristics. *POP-64* for `Kdl` and *POP-128* for `ASN` allocate $> 99\%$ of optimal flow with runtimes well under the 5-minute round durations. `COpter` allocates $> 99.9\%$ of optimal and the highest flows for `Kdl` and `ASN` topologies, respectively well under a minute.

Bimodal demands. *LP-Top* fails to allocate $> 90\%$ of optimal flow, but is still orders of magnitude quicker than other approaches. *LP-All* exceeds the 5-minute round duration for `Kdl` topology, and is omitted from comparisons for `ASN` as its runtime exceeds 24-hours for each round. *POP* compares favorably for the `Kdl` topology, but takes over 20 minutes to solve each problem for the `ASN` topology, rendering 5-minute durations impractical. `COpter`'s approach using continual optimization runs in less than a minute and allocates the highest total flow for the `ASN` topology — a $30\times$ speedup while allocating 1.5% more flow.

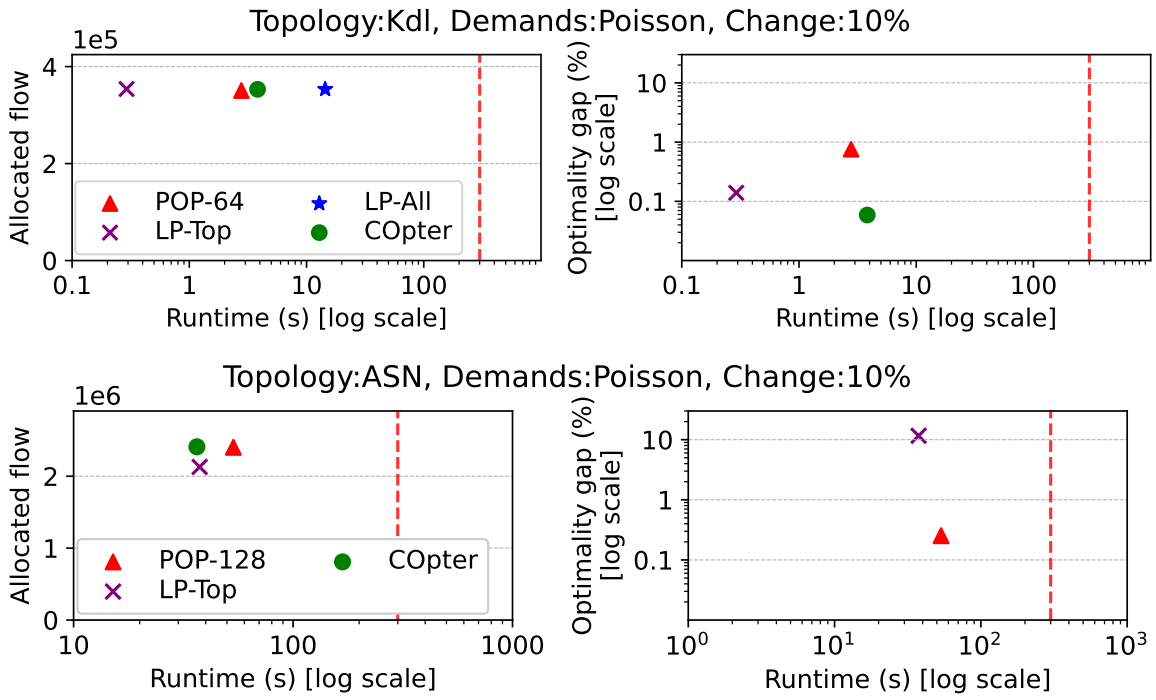


Figure 5.5: Summary of experiments with *Poisson* demands for `Kdl` (upper) and `ASN` (lower) topologies. Dashed red-line indicates the commonly used 5-minute round duration in WAN traffic engineering [102, 34].

5.3 Related Work and Discussion

In this section, we discuss some approaches to scaling traffic engineering that avoid solving large optimization problems (like `COpter`).

Traffic Engineering. NCFlow [2] uses a decomposition explicitly designed for the *max-flow* objective, similar in spirit to POP [67]. Teal [102] uses a learning-based approach that uses historical data for fast online allocation using reinforcement learning coupled with ADMM [9] to obtain feasible flow allocations in seconds. DOTE [77] bypasses formulating TE objectives as LPs and directly computes optimal flow splitting using deep neural networks trained on historical traffic demand data. Teal and DOTE are examples of learning-based approaches for the TE problem. RedTE [28] proposes using distributed decision making with routers computing allocations locally using multi-agent deep reinforcement learning for quicker response to bursts in demands. MegaTE [62] re-architects the TE control loop for *bottom-up* control that allows for asynchronous updates and finer-granularity flow allocations at container level. RedTE and MegaTE are examples that bypass centralized TE optimization.

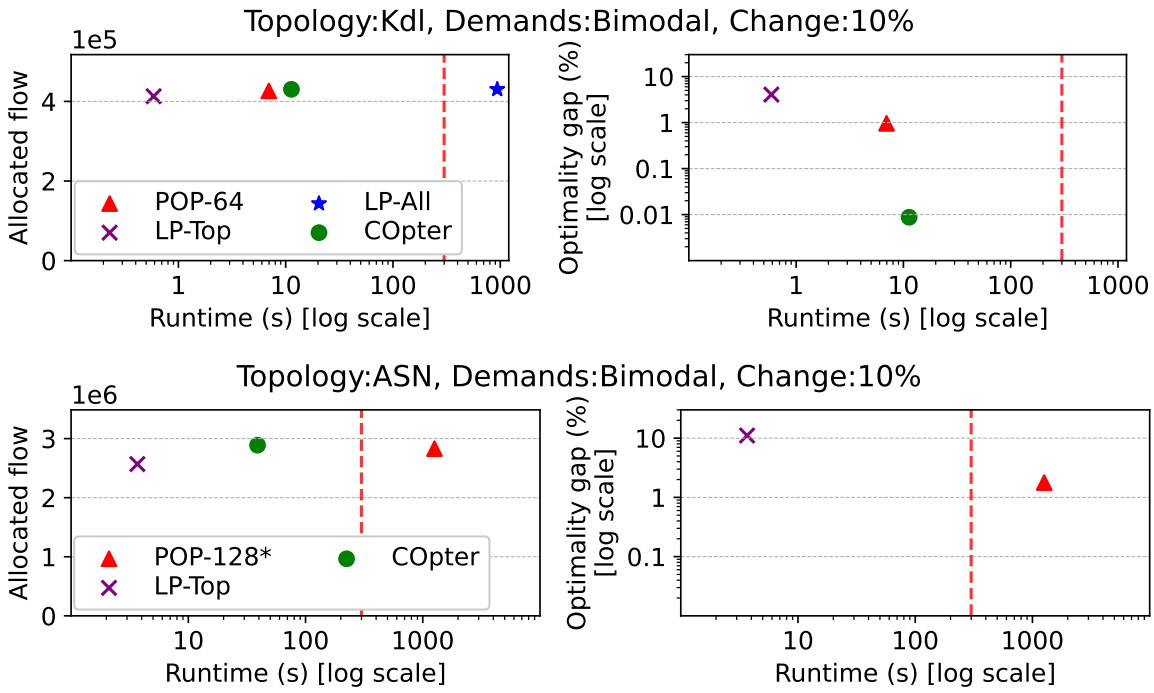


Figure 5.6: Summary of experiments with *Bimodal* demands for `Kdl` (upper) and `ASN` (lower) topologies.

5.4 Conclusion

Large-scale resource-allocation problems benefit from being formulated as a sequence of interconnected problems. Continual optimization as implemented in `COpter` exploits the structure and slowly-evolving nature of the problems and their optimal solutions efficiently by re-use computational effort from prior rounds to reduce solver runtimes in a given round. On the shard-load balancing and WAN traffic engineering resource-allocation problems, `COpter` finds allocations $10-100\times$ faster than state-of-the-art solvers with negligible loss in allocation quality, and outperforms POP in allocation quality with $1.5-30\times$ speedups and predictable solver runtimes.

Chapter 6

Conclusion

This chapter summarizes the contributions of this dissertation and outlines a few limitations and potential directions for future work.

6.1 Summary of contributions

This dissertation proposes and implements a heterogeneity and adaptivity-aware GPU cluster scheduler that continuously optimizes many dimensions of adaptivity for adaptive DNN training jobs in heterogeneous GPU clusters with tens of thousands of GPUs. Compared to state-of-the-art schedulers that are either heterogeneity-aware or adaptivity-aware but not both, our proposed scheduler reduces average job completion times by 30-93% while using 12-60% fewer GPU hours with minimal scheduler overheads. Additionally, continual optimization successfully scales our scheduler to clusters with tens of thousands of GPUs for one-minute round durations.

Sia is our heterogeneity and adaptivity-aware GPU cluster scheduler. Sia uses the notion of *configurations* to structure the ever-increasing state-space of adaptivity and allocation choices and introduces *bootstrapped throughput models* efficiently estimate jobs' goodputs on the reduced state-space of configurations with minimal profiling overheads. Sia scheduler policy maximizes cluster-wide training progress (aka goodput [79]) by assigning at-most one configuration for each job given limited GPU resources. Formulated as a mixed-integer linear program (MILP), the scheduler policy re-optimizes job allocations once a minute in response to job arrivals, completions and phase changes and scales to clusters with a few thousand GPUs and many GPU types.

`COpter` is our approach to continual optimization. Continual optimization is a new paradigm that explicitly models the slow-evolution in round-based resource-allocation problems between successive rounds, and aims to exploit the *slowly evolving* nature of the optimal allocations to reduce time-to-allocation in each round. `COpter` provides efficient continual optimization at scale by eliminating bottlenecks that prevent re-using of computational effort from prior rounds. First, it uses a differential problem update interface to efficiently manipulate their mathematical representations for small changes in resources and/or requests. Second, it employs a factorization-free LP solver that provably benefits from the slowly-evolving nature of the optimal solutions to problems in successive rounds. Third, it

employs problem-specific lightweight heuristics to recover feasible integral solutions with negligible quality loss. Compared to problem partitioning approaches like POP, we find that `COpter` also improves both allocation quality and solver runtime for many large-scale resource-allocation problems.

Our results demonstrate that Sia and `COpter` taken together provide a responsive and efficient scheduler for large GPU clusters with tens of thousands of GPUs and many GPU types, and capable of adapting multiple adaptivity dimensions with low scheduler overheads, fulfilling the goals of this dissertation.

6.2 Limitations and Future Work

This section highlights some limitations of our work as described in this dissertation and suggests potential directions for future work.

Mix predictive modeling and online profiling. The Sia scheduler profiles each new job on the smallest possible configuration for each GPU type. As the diversity in GPU types increases, this profiling step might consume non-trivial resources in large clusters. It may be possible to leverage the computation graph representation of DNN training jobs to predict iteration times and throughputs, and bypass the profiling stage entirely. These predictions need not be perfect, as evidenced by the efficacy of Sia's bootstrapped models in determining the right adaptivity choices with minimal overheads. Recent work in this space shows that historical data and sufficient offline profiling can help predict iteration times and memory usage for many models with little error[25, 57, 26]. Future work could look at mixing *predictive* modeling with the online profiling techniques employed by Sia to achieve low modeling error with zero initial profiling overheads.

Support diverse scheduling policies and improve solver integration. The Sia scheduler policy is formulated as linear program with integer constraints whose linear relaxation can be solved directly using the factorization-free solver implemented in `COpter`. Since the solver relies on forming an Augmented-Lagrangian for the input programs, it can also support *any* convex objective function. Future work could provide a generalized solver for resource-allocation problems with support for convex objectives to accommodate more diverse scheduling policies (e.g., Shockwave's objective function uses a `log` objective function [111]). The solver can also be integrated into a framework similar to `Rebalancer` [54] for ease of programming and reduced friction of adoption.

Leverage historical data for *predictive* scheduling. Schedulers are long-lived components of GPU clusters and future work could look at integrating predictions using historical data to improve Sia and other GPU cluster schedulers. We describe a few directions below:

- ***Diurnal demands and availabilities***: GPU demands show diurnal patterns (i.e., demands are different during the days compared to nights) [35, 99], so jobs can be scaled up/down in anticipation of reduced/increased demands. Anticipatory scaling can reduce the instantaneous load on shared networking and distributed storage by distributing the load over time, reducing the pressure on these systems significantly.

- ***Geographic variations in demands and availabilities***: GPU demands are not equally distributed across the globe. Planet-scale scheduling [13, 88] can leverage the

inequal GPU demand and availability to reduce job completion times and improve resource utilization. To move training jobs between from one geographic region to another, the scheduler must first copy the code, data and model checkpoints to the destination before it can resume training. Copying this context can be time consuming (may take tens of minutes), but can also alleviate GPU pressure in the source geographic region. One can leverage historical data to predict GPU demand and availabilities, and incorporate the transfer costs into the scheduler's objective function for anticipatory load shifting across geographic regions to reduce job completion times and improve resource utilization across the global fleet of GPUs.

- ***Spot availabilities***: Many cloud providers offer low-cost *spot* virtual machines (VMs) that offer the same performance as traditional VMs, but can be de-allocated at a moments notice to accommodate a traditional VM reservation at a higher cost. Spot instances can reduce cost of training (often a fraction of the cost of traditional reserved VMs), but carry the risk of increased runtimes from frequent VM allocation and de-allocation. A cost-sensitive scheduler can leverage the historical data on spot VM availability and eviction rates to seamlessly (and transparently) shift jobs from soon-to-be evicted spot resources to recently acquired spot resources. This would allow for low-cost training without the risk of unexpected preemption.

- ***Déjà vu***: DL practitioners often train the same DNN model with different *hyper-parameters* (e.g. learning rates, batch sizes, etc) in an attempt to discover the best performing parameters. Repeated training of models with the same structure can benefit from historical profiling data from prior executions by directly starting execution with optimal resources, reducing scheduler overhead and improving GPU efficiency. Additionally, with the rise of foundational models, DNN workloads using the same foundational model can also benefit from each other's profiling data in the same manner. Future work could look at leveraging model structures to reduce profiling overheads by re-using profiling data from prior execution and profiling of matching model structures.

Bibliography

[1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. "TensorFlow: A System for Large-Scale Machine Learning". In: *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*. OSDI'16. USA: USENIX Association, 2016.

[2] Firas Abuzaid, Srikanth Kandula, Behnaz Arzani, Ishai Menache, Matei Zaharia, and Peter Bailis. "Contracting wide-area network topologies to solve flow problems quickly". In: *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*. 2021, pp. 175–200.

[3] Dario Amodei, Sundaram Ananthanarayanan, Rishita Anubhai, Jingliang Bai, Eric Battenberg, Carl Case, Jared Casper, Bryan Catanzaro, Qiang Cheng, Guoliang Chen, et al. "Deep speech 2: End-to-end speech recognition in english and mandarin". In: *International conference on machine learning*. PMLR. 2016, pp. 173–182.

[4] David Applegate, Mateo Díaz, Oliver Hinder, Haihao Lu, Miles Lubin, Brendan O'Donoghue, and Warren Schudy. "Practical large-scale linear programming using primal-dual hybrid gradient". In: *Advances in Neural Information Processing Systems* 34 (2021), pp. 20243–20257.

[5] Mosek ApS. "Mosek optimizer API for python". In: *Version* 9.17 (2022), pp. 6–4.

[6] Sanjith Athlur, Nitika Saran, Muthian Sivathanu, Ramachandran Ramjee, and Nipun Kwatra. "Varuna: scalable, low-cost training of massive deep learning models". In: *Proceedings of the Seventeenth European Conference on Computer Systems*. 2022, pp. 472–487.

[7] *AWS Tranium*. https://aws.amazon.com/machine-learning/trainium/. 2022.

[8] Dheeraj Baby and Yu-Xiang Wang. "Optimal dynamic regret in proper online learning with strongly convex losses and beyond". In: *International Conference on Artificial Intelligence and Statistics*. PMLR. 2022, pp. 1805–1845.

[9] Stephen Boyd, Neal Parikh, Eric Chu, Borja Peleato, Jonathan Eckstein, et al. "Distributed optimization and statistical learning via the alternating direction method of multipliers". In: *Foundations and Trends® in Machine learning* 3.1 (2011), pp. 1–122.

[10] Paul H Calamai and Jorge J Moré. "Projected gradient methods for linearly constrained problems". In: *Mathematical programming* 39.1 (1987), pp. 93–116.

[11] Shubham Chaudhary, Ramachandran Ramjee, Muthian Sivathanu, Nipun Kwatra, and Srinidhi Viswanatha. "Balancing Efficiency and Fairness in Heterogeneous GPU Clusters for Deep Learning". In: *Proceedings of the Fifteenth European Conference on Computer Systems*. EuroSys '20.

[12] Tianqi Chen, Mu Li, Yutian Li, Min Lin, Naiyan Wang, Minjie Wang, Tianjun Xiao, Bing Xu, Chiyuan Zhang, and Zheng Zhang. "MXNet: A Flexible and Efficient Machine Learning Library for Heterogeneous Distributed Systems". In: *CoRR* (2015).

[13] Arnab Choudhury, Yang Wang, Tuomas Pelkonen, Kutta Srinivasan, Abha Jain, Shenghao Lin, Delia David, Siavash Soleimanifard, Michael Chen, Abhishek Yadav, et al. "{MAST}: Global scheduling of {ML} training across {Geo-Distributed} datacenters at hyperscale". In: *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*. 2024, pp. 563–580.

[14] A. R. Conn, N. I. M. Gould, and P. L. Toint. *Lancelot: A Fortran Package for Large-Scale Nonlinear Optimization (Release A)*. 1st. Springer Publishing Company, Incorporated, 2010. ISBN: 3642081398.

[15] IBM ILOG Cplex. "V12. 1: User's Manual for CPLEX". In: *International Business Machines Corporation* 46.53 (2009), p. 157.

[16] Henggang Cui, Hao Zhang, Gregory R. Ganger, Phillip B. Gibbons, and Eric P. Xing. "GeePS: Scalable Deep Learning on Distributed GPUs with a GPU-Specialized Parameter Server". In: *Proceedings of the Eleventh European Conference on Computer Systems*. EuroSys '16. Association for Computing Machinery, 2016.

[17] Carlo Curino, Evan PC Jones, Samuel Madden, and Hari Balakrishnan. "Workload-aware database monitoring and consolidation". In: *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data*. 2011, pp. 313–324.

[18] Frédéric Delbos and Jean Charles Gilbert. "Global linear convergence of an augmented Lagrangian algorithm for solving convex quadratic optimization problems". In: *Journal of Convex Analysis* 12.1 (2005), p. 25.

[19] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. "Imagenet: A large-scale hierarchical image database". In: *2009 IEEE conference on computer vision and pattern recognition*. Ieee. 2009, pp. 248–255.

[20] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. "BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding". In: *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*. Minneapolis, Minnesota: Association for Computational Linguistics, 2019.

[21] Steven Diamond and Stephen Boyd. "CVXPY: A Python-embedded modeling language for convex optimization". In: *The Journal of Machine Learning Research* 17.1 (2016), pp. 2909–2913.

[22] Mark Everingham, Luc Van Gool, Christopher KI Williams, John Winn, and Andrew Zisserman. "The pascal visual object classes (voc) challenge". In: *International journal of computer vision* 88.2 (2010), pp. 303–338.

[23] Lisa K Fleischer. "Approximating fractional multicommodity flow independent of the number of commodities". In: *SIAM Journal on Discrete Mathematics* 13.4 (2000), pp. 505–520.

[24] John Forrest, Stefan Vigerske, Ted Ralphs, John Forrest, Lou Hafer, jpfasano, Haroldo Gambini Santos, Jan-Willem, Matthew Saltzman, a-andre, Bjarni Kristjansson, h-i-gassmann, Alan King, Arevall, Bohdan Mart, Pierre Bonami, Ruan Luies, Samuel Brito, and to-st. *coin-or/Clp: Release releases/1.17.10*. Version releases/1.17.10. Aug. 2024. DOI: 10.5281/zenodo.13347196. URL: https://doi.org/10.5281/zenodo.13347196.

[25] Yanjie Gao, Xianyu Gu, Hongyu Zhang, Haoxiang Lin, and Mao Yang. "Runtime performance prediction for deep learning models with graph neural network". In: *2023 IEEE/ACM 45th International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*. IEEE. 2023, pp. 368–380.

[26] X Yu Geoffrey, Yubo Gao, Pavel Golikov, and Gennady Pekhimenko. "Habitat: A {Runtime-Based} computational performance predictor for deep neural network training". In: *2021 USENIX Annual Technical Conference (USENIX ATC 21)*. 2021, pp. 503–521.

[27] Juncheng Gu, Mosharaf Chowdhury, Kang G. Shin, Yibo Zhu, Myeongjae Jeon, Junjie Qian, Hongqiang Liu, and Chuanxiong Guo. "Tiresias: A GPU Cluster Manager for Distributed Deep Learning". In: *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*. Boston, MA: USENIX Association, Feb. 2019, pp. 485–500. ISBN: 978-1-931971-49-2. URL: https://www.usenix.org/conference/nsdi19/presentation/gu.

[28] Fei Gui, Songtao Wang, Dan Li, Li Chen, Kaihui Gao, Congcong Min, and Yi Wang. "RedTE: Mitigating subsecond traffic bursts with real-time and distributed traffic engineering". In: *Proceedings of the ACM SIGCOMM 2024 Conference*. 2024, pp. 71–85.

[29] Gurobi Optimization, LLC. *Gurobi Optimizer Reference Manual*. 2023. URL: https://www.gurobi.com.

[30] Elad Hazan et al. "Introduction to online convex optimization". In: *Foundations and Trends® in Optimization* 2.3-4 (2016), pp. 157–325.

[31] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. "Deep residual learning for image recognition". In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2016, pp. 770–778.

[32] Ben Hermans, Andreas Themelis, and Panagiotis Patrinos. "QPALM: A proximal augmented Lagrangian method for nonconvex quadratic programs". In: *Mathematical Programming Computation* 14.3 (2022), pp. 497–541.

[33] Chi-Yao Hong, Srikanth Kandula, Ratul Mahajan, Ming Zhang, Vijay Gill, Mohan Nanduri, and Roger Wattenhofer. "Achieving high utilization with software-driven WAN". In: *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM*. 2013, pp. 15–26.

[34] Chi-Yao Hong, Subhasree Mandal, Mohammad Al-Fares, Min Zhu, Richard Alimi, Chandan Bhagat, Sourabh Jain, Jay Kaimal, Shiyu Liang, Kirill Mendelev, et al. "B4 and after: managing hierarchy, partitioning, and asymmetry for availability and scale in google's software-defined WAN". In: *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*. 2018, pp. 74–87.

[35] Qinghao Hu, Peng Sun, Shengen Yan, Yonggang Wen, and Tianwei Zhang. "Characterization and prediction of deep learning workloads in large-scale gpu datacenters". In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 2021, pp. 1–15.

[36] Yanping Huang, Youlong Cheng, Ankur Bapna, Orhan Firat, Mia Xu Chen, Dehao Chen, HyoukJoong Lee, Jiquan Ngiam, Quoc V. Le, Yonghui Wu, and Zhifeng Chen. "GPipe: Efficient Training of Giant Neural Networks Using Pipeline Parallelism". In: *Proceedings of the 33rd International Conference on Neural Information Processing Systems*. 2019.

[37] Qi Huangfu and JA Julian Hall. "Parallelizing the dual revised simplex method". In: *Mathematical Programming Computation* 10.1 (2018), pp. 119–142.

[38] Changho Hwang, Taehyun Kim, Sunghyun Kim, Jinwoo Shin, and KyoungSoo Park. "Elastic resource sharing for distributed deep learning". In: *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*. 2021, pp. 721–739.

[39] IBM Corporation. *IBM ILOG CPLEX Optimization Studio V22.1.1: User's Manual*. IBM Corporation. 2022.

[40] Petuum Inc. *petuum/adaptdl: Resource-adaptive cluster scheduler for deep learning training*. Apr. 2021. URL: https://github.com/petuum/adaptdl/tree/osdi21-artifact.

[41] Sushant Jain, Alok Kumar, Subhasree Mandal, Joon Ong, Leon Poutievski, Arjun Singh, Subbaiah Venkata, Jim Wanderer, Junlan Zhou, Min Zhu, et al. "B4: Experience with a globally-deployed software defined WAN". In: *ACM SIGCOMM Computer Communication Review* 43.4 (2013), pp. 3–14.

[42] Suhas Jayaram Subramanya, Daiyaan Arfeen, Shouxu Lin, Aurick Qiao, Zhihao Jia, and Gregory R Ganger. "Sia: Heterogeneity-aware, goodput-optimized ML-cluster scheduling". In: *Proceedings of the 29th Symposium on Operating Systems Principles*. 2023, pp. 642–657.

[43] Myeongjae Jeon, Shivaram Venkataraman, Amar Phanishayee, Junjie Qian, Wencong Xiao, and Fan Yang. "Analysis of {Large-Scale}{Multi-Tenant}{GPU} Clusters for {DNN} Training Workloads". In: *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. 2019, pp. 947–960.

[44] Zhe Jia, Blake Tillman, Marco Maggioni, and Daniele Paolo Scarpazza. "Dissecting the graphcore ipu architecture via microbenchmarking". In: *arXiv preprint arXiv:1912.03413* (2019).

[45] Zhihao Jia, Matei Zaharia, and Alex Aiken. "Beyond Data and Model Parallelism for Deep Neural Networks." In: *Proceedings of Machine Learning and Systems* 1 (2019), pp. 1–13.

[46] Elizabeth John and E Alper Yıldırım. "Implementation of warm-start strategies in interior-point methods for linear programming in fixed dimension". In: *Computational Optimization and Applications* 41.2 (2008), pp. 151–183.

[47] Tyler Johnson, Pulkit Agrawal, Haijie Gu, and Carlos Guestrin. "AdaScale SGD: A User-Friendly Algorithm for Distributed Training". In: *Proceedings of the 37th International Conference on Machine Learning*. Ed. by Hal Daumé III and Aarti Singh. Vol. 119. Proceedings of Machine Learning Research. PMLR, July 2020, pp. 4911–4920. URL: https://proceedings.mlr.press/v119/johnson20a.html.

[48] Norman P Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, et al. "In-datacenter performance analysis of a tensor processing unit". In: *Proceedings of the 44th annual international symposium on computer architecture*. 2017, pp. 1–12.

[49] Nitish Shirish Keskar, Dheevatsa Mudigere, Jorge Nocedal, Mikhail Smelyanskiy, and Ping Tak Peter Tang. "On large-batch training for deep learning: Generalization gap and sharp minima". In: *arXiv preprint arXiv:1609.04836* (2016).

[50] Diederik P. Kingma and Jimmy Ba. "Adam: A Method for Stochastic Optimization". In: *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*. 2015.

[51] John Kominek and Alan W Black. "The CMU Arctic speech databases". In: *Fifth ISCA workshop on speech synthesis*. 2004.

[52] Alex Krizhevsky. "One weird trick for parallelizing convolutional neural networks". In: *arXiv preprint arXiv:1404.5997* (2014).

[53] Alex Krizhevsky, Geoffrey Hinton, et al. *Learning multiple layers of features from tiny images*. Tech. rep. 2009.

[54] Neeraj Kumar, Pol Mauri Ruiz, Vijay Menon, Igor Kabiljo, Mayank Pundir, Andrew Newell, Daniel Lee, Liyuan Wang, and Chunqiang Tang. "Optimizing resource allocation in hyperscale datacenters: Scalability, usability, and experiences". In: *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*. 2024, pp. 507–528.

[55] Gary Lauterbach. "The Path to Successful Wafer-Scale Integration: The Cerebras Story". In: *IEEE Micro* 41.6 (2021), pp. 52–57.

[56] Tan N. Le, Xiao Sun, Mosharaf Chowdhury, and Zhenhua Liu. "AlloX: Compute Allocation in Hybrid Clusters". In: EuroSys '20. Association for Computing Machinery, 2020.

[57] Seonho Lee, Amar Phanishayee, and Divya Mahajan. "Forecasting GPU Performance for Deep Learning Training and Inference". In: *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1*. ASPLOS '25. Rotterdam, Netherlands: Association for Computing Machinery, 2025, pp. 493–508. ISBN: 9798400706981. DOI: 10.1145/3669940.3707265. URL: https://doi.org/10.1145/3669940.3707265.

[58] Ilya Loshchilov and Frank Hutter. "Decoupled Weight Decay Regularization". In: *7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019*. 2019.

[59] Kshiteej Mahajan, Arjun Balasubramanian, Arjun Singhvi, Shivaram Venkataraman, Aditya Akella, Amar Phanishayee, and Shuchi Chawla. "Themis: Fair and efficient {GPU} cluster scheduling". In: *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*. 2020, pp. 289–304.

[60] Andrew Makhorin. "GLPK (GNU linear programming kit)". In: *http://www. gnu. org/s/glpk/glpk. html* (2022).

[61] Sam McCandlish, Jared Kaplan, and Dario Amodei. "An Empirical Model of Large-Batch Training". In: *arXiv preprint arXiv:1812.06162* (2018).

[62] Congcong Miao, Zhizhen Zhong, Yunming Xiao, Feng Yang, Senkuo Zhang, Yinan Jiang, Zizhuo Bai, Chaodong Lu, Jingyi Geng, Zekun He, et al. "MegaTE: Extending WAN Traffic Engineering to Millions of Endpoints in Virtualized Cloud". In: *Proceedings of the ACM SIGCOMM 2024 Conference*. 2024, pp. 103–116.

[63] Xupeng Miao, Yujie Wang, Youhe Jiang, Chunan Shi, Xiaonan Nie, Hailin Zhang, and Bin Cui. "Galvatron: Efficient Transformer Training over Multiple GPUs Using Automatic Parallelism". In: *Proc. VLDB Endow.* 16.3 (2022).

[64] Philipp Moritz, Robert Nishihara, Stephanie Wang, Alexey Tumanov, Richard Liaw, Eric Liang, Melih Elibol, Zongheng Yang, William Paul, Michael I Jordan, et al. "Ray: A distributed framework for emerging {AI} applications". In: *13th USENIX symposium on operating systems design and implementation (OSDI 18)*. 2018, pp. 561–577.

[65] Pooria Namyar, Behnaz Arzani, Ryan Beckett, Santiago Segarra, Himanshu Raj, and Srikanth Kandula. "Minding the gap between fast heuristics and their optimal counterparts". In: *Proceedings of the 21st ACM Workshop on Hot Topics in Networks*. 2022, pp. 138–144.

[66] Deepak Narayanan, Aaron Harlap, Amar Phanishayee, Vivek Seshadri, Nikhil R. Devanur, Gregory R. Ganger, Phillip B. Gibbons, and Matei Zaharia. "PipeDream: Generalized Pipeline Parallelism for DNN Training". In: *Proceedings of the 27th ACM Symposium on Operating Systems Principles*. SOSP '19. Association for Computing Machinery, 2019.

[67] Deepak Narayanan, Fiodar Kazhamiaka, Firas Abuzaid, Peter Kraft, Akshay Agrawal, Srikanth Kandula, Stephen Boyd, and Matei Zaharia. "Solving large-scale granular resource allocation problems efficiently with pop". In: *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*. 2021, pp. 521–537.

[68] Deepak Narayanan, Keshav Santhanam, Fiodar Kazhamiaka, Amar Phanishayee, and Matei Zaharia. "{Heterogeneity-Aware} Cluster Scheduling Policies for Deep Learning Workloads". In: *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. 2020, pp. 481–498.

[69] Deepak Narayanan, Mohammad Shoeybi, Jared Casper, Patrick LeGresley, Mostofa Patwary, Vijay Korthikanti, Dmitri Vainbrand, Prethvi Kashinkunti, Julie Bernauer, Bryan Catanzaro, Amar Phanishayee, and Matei Zaharia. "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*. SC '21. Association for Computing Machinery, 2021.

[70] Andrew Newell, Dimitrios Skarlatos, Jingyuan Fan, Pavan Kumar, Maxim Khutornenko, Mayank Pundir, Yirui Zhang, Mingjun Zhang, Yuanlai Liu, Linh Le, et al. "RAS: continuously optimized region-wide datacenter resource allocation". In: *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*. 2021, pp. 505–520.

[71] Jorge Nocedal and Stephen J. Wright, eds. *Numerical Optimization*. Springer Series in Operations Research and Financial Engineering. 1999.

[72] Brendan O'Donoghue, Eric Chu, Neal Parikh, and Stephen Boyd. "Operator splitting for conic optimization via homogeneous self-dual embedding". In: *arXiv preprint arXiv:1312.3039* 1.2.1 (2013), p. 3.

[73] Neal Parikh, Stephen Boyd, et al. "Proximal algorithms". In: *Foundations and trends® in Optimization* 1.3 (2014), pp. 127–239.

[74] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. "Pytorch: An imperative style, high-performance deep learning library". In: *Advances in neural information processing systems* 32 (2019).

[75] Yanghua Peng, Yixin Bao, Yangrui Chen, Chuan Wu, and Chuanxiong Guo. "Optimus: An Efficient Dynamic Resource Scheduler for Deep Learning Clusters". In: *Proceedings of the Thirteenth EuroSys Conference*. EuroSys '18. Porto, Portugal: Association for Computing Machinery, 2018. ISBN: 9781450355841. DOI: 10.1145/3190508.3190517. URL: https://doi.org/10.1145/3190508.3190517.

[76] Laurent Perron and Vincent Furnon. *OR-Tools*. Version v9.11. Google, May 7, 2024. URL: https://developers.google.com/optimization/.

[77] Yarin Perry, Felipe Vieira Frujeri, Chaim Hoch, Srikanth Kandula, Ishai Menache, Michael Schapira, and Aviv Tamar. "{DOTE}: Rethinking (Predictive){WAN} Traffic Engineering". In: *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*. 2023, pp. 1557–1581.

[78] Michael JD Powell. "A method for nonlinear constraints in minimization problems". In: *Optimization* (1969), pp. 283–298.

[79] Aurick Qiao, Sang Keun Choe, Suhas Jayaram Subramanya, Willie Neiswanger, Qirong Ho, Hao Zhang, Gregory R Ganger, and Eric P Xing. "Pollux: Co-adaptive cluster scheduling for goodput-optimized deep learning". In: *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*. 2021.

[80] Alec Radford, Karthik Narasimhan, Tim Salimans, Ilya Sutskever, et al. "Improving language understanding by generative pre-training". In: (2018).

[81] Pranav Rajpurkar, Jian Zhang, Konstantin Lopyrev, and Percy Liang. "SQuAD: 100,000+ Questions for Machine Comprehension of Text". In: *Proceedings of the 2016 Conference on Empirical Methods in Natural Language Processing*. Association for Computational Linguistics, 2016.

[82] Sasha Rakhlin and Karthik Sridharan. "Optimization, learning, and games with predictable sequences". In: *Advances in Neural Information Processing Systems* 26 (2013).

[83] Joseph Redmon and Ali Farhadi. "Yolov3: An incremental improvement". In: *arXiv preprint arXiv:1804.02767* (2018).

[84] Michel Schubiger, Goran Banjac, and John Lygeros. "GPU acceleration of ADMM for large-scale quadratic programming". In: *Journal of Parallel and Distributed Computing* 144 (2020), pp. 55–67.

[85] Marco Serafini, Essam Mansour, Ashraf Aboulnaga, Kenneth Salem, Taha Rafiq, and Umar Farooq Minhas. "Accordion: Elastic scalability for database systems supporting distributed transactions". In: *Proceedings of the VLDB Endowment* 7.12 (2014), pp. 1035–1046.

[86] Alexander Sergeev and Mike Del Balso. "Horovod: fast and easy distributed deep learning in TensorFlow". In: *CoRR* (2018).

[87] Noam Shazeer, Youlong Cheng, Niki Parmar, Dustin Tran, Ashish Vaswani, Penporn Koanantakool, Peter Hawkins, HyoukJoong Lee, Mingsheng Hong, Cliff Young, Ryan Sepassi, and Blake Hechtman. "Mesh-TensorFlow: Deep Learning for Supercomputers". In: *Advances in Neural Information Processing Systems*. 2018.

[88] Dharma Shukla, Muthian Sivathanu, Srinidhi Viswanatha, Bhargav Gulavani, Rimma Nehme, Amey Agrawal, Chen Chen, Nipun Kwatra, Ramachandran Ramjee, Pankaj Sharma, et al. "Singularity: Planet-scale, preemptive and elastic scheduling of AI workloads". In: *arXiv preprint arXiv:2202.07848* (2022).

[89] Prasoon Sinha, Akhil Guliani, Rutwik Jain, Brandon Tran, Matthew D Sinclair, and Shivaram Venkataraman. "Not all GPUs are created equal: characterizing variability in large-scale, accelerator-rich systems". In: *arXiv preprint arXiv:2208.11035* (2022).

[90] Muthian Sivathanu, Tapan Chugh, Sanjay S Singapuram, and Lidong Zhou. "Astra: Exploiting predictability to optimize deep learning". In: *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*. 2019, pp. 909–923.

[91] Bartolomeo Stellato, Goran Banjac, Paul Goulart, Alberto Bemporad, and Stephen Boyd. "OSQP: An operator splitting solver for quadratic programs". In: *Mathematical Programming Computation* 12.4 (2020), pp. 637–672.

[92] Rebecca Taft, Essam Mansour, Marco Serafini, Jennie Duggan, Aaron J Elmore, Ashraf Aboulnaga, Andrew Pavlo, and Michael Stonebraker. "E-store: Fine-grained elastic partitioning for distributed transaction processing systems". In: *Proceedings of the VLDB Endowment* 8.3 (2014), pp. 245–256.

[93] Jakub M Tarnawski, Deepak Narayanan, and Amar Phanishayee. "Piper: Multidimensional Planner for DNN Parallelization". In: *Advances in Neural Information Processing Systems*. 2021.

[94] John Thorpe, Pengzhan Zhao, Jonathan Eyolfson, Yifan Qiao, Zhihao Jia, Minjia Zhang, Ravi Netravali, and Guoqing Harry Xu. "Bamboo: Making Preemptible Instances Resilient for Affordable Training of Large DNNs". In: *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*. Boston, MA: USENIX Association, Apr. 2023, pp. 497–513. ISBN: 978-1-939133-33-5. URL: https://www.usenix.org/conference/nsdi23/presentation/thorpe.

[95] Muhammad Tirmazi, Adam Barker, Nan Deng, Md E Haque, Zhijing Gene Qin, Steven Hand, Mor Harchol-Balter, and John Wilkes. "Borg: the next generation". In: *Proceedings of the fifteenth European conference on computer systems*. 2020, pp. 1–14.

[96] Alexey Tumanov, Timothy Zhu, Jun Woo Park, Michael A Kozuch, Mor Harchol-Balter, and Gregory R Ganger. "TetriSched: global rescheduling with adaptive plan-ahead in dynamic heterogeneous clusters". In: *Proceedings of the Eleventh European Conference on Computer Systems*. 2016, pp. 1–16.

[97] Colin Unger, Zhihao Jia, Wei Wu, Sina Lin, Mandeep Baines, Carlos Efrain Quintero Narvaez, Vinay Ramakrishnaiah, Nirmal Prajapati, Pat McCormick, Jamaludin Mohd-Yusof, Xi Luo, Dheevatsa Mudigere, Jongsoo Park, Misha Smelyanskiy, and Alex Aiken. "Unity: Accelerating DNN Training Through Joint Optimization of Algebraic Transformations and Parallelization". In: *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*. Carlsbad, CA: USENIX Association, July 2022, pp. 267–284. ISBN: 978-1-939133-28-1. URL: https://www.usenix.org/conference/osdi22/presentation/unger.

[98] Sinong Wang and Ness Shroff. "A new alternating direction method for linear programming". In: *Advances in neural information processing systems* 30 (2017).

[99] Qizhen Weng, Wencong Xiao, Yinghao Yu, Wei Wang, Cheng Wang, Jian He, Yong Li, Liping Zhang, Wei Lin, and Yu Ding. "{MLaaS} in the wild: Workload analysis and scheduling in {Large-Scale} heterogeneous {GPU} clusters". In: *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*. 2022, pp. 945–960.

[100] Wencong Xiao, Romil Bhardwaj, Ramachandran Ramjee, Muthian Sivathanu, Nipun Kwatra, Zhenhua Han, Pratyush Patel, Xuan Peng, Hanyu Zhao, Quanlu Zhang, et al. "Gandiva: Introspective cluster scheduling for deep learning". In: *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. 2018, pp. 595–610.

[101] Wencong Xiao, Shiru Ren, Yong Li, Yang Zhang, Pengyang Hou, Zhi Li, Yihui Feng, Wei Lin, and Yangqing Jia. "AntMan: Dynamic Scaling on GPU Clusters for Deep Learning". In: *Proceedings of the 14th USENIX Conference on Operating Systems Design and Implementation*. OSDI'20. USENIX Association, 2020.

[102] Zhiying Xu, Francis Y Yan, Rachee Singh, Justin T Chiu, Alexander M Rush, and Minlan Yu. "Teal: Learning-accelerated optimization of wan traffic engineering". In: *Proceedings of the ACM SIGCOMM 2023 Conference*. 2023, pp. 378–393.

[103] Yu Yang, Xiaohong Guan, Qing-Shan Jia, Liang Yu, Bolun Xu, and Costas J. Spanos. *A Survey of ADMM Variants for Distributed Optimization: Problems, Algorithms and Features*. 2022. arXiv: 2208.03700.

[104] Ian En-Hsu Yen, Kai Zhong, Cho-Jui Hsieh, Pradeep K Ravikumar, and Inderjit S Dhillon. "Sparse linear programming via primal and dual augmented coordinate descent". In: *Advances in neural information processing systems* 28 (2015).

[105] E Alper Yildirim and Stephen J Wright. "Warm-start strategies in interior-point methods for linear programming". In: *SIAM Journal on Optimization* 12.3 (2002), pp. 782–810.

[106] Dan Zhang, Safeen Huda, Ebrahim Songhori, Kartik Prabhu, Quoc Le, Anna Goldie, and Azalia Mirhoseini. "A Full-Stack Search Technique for Domain Optimized Deep Learning Accelerators". In: *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS '22. Association for Computing Machinery, 2022.

[107] Hao Zhang, Yuan Li, Zhijie Deng, Xiaodan Liang, Lawrence Carin, and Eric Xing. "AutoSync: Learning to Synchronize for Data-Parallel Distributed Deep Learning". In: *Advances in Neural Information Processing Systems*. 2020.

[108] Peng Zhao, Yan-Feng Xie, Lijun Zhang, and Zhi-Hua Zhou. "Efficient methods for non-stationary online learning". In: *Advances in Neural Information Processing Systems* 35 (2022), pp. 11573–11585.

[109] Peng Zhao and Lijun Zhang. "Improved analysis for dynamic regret of strongly convex and smooth functions". In: *Learning for Dynamics and Control*. PMLR. 2021, pp. 48–59.

[110] Lianmin Zheng, Zhuohan Li, Hao Zhang, Yonghao Zhuang, Zhifeng Chen, Yanping Huang, Yida Wang, Yuanzhong Xu, Danyang Zhuo, Eric P. Xing, Joseph E. Gonzalez, and Ion Stoica. "Alpa: Automating Inter- and Intra-Operator Parallelism for Distributed Deep Learning". In: *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*. USENIX Association, 2022.

[111] Pengfei Zheng, Rui Pan, Tarannum Khan, Shivaram Venkataraman, and Aditya Akella. "Shockwave: Fair and efficient cluster scheduling for dynamic adaptation in machine learning". In: *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*. 2023, pp. 703–723.