University of Nevada, Reno

Towards Efficient AI for Science in Scalable and High Performance Distributed System

A dissertation submitted in partial fulfillment
of the requirements for the degree of Doctor of
Philosophy in Computer Science and Engineering

by

Xiaolong Ma

Dr. Feng Yan, Dissertation Advisor

Dr. Lei Yang, Dissertation Co-advisor

December 2024

© by Xiaolong Ma 2024
All Rights Reserved

The Graduate School

We recommend that the dissertation prepared under our supervision by

Xiaolong Ma

entitled

Towards Efficient AI for Science in Scalable and High Performance Distributed System

be accepted in partial fulfillment of the requirements for the degree of

DOCTOR OF PHILOSOPHY

Feng Yan, Ph.D., Advisor

Lei Yang, Ph.D., Co-Advisor

Dongfang Zhao, Ph.D., Committee Member

Rui Hu, Ph.D., Committee Member

Wenrong Cao, Ph.D., Graduate School Representative

Markus Kemmelmeier, Ph.D., Dean, Graduate School

December 2024

Abstract

Artificial intelligence (AI) has seen rapid development over the last few decades, significantly impacting various domains such as computer vision and natural language processing. In recent years, machine learning methods have been increasingly applied to the scientific discovery process, accelerating advances in diverse fields. Notable examples include AlphaFold, which predicts protein structures, and ClimateX, which enhances weather prediction capabilities. The ability of AI to process large volumes of data, recognize complex patterns with precision, and uncover intricate relationships has made it an indispensable tool for innovation across scientific domains.

Scientific research often demands extensive data processing and computation, typically facilitated by high-performance computing (HPC) clusters or major cloud providers such as AWS, Azure, and GCP. However, efficiently leveraging these infrastructures for accelerated scientific discovery poses significant challenges. This dissertation addresses the efficient management of AI-driven science workloads on scalable, high-performance distributed systems. Motivated by the requirements of extensive machine learning applications and the need to effectively handle large scientific datasets, this dissertation develops novel frameworks aimed at optimizing resource allocation on supercomputers, minimizing storage costs in the cloud, and harnessing scalable serverless resources for machine learning training. Additionally, this dissertation introduces an AI application for climate research, employing diffusion models for super-resolution and data assimilation to enhance climate prediction accuracy.

We first address the challenges prevalent in high-performance computing by analyzing supercomputer clusters at DOE National Laboratories. Our findings indicate that roughly

10% of the node resources in these clusters, including major installations like Aurora at Argonne National Laboratory (over 10,000 nodes) and Summit at Oak Ridge National Laboratory (over 4,000 nodes), remain unutilized by the main scheduler. Such underutilization represents a significant loss of computational potential. To address these challenges, we develop a framework named MalleTrain, which efficiently utilizes these otherwise wasted dynamic resources for scalable data-parallel distributed deep learning training.

Next, we investigate AI for science challenges in cloud computing, specifically the emerging paradigm of serverless computing. Our investigation includes examining serverless function instances for data caching. We introduce the InfiniCache framework, an in-memory caching system. Compared to AWS ElastiCache, InfiniCache achieves cost savings of 31 to 96 times for large object caching without compromising performance. We further explore the potential of using serverless computing for machine learning training. We introduce SMLT, a user-centric framework designed to facilitate scalable and adaptive machine learning training on public cloud platforms using serverless technologies.

Finally, we introduce WindSR, a diffusion-based framework tailored for wind speed super-resolution that innovatively incorporates data assimilation. This framework enables seamless integration of data assimilation techniques into a diffusion-based super-resolution model.

ACKNOWLEDGEMENTS

I would like to express my deepest gratitude to my advisor, Professor Feng Yan, and co-advisor, Professor Lei Yang, for their invaluable guidance throughout my Ph.D. journey. This dissertation could not have been accomplished without their support. Their dedication to research and professionalism has greatly benefited me, and their broad vision and extensive connections in both academia and industry have provided me with invaluable opportunities to collaborate with best researchers. The careful mentoring I received during my doctoral studies will continue to inspire me in my future work and learning endeavors.

I am also deeply grateful to Dr. Rajkumar Kettimuthu for his exceptional leadership and guidance on the exciting projects. During the last year and a half of my Ph.D., I was fortunate to work on-site as a long-term research intern and visiting student at Argonne National Laboratory. This invaluable experience enabled me to collaborate with exceptionally talented scientists across various domains. I had the chance of working on challenging yet impactful projects that contribute to scientific discovery and innovation.

I would like to extend my thanks to all my committee members, Professor Dongfang Zhao, Professor Rui Hu, and Professor Wenrong Cao, for their valuable suggestions, insightful feedback, and support throughout this dissertation.

I would like to express my gratitude to all the members of the Intelligent Data and Systems Lab (IDS Lab) and the Big Data Analytics and Informatics (BASIS) Lab. I have enjoyed memorable times discussing research, attending classes, and collaborating with everyone. Special thanks go to Dr. Heyang Qin, Dr. Jun Yi, Dr. Ali Ashan, Dr. Syed Za-

wad and Dr. Yunchuan Liu. I cannot imagine my Ph.D. journey without the companionship and support of these exceptional individuals.

I would also like to extend my gratitude to my collaborators, Dr. Jingyuan Zhang, Ao Wang, Benjamin Carver, and Prof. Yue Cheng. I feel incredibly fortunate to have had the opportunity to work with such outstanding individuals. We shared enjoyable collaborations and accomplished some fascinating work together. Their admirable qualities, such as their passion for research and dedication to excellence, have left a deep and lasting impression on me.

Special thanks to my parents for their endless love and support. Being a parent is one of the greatest roles in the world, and I am deeply grateful for everything they have done for me. Their love and encouragement have always meant so much to me.

I am grateful to the University of Nevada, Reno, the National Science Foundation, Amazon Web Services, and Argonne National Laboratory for their financial support and resources during my Ph.D. studies. This dissertation is based on work supported by NSF grants CAREER-2048044 and the Department of Energy, Office of Science, under contract DE-AC02-06CH11357. The views expressed in this work are solely my own and do not necessarily reflect those of the supporting institutions.

TABLE OF CONTENTS

Abstract **i**

Acknowledgements ii

List of Tables **xi**

List of Figures **xii**

1 Introduction **1**

1.1 Overview 1

1.2 Efficient Machine Learning Training on Supercomputer Cluster 2

1.3 Efficient Caching on Serverless Computing 7

1.4 Efficient Machine Learning Training on Serverless Computing 8

1.5 AI for Climate Application 13

1.6 Summary of Contributions 16

1.6.1 MalleTrain: Deep Neural Network Training on Unfillable Supercomputer Nodes 16

1.6.2 InfiniCache: Exploiting Ephemeral Serverless Functions to Build a Cost-Effective Memory Cache 17

1.6.3 Enabling Scalable and Adaptive Machine Learning Training via Serverless Computing on Public Cloud 17

1.6.4 WindSR: Diffusion-based, Data Assimilation enabled Wind Super Resolution 18

1.7 Organization 19

2 Background and Related Work 21

2.1 Efficient Machine Learning Training on Supercomputer Cluster 21

2.1.1 Cloud-Preemptable Instances 21

2.1.2 Fragment Resources on HPC 22

2.1.3 FreeTrain 24

2.1.4 Topology 26

2.2 Efficient Caching on Serverless Computing 28

2.3 Efficient Machine Learning Training on Serverless Computing 30

2.3.1 Modern Machine Learning Workflows 30

2.3.2 Machine Learning on the Cloud 32

2.4 AI for Climate Application 36

3 Deep Neural Networks Training on Unfillable Supercomputer Nodes 39

3.1 System Design and Realization 39

3.1.1 System Architecture Overview 39

3.1.2 Event-Driven Resource Adjustment 42

3.1.3 Job Profiling Advisor 43

3.1.4 Cluster Configuration 47

3.2 Evaluation and Discussion 48

3.2.1 Experiment Setup 48

3.2.1.1 Workload 50

3.2.1.2 Testbed 50

3.2.2 Performance Evaluation 51

3.2.3 Topology Impact Analysis 52

3.3 Related Work 54

3.4 Summary 55

4 Exploiting Ephemeral Serverless Functions to Build a Cost-Effective Memory Cache **57**

4.1 InfiniCache Design 57

4.1.1 Client Library 58

4.1.2 Proxy 61

4.2 Evaluation 62

4.2.1 Microbenchmark Performance 63

4.2.2 Summary 66

5 Enabling Scalable and Adaptive Machine Learning Training via Serverless Computing on Public Cloud 67

5.1 SMLT Design 67

5.1.1 Overarching View and Dynamic Adaptation 68

5.1.2 User-centric Deployment and Execution 69

5.1.3 End-to-end Scalability 73

5.2 SMLT Framework 77

5.2.1 End Client 77

5.2.2 Serverless Worker 81

5.2.3 Hybrid Storage 82

5.3 Evaluation 83

5.3.1 Experimental Setup 83

5.3.2 Effectiveness of Hierarchical Model Synchronization 85

5.3.3 Intermediate Storage: S3 vs Redis 86

5.3.4 User-centric Deployments 87

5.3.5 Dynamic Batching and Online Learning 89

5.3.6 Neural Architecture Search 91

5.4 Related Work 92

5.5 Summary 93

6 Diffusion-based, Data Assimilation enabled Wind Super-Resolution 95

6.1 Preliminary 95

6.1.1 Denoising Diffusion Probabilistic Models 95

6.1.2 Data 97

6.2 Method 100

6.2.1 Overview 100

6.2.2 Condition in Training 101

6.2.3 Data Assimilation in Inference 102

6.3 Experiment 104

6.3.1 Implementation 104

6.3.2 Comparison of Downscaling 106

6.3.3 Comparison of Data Assimilation 107

6.4 Summary 108

7 Conclusion and Future Work 110

7.1 Conclusion 110

7.2 Future Work 111

7.2.1 Large languagus model traning and serving on unfilled supercomputer nodes 111

7.2.2 Extend in memory caching system to persistent storage system on serverless platform 112

LIST OF TABLES

2.1 Queue types and their characteristics. *Queue* is the queue type name on the Polaris cluster, the *Min* and *Max* columns give minimum/maximum number of nodes, and time, allowed per job request, and *Priority* is the priority for jobs in the queue. 24

3.1 Example jobs-to-nodes map, as determined by MILP. Each row corresponds to a job, with scale given by the sum of the cells in the row; each column corresponds to a node, with at most one cell in the column with value 1 indicating the job to which the node is allocated. 43

5.1 Modules of SMLT and their functionalities. 76

6.1 Dataset specifications for WRF, HRRR, and HGT 99

6.2 The average SSIM and PSNR across 200 random sampled images for various models. 103

6.3 Different models with different radius comparison. 104

LIST OF FIGURES

2.1 Illustration of dynamic fragment resources on a portion of a cluster. At time *t*, there are three idle nodes in the `MalleTrain` resource pool. 22

2.2 Example of fragment resources distribution on Polaris (27th in the TOP500 supercomputer list on Nov. 2023). Red stars mark fragmented idle resources scattered on the cluster. Note: For clarity in presentation, the figure depicts a majority of the cluster rather than its entirety. 26

2.3 Characteristics of object sizes and access patterns in the IBM Docker registry production traces. 27

2.4 Scalability of Bert-Small and Bert-Medium using Siren [161]. 33

2.5 Scalability of Bert-Small and Bert-Medium using Cirrus [44]. 34

2.6 Per iteration computation time and cost distributions with varying deployment configurations for Bert-Medium, Bert-Small, Resnet-18 (Tensorflow), as well as Resnet-50 (MXNet). 36

3.1 Schematic of the `MalleTrain` architecture. Scavenger adopts idle nodes, Resource Allocator determines a map of nodes to jobs, Job Manager rescales jobs according to the map, Job Monitor tracks job progress, and Job Profiling Advisor manages the online profiling process. 40

3.2 Event-driven resource allocation process. 42

3.3 Rescaling overhead costs on Polaris A100 GPU nodes: (a) Time to scale up and down a single node, for different models; (b) Time to scale up different numbers of nodes, for ResNet-50 model. 44

3.4 Inverse-order rescaling sequence. The solid curve represents scale-up and the dashed curve scale-down. JPA aims to minimize the number of scale-up operations in order to reduce overhead. 45

3.5 Online profiling process. 45

3.6 Average time taken for an example MILP computation as the numbers of concurrent MILP computations performed on a single 32-core head node scales from 1 to 40. 47

3.7 Cumulative histograms of idle gap counts on Summit and Polaris, for short gaps (0–50 secs: left) and longer gaps (0–3600 secs: right). Polaris has more shorter gaps (≤60 secs) while Summit has more gaps in the range from 60 to 600 secs. 51

3.8 Idle nodes on Summit over two-week period. 51

3.9 Comparison histogram of fragment length between real logs and synthetic. Synthetic (10k) shows the statistics for 10k fragments, and Synthetic (1k) shows the statistics for 1k fragments. The synthetic traces keep the same distribution as that of the real log. 52

3.10 FreeTrain vs. MalleTrain performance for the NAS and HPO applications, as measured both with real logs on a simulator and synthetic logs on a real cluster. 53

3.11 We analyzed training performance for sample `MalleTrain` jobs under four different scenarios: *Same Group, Empty* (where all nodes are located within the same Dragonfly group and the cabinet is empty), *Same Group, Busy* (where all nodes are within the same Dragonfly group but are collocated with other jobs), *Different Group, Empty* (where nodes are distributed across two Dragonfly groups with the two cabinets empty), and *Different Group, Busy* (where nodes are distributed across two Dragonfly groups and collocated with other jobs). The results demonstrate consistent training speeds for both models across all scenarios. The error bars for the *Different Group, Busy* scenario reveal higher variances in training speed, indicating fluctuations occur primarily in this scenario. However, the average training speed remains consistent despite these fluctuations. . . 53

3.12 Trend analysis of model scalability on 32 Polaris nodes, each with 4 A100 GPUs. 54

4.1 InfiniCache architecture overview. Icon denotes EC-encoded object chunks. Chunks with same color belong to the same object. 58

4.2 InfiniCache client library (`CH`: consistent hashing). 58

4.3 The box-and-whisker plot of latencies as a function of the number of VM hosts touched per request. 60

4.4 InfiniCache proxy. 61

4.5 Microbenchmark performance. 63

4.6 Scalability of InfiniCache. 64

5.1 Comparison of Bayesian optimization and reinforcement learning in terms of accuracy and training overhead. 71

5.2 Hierarchical model synchronization mechanism. 75

5.3 Workflow of SMLT. Solid lines indicate data transfer, and dashed lines indicate control signals. 78

5.4 Communication time breakdown comparison of SMLT, Cirrus and Siren. . 84

5.5 Per iteration communication time comparison between SMLT, Cirrus and Siren. 85

5.6 Performance Comparison of Bert Medium Redis vs S3 as an external Storage. 87

5.7 (Scenario 1) Minimize the monetary cost with a 1-hour training time limit for Bert-Medium with Pytorch. Note that, SMLT has profiling time and cost while other frameworks do not. For a fair comparison, we also demonstrate the profiling time and cost in SMLT. 88

5.8 (Scenario 2) Minimize the training time with a $50 training cost budget for Bert-Medium with Pytorch. Note that, SMLT has profiling time and cost while other frameworks do not. For a fair comparison, we also demonstrate the profiling time and cost in SMLT. 88

5.9 Cost comparison of profiling and training via our Bayesian optimizer, for dynamic batching and online learning, using Resnet-50 with Pytorch. Experiments using Resnet-18 with Tensorflow produced similar results. . . . 89

5.10 Throughput comparison for dynamic batching. 90

5.11 Throughput comparison for ENAS. 91

6.1 The figure demonstrates the DDPM's forward process of adding noise and the reverse denoising process. 96

6.2 Correlation coefficient (left) and root mean square error (right) between WRF and HRRR data averaged over 5 days. Despite advances in NWP, significant model biases remain, especially at coarser resolutions and in areas where complex terrain makes parametrization difficult. Data driven approaches can rectify these differences without the need for computationally expensive physics. 96

6.3 The figure illustrates the relationship between terrain complexity and the corresponding average wind speed variance over five days, showing that more complex terrain correlates with greater wind speed variance. 99

6.4 Image generation process incorporating both data assimilation and diffusion to downscale wind speed data. We interpolate sparse observation data over the inference grid then apply a soft bleed mask. The observation pixels are in-painted to the simulation results, and the composited image is used to guide the image generation process through the diffusion model, extra terrain information serve as condition during the reverse diffusion process. 100

6.5 Visual comparison of different SR models 105

6.6 All six images represent the same geo-spatial location, and each row corresponds to images from the same time step. The images exhibit minimal model bias, indicated by white or near-white colors. 105

6.7 Cumulative density functions of wind speeds across different terrain regions of the US, comparing values recorded by HRRR, WRF, and our model. In both regions, data assimilation brings the distribution of wind speeds from our model closer to HRRR data. 106

CHAPTER 1

INTRODUCTION

1.1 Overview

Over the past few decades, the development of artificial intelligence (AI) has significantly impacted numerous fields, with a wide array of applications emerging [39, 66, 100, 106, 126, 147, 159, 192]. These applications span from smartphone software to scientific research tools, and from financial markets to industrial products. Notable examples can be found across various domains including biomedical [107, 108, 165, 186], climate science [74, 99], finance [72, 160], transportation [101, 102], wireless communication [179, 180], etc. In scientific research, breakthrough achievements include AlphaFold [82], which has revolutionized protein structure prediction; GNoME [119], which facilitates new materials synthesis; and the Pangu [39] climate model, which has enhanced weather forecasting accuracy. The integration of AI with science has become an irreversible trend, with ongoing projects continuing to inspire and revolutionize scientific discovery.

Scientific facilities generate vast amounts of data, from astronomical observations captured by telescopes like the James Webb space telescope [64] to diverse climate data [98] collected from around the globe, and high-resolution images from the Advanced Photon Source for material analysis [113]. This massive influx of data requires timely and efficient processing to benefit scientific endeavors. Much of this data is used to train surrogate machine learning models for specific tasks.

High-Performance Computing (HPC) and cloud platforms are essential for processing and storing scientific data. Researchers often utilize large scale computing resources from national laboratories, such as the Aurora and Polaris supercomputers at the Argonne Leadership Computing Facility, and the Frontier and Summit supercomputers at the Oak Ridge Leadership Computing Facility, in addition to university HPC centers and major cloud providers like AWS, Azure, and Google Cloud Platform (GCP). Modern Machine learning workflow are getting more and more complex and highly dynamic [96, 135, 148, 171], different machine learning workloads for various applications require diverse computing resources [95, 187, 189, 190], large amount of adaptive resource scheduling methods have been proposed to accommodate these fluctuations, ensuring efficient utilization of computing resources [29, 94, 115, 117, 162, 191, 193].

Particularly in the era of generative AI [24, 86, 97, 110], training transformer-based models, which require increasingly large datasets and have escalating parameter counts into the trillions, is both costly and carbon-intensive. For example, training models like GPT-3 [40] and GPT-4 [24] can cost millions of dollars. Therefore, enhancing the efficiency of these infrastructures is crucial not only to reduce expenses and save time for users but also to minimize the environmental impact. Our ongoing work is dedicated to improving the productivity of these platforms, addressing these critical needs.

1.2 Efficient Machine Learning Training on Supercomputer Cluster

Batch-scheduled high-performance computing (HPC) systems typically maintain a queue of runnable jobs, with the order in which queued jobs are run being determined by resource scheduling policies established by administrators to meet higher-level goals. For example,

the largest supercomputers often implement policies to encourage capability computing, wherein they prioritize large jobs that cannot run elsewhere. Other criteria, such as job wait time and recent usage by a user or group, may also be considered when determining job priorities. But regardless of policy goals, the fact that jobs are typically given exclusive access to a fixed number of nodes while running means that nodes will be idle whenever the number of free nodes is less than the number needed to run the next job (as identified by policy).

Backfilling [122], a method by which lower-priority, shorter, and/or smaller jobs are run on idle resources ahead of higher-priority jobs as long as they do not delay the start time of the higher-priority jobs, can reduce, but not eliminate, inefficiencies, which can be substantial.

For example, in 2012, a comprehensive analysis of a 12-month workload trace of the Kraken supercomputer showed an average utilization of 94% [176]; a four-year study of the Blue Waters system revealed that monthly utilization rarely exceeded 80%. [81]; and other studies have reported utilizations of around 90% [31, 112, 129]. These numbers can represent thousands of idle GPUs on large supercomputers.

One approach to enhancing utilization in such environments is to devise new approaches for structuring applications in `malleable forms` and for mapping these malleable applications to supercomputer resources. A malleable computation adapts its degree of parallelism at runtime in response to external requests [58], for example by using checkpointing for semi-automated stop/restart [157] or specialized languages and libraries [25, 41, 50, 51]. If well managed, malleable applications can improve system utilization and scheduling efficiency and reduce average response times, compared with unmalleable jobs. However, to realize these benefits, (a) malleable jobs need to be able to adapt dynamically to changing

resource allocations and (b) job schedulers must be able to expand or shrink their resources to improve system utilization, throughput, and/or response times.

In practice, the rigid nature of both commonly used programming models like MPI and many current schedulers makes writing and running malleable applications a daunting task, which is why few malleable applications exist.

One intriguing source of malleable applications is deep neural network (DNN) training. DNNs are being employed widely in scientific computing [65, 70, 84, 87, 103], and DNN training is becoming a major workload in today's supercomputers. Furthermore, deep learning frameworks such as AdaptDL [134], PyTorch TorchElastic [128], and Elastic Horovod [146] enable scaling up and down the number of workers dynamically during training at modest cost without requiring a restart. A DNN training job is divided into many smaller tasks (mini-steps) that can be fitted into node×time gaps in a supercomputer computing infrastructure. In other words, DNN training workloads can in principle be structured as malleable computations. However, practical realization of this malleability requires the ability to 1) determine, quickly and accurately, what mini-steps should be configured for different batch queue states, and 2) assign resources and computations to run those mini-steps.

Liu et al. recently showed how, given knowledge of scheduler state, the task of identifying mini-steps can be formulated as a deterministic mixed-integer linear programming–based resource allocation problem [112]. However, while they showed via simulation that this "FreeTrain" approach could construct effective schedules for real scheduler traces, they did not address the second task just listed, by providing a practical implementation of their proposed approach. This is a significant obstacle to the effective realization of malleable DNN training due to the need for several system components to coordinate and interact

coherently: idle resource management, job progress monitoring, resource negotiation, and resource allocation. These components as well as their coordination are not readily available in today's job schedulers that were designed for unmalleable computing tasks.

A second deficiency of the FreeTrain approach is that it requires users to provide accurate scaling information, such as measured throughput when using different numbers of nodes for DNN training jobs. Providing this information is a substantial challenge because in many modern DNN training workflows, such as neural architecture search (NAS) [109, 138, 194] and hyperparameter tuning (HPO) [92], jobs are generated on the fly based on results produced in previous iterations by methods such as reinforcement learning [194] and Bayesian optimization [57]. Thus, even experienced DNN experts do not know all the model details beforehand, let alone their scalability characteristics.

In the work reported here, we propose and demonstrate solutions to the two obstacles to the practical realization of malleable DNN training just noted. First, we present a malleable DNN training system architecture, `MalleTrain`, which achieves the efficient coordination of the required idle resource management, job progress monitoring, resource negotiation, and resource allocation functions. For instance, in order to make malleable scheduling decisions, the Resource Allocator must first get information about unfillable nodes from the batch scheduler (e.g., PBS [59] or Slurm [175]), profiling information from a profiler, and current running and waiting DNN jobs from the job monitor; then, it needs to control a DNN scaling framework (e.g., Elastic Horovod) to execute the scheduling decisions. Throughout this process, it must also avoid negative impacts on jobs submitted to the main batch scheduler.

Second, we address the challenge of obtaining accurate scaling information by introducing a lightweight job profiling advisor (`JPA`) to obtain automatically the information

required for making resource management decisions. `JPA` runs experiments whenever a DNN training task starts, according to a schedule that minimizes associated costs by taking advantage of the fact that removing a node is faster than adding a node in distributed DNN training. By thus obtaining accurate job information at modest cost, `JPA` permits the MILP to make more accurate decisions, with significant benefits in practice. We conducted extensive simulation evaluations using workloads from production supercomputer clusters, alongside experiments on a smaller cluster with synthetic logs derived from real Summit cluster logs. Our findings indicate that the more accurate information provided by `JPA` allows `MalleTrain` to achieve performance improvements of up to 22.3% relative to FreeTrain. In addition, it permits the scheduling of malleable DNN training applications, such as NAS and HPO, for which no performance information may be available.

This chapter thus makes three important contributions. First, we propose a system architecture for running malleable DNNs on supercomputers, and implement `MalleTrain` according to this architecture. Second, we propose a lightweight online profiler that employs an inverse-order profiling method to obtain accurate scalability information for dynamic DNN jobs. Third, we present results from both simulations with supercomputer traces and real-world executions on a cluster using synthetic traces that demonstrate the efficiency of these methods in harnessing previously idle nodes for DNN training—and thus the feasibility of using what may often be 10% or more of previously unfillable supercomputer nodes for large-scale DNN training.

1.3 Efficient Caching on Serverless Computing

Data-intensive applications are widely utilized in both internet-scale web applications and scientific research. For instance, popular web platforms like Instagram, YouTube, and TikTok serve billions of photo and video files daily [36], while large container image repositories, such as Docker Hub, accommodate vast numbers of resources [8]. These applications require substantial storage capacity to manage the massive volumes of data they generate. For example, Docker Hub hosts over 2.6 million container images, and Facebook produces approximately 4 petabytes (PB) of data each day [9].

Large object caching has been demonstrated to be effective and beneficial in cluster computing environments [33, 90, 137, 178]. To explore whether these benefits extend to web applications, we analyzed production traces from an IBM Docker registry [34] and identified two critical properties of large objects: (1) they exhibit heavy reuse with strong data locality but are accessed less frequently than smaller objects, and (2) they require fast access speeds for optimal system performance, though not as stringent as the sub-millisecond latencies demanded by smaller objects. These findings suggest that web applications could significantly benefit from large object caching—a feature that is notably absent in current In-Memory Object Caches (IMOCs).

Our method uses an In-Memory Object Cache (IMOC) created with cloud functions in a serverless application. Each function has memory to store objects while it runs, using this memory for caching. These functions activate only when needed, with FaaS providers managing the functions and their memory states. This allows objects to remain in memory between activations. Users are charged only when they request these functions, reducing costs significantly compared to traditional IMOCs, which charge continuously for memory

usage whether objects are accessed or not.

We summarize our main contributions as follows:

- Identify the opportunities and challenges of serverless function-based object caching by performing a long-term analysis of the internal mechanisms of a popular serverless computing platform (AWS Lambda [2]).
- Design and implement InfiniCache, the very first in-memory object caching system powered by ephemeral and "stateless" cloud functions.
- Provide an analytical model of InfiniCache's fault tolerance mechanism built using erasure coding and periodic delta-sync techniques.
- Perform an extensive evaluation and our experimental results show that InfiniCache achieves performance comparable to ElastiCache for large objects and improves the cost effectiveness.

1.4 Efficient Machine Learning Training on Serverless Computing

Motivation: The success of machine learning (ML) has prospered intelligent applications, such as translation [32], image classification [71] and autonomous driving [151]. Designing and training ML models have become a daily routine for many ML engineers and practitioners. In production ML systems, models are continuously improved, trained, and deployed for providing inference services. Therefore, today's ML design and training are part of a continuous workflow with different training tasks that have various dynamic resource demands, usually involving pre-processing, architecture engineering, hyperparam-

eter tuning and training via either manual efforts or automated approaches (e.g., neural architecture search [56]).

Limitation of state-of-art approaches: Thanks to its elastic resource offering, cloud is one preferred choice to perform ML design and training workloads. In particular, Machine-Learning-as-a-Service (MLaaS) offered by major cloud providers, such as Amazon SageMaker [76], Microsoft Azure [5], and Google Cloud [10], support optimized execution of ML tasks. Despite various cloud management tools, these systems usually require users to have extensive expertise. For instance, Amazon SageMaker provides hundreds of instance types with different computation and communication capabilities under a wide range of prices. Selecting the appropriate instance types for scaling up and the suitable number of instances for scaling out from such a wide selection is a non-trivial task. Recent efforts [91, 118, 173] on the optimization of ML task deployment target the automation of the deployment process. However, they incur significant monetary costs just for tuning the environment before training (up to 60% of the total) [173]. These approaches are not ideal for the state-of-the-art ML workloads that exhibit highly dynamic resource demands throughout the training process, such as NAS [56], dynamic batching [45] and online training.

Key insights and contributions: Serverless computing is an cloud computing paradigm with one popular incarnation being Function-as-a-Service, where the unit of computation is a function,[1] and it has been adopted by a diverse range of applications [123, 158, 184]. Serverless computing has great potential to match the requirements of ML workloads with dynamic resource demands, because it aims to eliminate the resource and scaling management from users and offers a flexible "pay-as-you-go" pricing model. As such, serverless

[1]Other incarnations of serverless computing include, e.g., the Container-as-a-Service with a full automation capability.

computing is gaining popularity for inference from both academia [30, 89, 116] and industry, with the latest example being the AWS SageMaker Serverless Inference [1, 3, 14].

On the other hand, ML model design and training on serverless platforms is still at an incipient stage. Initial attempts to enable ML training on serverless platforms [44, 60] are limited to specialized frameworks, support only simple approaches or small models (e.g., linear regression, MNIST), and scale poorly to larger and more sophisticated ML training tasks. Due to these limitations, the advantages in performance and cost when using serverless computing for large practical training tasks have been questioned [80].

Alleviating these concerns requires three challenges to be addressed. *First*, modern ML workflows usually have dynamic scaling and resource demands [117, 133]. For example, when the batch size [45] or model size changes during the training process, it may inevitably change the scalability and resource demands of the training tasks. Not adjusting the underlying serverless resources accordingly may result in lower performance and/or cost overheads. Hence, leveraging a generic serverless platform for ML training tasks requires overarching monitoring of the training dynamics, and a resource adaptation mechanism. The stateless nature of public cloud serverless platforms creates a challenge for such a mechanism.

Second, existing serverless platforms either do not provide user-centric deployment guarantees (such as meeting training deadlines and training budgets) or incur a heavy runtime overhead in doing so [44]. To achieve these goals and to optimize the cost and resource usage, the number of serverless functions and their memory configurations may need to be adjusted at a fine grained level for different ML tasks. These adjustments require going beyond simply monitoring the progress of the training pipeline, and introducing an optimizer that can, at runtime, automatically search for necessary configurations to meet a

user-specified goal.

Finally, the end-to-end ML training platform based on serverless computing needs to address the challenges arising with the increasing size and sophistication of the ML models as well as the different training frameworks that can be used. It is already challenging to scale ML training using the traditional infrastructure with dedicated computation and network resources, and using a serverless platform may amplify any existing issues. For example, the communication overhead for model synchronization already presents challenges; using stateless serverless functions that communicate via external storage can exacerbate these bottlenecks. In addition, the stateless functions may experience substantial repeated framework initialization overheads.

To address the above challenges, we propose `SMLT` — a public cloud serverless framework for scalable and adaptive ML design and training. To enable an overarching view of the ML workflow and the swift scale-adaptation, we propose an automated and adaptive scheduler for deploying and scaling training instances dynamically on the fly. Our scheduler can cater to varying resource requirements during the execution of ML tasks, such as the dynamic batch size scheduler [45] and the architecture exploration in NAS [56]. To achieve the user-centric deployment and execution guarantees, we offer a light-weight Bayesian optimizer to automate and optimize the ML task deployment and resource scaling to meet the user-specified performance and cost goals.

`SMLT` provides an end-to-end ML training service. It addresses the challenges caused by the stateless nature of serverless functions on public cloud, including the initialization and communication overheads. Specifically, it employs two key system components: 1) a hybrid storage consisting of a cloud object store for infrequently-accessed training data and an in-memory key-value store for frequently-accessed model parameters, and 2) a hierar-

chical model synchronization scheme exploiting parallelism during model synchronization. In addition, SMLT is designed to be agnostic of ML frameworks by abstracting the implementations of their common interfaces (e.g., input pipelines, gradient transfers).

Experimental methodology and artifact availability: To our knowledge, SMLT is the *first* fully-automated ML design and training framework that can support modern ML workflows in a scalable and cost-effective manner with the emerging serverless principles. To demonstrate its flexibility, we evaluate SMLT using three different ML frameworks (i.e., Tensorflow [23], Pytorch [127] and MXNet [47]), and show that our results are consistent across all of them. We perform our evaluation across a variety of ML workflows, including user-specified goals to minimize training time or budget, dynamic batching, online learning and NAS, and show that SMLT can reduce cost by up to 3x compared to the state-of-the-art. We also demonstrate that our system scales well with an increasing number of workers, leading to an overall reduction of up to 8x in the total training time when combined with other optimizations. We open-source the codebase of SMLT for public access[2].

Limitations of the proposed approach: One key challenge of SMLT is the unavailability of GPUs in typical public serverless platforms. However, we believe with the wide adoption of serverless computing in academia and industry for a wide range of applications, public cloud providers will soon offer GPUs for serverless computing as well. Initial prototypes of serverless platforms with GPU support have already started emerging [143], and ML training using GPUs in a serverless environment has already been employed in HPC [111] with promising results. We believe SMLT will demonstrate benefits on GPUs similar to our CPU evaluation.

[2]https://github.com/xiaolongm0/Enabling-Scalable-and-Adaptive-Machine-Learning-Workflows-via-Serverless-Computing

1.5 AI for Climate Application

The production of clean energy from renewable resources is critical for mitigating climate change. Among these, wind power is one of the most widely distributed and promising renewable energy resources, offering significant potential for global electricity generation [52, 68, 120]. The electricity generated by a wind turbine scales with the cube of wind speed [78, 169], highlighting the importance of accurate wind speed forecasting for efficient wind energy production [52]. Wind is a highly localized meteorological variable, both spatially and temporally, influenced by various environmental factors such as landuse and landcover, topography, and synoptic-scale atmospheric circulations or weather patterns [130]. Accurately characterizing wind resources thus requires data with fine spatial and temporal resolutions. While thousands of in-situ observations exist at 10m above ground level, observations at wind turbine rotor heights (ranging from dozens to 200m) remain extremely limited. Consequently, the wind energy community relies on model simulations, specifically downscaling techniques, to obtain high-resolution wind speed data [153], typically at spatial resolution of a few kilometers to tens of kilometers.

There are two conventional approaches for downscaling: dynamical downscaling and statistical downscaling. Dynamical downscaling uses numerical models to simulate atmospheric processes. It can produce comprehensive data, including wind, solar, and hydro resources, by solving physical equations that account for various atmospheric and earth system variables. This method uses low-resolution data as input and outputs high-resolution data. Such simulations are computationally expensive and time-consuming, making them less feasible for long-term resource assessments over large areas. Statistical downscaling, on the other hand, uses statistical methods to establish empirical relationships between large-scale atmospheric features and local climate variables [144]. This approach has less

computational demands and can focus on the variables of interest, such as wind speed. However, it depends heavily on observational data, which is often sparse in space and time. For example, wind speed observation data is sparse at hub-height levels.

In recent years, machine learning (ML) methods [79, 140, 172] have been widely adopted in downscaling applications. These methods combine the efficiency of conventional statistical downscaling with the ability to incorporate physical processes into the training process. By doing so, ML models produce more physically consistent and realistic results. For example, [163] integrates topography, sea level pressure, air temperature, and total atmospheric water vapor to generate high-resolution precipitation data using GAN-based methods. However, GAN-based models are difficult to converge, and their performance in super-resolution tasks is not as promising as some recent techniques, such as diffusion-based models.

Another line of research focuses on data assimilation, which is a widely used technique in climatology that integrates observational data from various sources, such as satellites and weather stations, with outputs from climate models. This approach refines and enhances the accuracy of weather forecasts and climate predictions, effectively bridging the gap between theoretical models and real-world observations. Data assimilation is particularly valuable for variables that are challenging to simulate accurately, such as wind and precipitation, which exhibit sharp spatial and temporal variability and depend heavily on the correct simulation of other atmospheric processes. Recent studies have demonstrated the effectiveness of incorporating data assimilation into machine learning forecasting models for meteorological variables. Although previous work DiffDA [74] adopts data assimilation to the diffusion process, they do not fully utilize physical world information such as terrain and pressure, and it treats every observation point equally, an approach that does not align with real-world scenarios and leads to suboptimal data assimilation performance.

To address the aforementioned issues of existing work and generate high-quality, high-resolution wind data, we introduce `WindSR`, a novel wind super-resolution approach that integrates the sparse, high quality observational data with the more spatially detailed simulation data using diffusion models. `WindSR` integrates a data assimilation process into a wind speed downscaling workflow by employing Denoising Diffusion Probabilistic Models (DDPM) [73]. Our approach leverages the power of diffusion models for downscaling while incorporating sparse observational data to reduce biases in the model-simulation data. We introduce a dynamic radius method to effectively merge observational data with simulation data, creating a new condition for the diffusion process. Unlike existing data assimilation work, we incorporate physical details such as terrain information into the training process. Our main workflow consists of three key steps: (1) training a deep learning-based super-resolution (SR) model with terrain information as extra condition; (2) inpainting sparse but invaluable observational data with dynamic impact radius into the wind data from numerical simulations; and (3) conditioning with this blended data to guide the super-resolution process.

We summarize our contributions below. 1) We introduce a new wind super-resolution approach `WindSR` powered by diffusion models and data assimilation. 2) We incorporate terrain information into the training phase, enhancing the downscaling performance of the diffusion model. 3) We employ a dynamic impact radius design within the diffusion model to optimize the effectiveness of the data assimilation process. 4) We prototype `WindSR` and conduct extensive evaluations against various deep learning models and settings demonstrate the effectiveness of our approach.

1.6 Summary of Contributions

We highlight the major contributions of this dissertation in this section.

1.6.1 MalleTrain: Deep Neural Network Training on Unfillable Supercomputer Nodes

First-come first-serve scheduling can result in substantial (up to 10%) of transiently idle nodes on supercomputers. Recognizing that such unfilled nodes are well-suited for deep neural network (DNN) training, due to the flexible nature of DNN training tasks, Liu et al. proposed that the re-scaling DNN training tasks to fit gaps in schedules be formulated as a mixed-integer linear programming (MILP) problem, and demonstrated via simulation the potential benefits of the approach. Here, we introduce MalleTrain, a system that provides the first practical implementation of this approach and that furthermore generalizes it by allowing it use even for DNN training applications for which model information is unknown before runtime. Key to this latter innovation is the use of a lightweight online job profiling advisor (JPA) to collect critical scalability information for DNN jobs – information that it then employs to optimize resource allocations dynamically, in real time. We describe the MalleTrain architecture and present the results of a detailed experimental evaluation on a supercomputer GPU cluster and several representative DNN training workloads, including neural architecture search and hyperparameter optimization. Our results not only confirm the practical feasibility of leveraging idle supercomputer nodes for DNN training but improve significantly on prior results, improving training throughput by up to 22.3% without requiring users to provide job scalability information.

1.6.2 InfiniCache: Exploiting Ephemeral Serverless Functions to Build a Cost-Effective Memory Cache

Internet-scale web applications, which are increasingly storage-intensive, rely heavily on in-memory object caching to achieve the necessary I/O performance. We propose that the serverless computing paradigm is ideally suited and cost-effective for object caching. Our system, INFINICACHE, is the first in-memory object caching system built entirely on ephemeral serverless functions. It utilizes serverless functions' memory resources for elastic, pay-per-use caching. INFINICACHE's innovative design incorporates erasure coding, intelligent control over billed duration, and an effective data backup mechanism. These features maximize data availability and cost efficiency while minimizing the risks associated with loss of cached data and performance degradation.

1.6.3 Enabling Scalable and Adaptive Machine Learning Training via Serverless Computing on Public Cloud

In today's production machine learning (ML) systems, models are continuously trained, improved, and deployed. ML design and training are becoming a continuous workflow of various tasks that have dynamic resource demands. Serverless computing is an emerging cloud paradigm that provides transparent resource management and scaling for users and has the potential to revolutionize the routine of ML design and training. However, hosting modern ML workflows on existing serverless platforms has non-trivial challenges due to their intrinsic design limitations such as stateless nature, limited communication support across function instances, and limited function execution duration. These limitations result

in a lack of an overarching view and adaptation mechanism for training dynamics, and an amplification of existing problems in ML workflows. To address the above challenges, we propose SMLT, an automated, scalable and adaptive serverless framework on public cloud to enable efficient and user-centric ML design and training. SMLT employs an automated and adaptive scheduling mechanism to dynamically optimize the deployment and resource scaling for ML tasks during training. SMLT further enables user-centric ML workflow execution by supporting user-specified training deadline and budget limit. In addition, by providing an end-to-end design, SMLT solves the intrinsic problems in public cloud serverless platforms such as the communication overhead, limited function execution duration, need for repeated initialization, and also provides explicit fault tolerance for ML training. SMLT is open-sourced and compatible with all major ML frameworks. Our experimental evaluation with large, sophisticated modern ML models demonstrates that SMLT outperforms the state-of-the-art VM-based systems and existing public cloud serverless ML training frameworks in both training speed (up to 8×) and monetary cost (up to 3×).

1.6.4 WindSR: Diffusion-based, Data Assimilation enabled Wind Super Resolution

Environmental changes have significant impacts on our planet and human lives. In response, deep learning has become a valuable tool in climate science, especially with super-resolution techniques that enhance the detail and accuracy of climate data, particularly in downscaling precipitation, solar radiation, and wind data. Recently, state-of-the-art diffusion models have proven to be highly effective in advancing downscaling, greatly increasing the precision of climate data predictions. Typically, wind data such as that from the Weather Research and Forecasting (WRF) and High-Resolution Rapid Refresh (HRRR)

is generated by various simulation models, resulting in different resolution and quality. High-quality or ground truth observational data is limited and challenging to obtain, while simulated data, though generally lower in quality, is more readily available and often produced at a finer spatial scale. To fully utilize both types of data for generating high-quality, high-resolution outputs, it is essential to assimilate the sparse, high-quality observational data with the more spatially detailed simulation data. In this work, we introduce `WindSR`, a novel approach that integrates sparse observational data with simulation data during downscaling, using diffusion models. A dynamic radius method is introduced to effectively merge observational data with simulation data, creating a new condition for the diffusion process. Terrain information is also incorporated during both the training and inference phases to improve data integration. We evaluate our methods against CNN and GAN-based models. Our approach outperforms these methods in both downscaling efficiency and data assimilation effectiveness.

1.7 Organization

The remainder of this dissertation explores topics including efficient machine learning training on supercomputer clusters, effective data caching in serverless computing, scalable machine learning training on serverless platforms, and an AI for climate application, showcasing the example of AI for science work. Chapter 2 introduce the background and related works of the dissertation, establishing the foundational context for the subsequent discussions. Chapter 3 details the innovative use of unfillable supercomputer nodes for machine learning training, presenting a solution to leverage these otherwise idle resources effectively. Chapter 4 investigates efficient data caching mechanisms on serverless computing platforms, highlighting the cost and performance benefits of our solution. Chapter

5 introduces a framework that enables scalable and adaptive machine learning training on serverless computing platforms within public clouds. Chapter 6 discusses WindSR, an AI application for climate modeling that integrates data assimilation into a diffusion-based super-resolution process, illustrating the practical impact of AI on climate science. The dissertation concludes with Chapter 7, which reflects on current and future work, summarizing research achievements, and delineating future developments and applications.

CHAPTER 2
BACKGROUND AND RELATED WORK

In this chapter we present the essential background and related works.

2.1 Efficient Machine Learning Training on Supercomputer Cluster

We present background information on the methods used by cloud providers to support malleability, fragmented resources in HPC, FreeTrain, and HPC network topologies.

2.1.1 Cloud-Preemptable Instances

Cloud providers such as AWS [18], Google [19], and Azure [20] make preemptable compute capacity available at a reduced cost via mechanisms such as AWS Spot Instances. For AWS, Spot Instances enable strategic utilization of surplus capacity; for users, they provide an opportunity to reduce their cloud expenses. To make use of such resources, however, users must be flexibile in their application runtime and tolerance for interruptions.

Spot Instances are particularly well suited for certain noncritical tasks such as data analysis, batch processing, and background operations. As noted, their costs are typically lower than for regular instances; on the other hand, they do not provide a time guarantee, introducing the possibility of unexpected interruption due to the cloud provider reclaiming

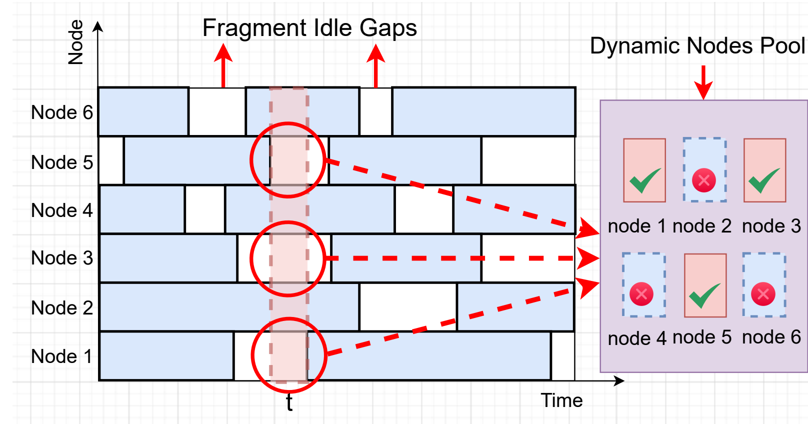


Figure 2.1: Illustration of dynamic fragment resources on a portion of a cluster. At time t, there are three idle nodes in the `MalleTrain` resource pool.

running instances. To mitigate the potential impact of such interruptions, cloud providers often grant a brief time window and prior notification to clients. This advance notice enables clients to reconfigure their workload distribution, effectively rebalancing the workload across available resources. By reallocating tasks and resources in response to an impending reclamation, clients can minimize disruptions and maintain a good level of user experience. Spot Instance resources in cloud environments resemble the preemptible HPC nodes addressed by `MalleTrain`.

2.1.2 Fragment Resources on HPC

As explored in recent research [31, 112, 129], leadership supercomputer clusters such as Mira, Theta, and Summit exhibit utilization rates of around 90%. Considering the substantial scale of these leadership supercomputer clusters, the unutilized resources become a significant concern. To put this in perspective, 10% idle capacity corresponds to 460 nodes on the 4608-node Summit and more than 1000 nodes on the 10,624-node Aurora.

Resource allocation within supercomputer clusters is typically managed by main schedulers such as Slurm [175] or PBS [59]. These schedulers administer multiple queues to prioritize resource assignments for user requests. As depicted in Figure 2.1, inevitable fragmentary resources emerge. These fragments may not always be backfilled, and (a portion of them) may remain unassigned. However, these seemingly negligible fragments are well suited for scalable and/or fault-tolerant workloads. The nature of these unassigned fragment resources resembles that of Spot VMs, as discussed in subsection 2.1.1. In subsequent sections we will refer to these fragment resources within supercomputer clusters as *preemptible nodes*. Their allocation timing lacks guarantees, rendering them unsuitable for typical fixed-size supercomputer workloads. Nonetheless, the paradigm of malleable applications, exemplified by DNN training, aligns seamlessly with this computational context. This suitability is underscored by several key factors: (1) DNN training demands substantial time and computational resources; (2) the distributed data-parallel training paradigm is inherently scalable; (3) leading DNN training frameworks, such as Horovod Elastic [146] and TorchElastic [128], adeptly support elastic training; and (4) DNN training often involves exhaustive searches for optimal neural network architectures and hyperparameters, consuming extensive computational resources.

The objective of `MalleTrain` is to empower users to effectively leverage the unfilled fragments in supercomputers. Some supercomputers have a preemptable queue (the jobs submitted to this queue may be preempted anytime) explicitly to encourage the use of the unfilled nodes. A preemptable queue can be designated for `MalleTrain` to which the users will submit adaptable DNN training jobs. `MalleTrain` will optimally manage the allocation of unfilled nodes by dynamically expanding and shrinking these adaptable DNN jobs. To incentivize the adoption of this preemptable queue, benefits such as reduced charges, in terms of either monetary cost or node-time consumption, can be extended to users.

Table 2.1
QUEUE TYPES AND THEIR CHARACTERISTICS. *Queue* IS THE QUEUE TYPE NAME ON THE POLARIS CLUSTER, THE *Min* AND *Max* COLUMNS GIVE MINIMUM/MAXIMUM NUMBER OF NODES, AND TIME, ALLOWED PER JOB REQUEST, AND *Priority* IS THE PRIORITY FOR JOBS IN THE QUEUE.

	Nodes		Time		
Queue	**Min**	**Max**	**Min**	**Max**	**Priority**
debug	1	2	5 min	1 hr	debug
debug-scaling	1	10	5 min	1 hr	debug
demand	1	56	5 min	1 hr	High
prod	10	496	5 min	24 hr	High
preemptable	1	10	5 min	72 hr	Low

Table 2.1 displays the different queue types in the Argonne Leadership Computing Facility (ALCF) Polaris cluster [21]. The low job priority means that nodes allocated for the job in the preemptable queue could be reclaimed.

2.1.3 FreeTrain

As noted earlier, FreeTrain [112] introduces an approach to dynamically allocating idle resources in which nodes and running job information are taken as inputs and user-defined metrics such as throughput or scalability are adopted as optimization objectives. By formulating the problem using MILP, FreeTrain is able to compute an optimal allocation of idle resources to DNN training jobs, subject to constraints such as allowed job size, feasible resource allocation, job scale information, and job migration overhead.

However, several practical challenges must be overcome before this approach can be realized into a production environment:

(1) **Expecting users to provide specific runtime job details can be a significant burden to users.** The MILP algorithm requires users to supply precise job-specific information, such as model training throughput and scalability, since these details serve as essential

inputs for the optimization process. This requirement will significantly increase the burden on users.

(2) **Job runtimes often correlate closely with specific hardware capability and configurations.** Thus, to attain accurate job runtime information, users would have to prerun their jobs under nearly identical system settings and hardware configurations. However, this approach would be prohibitively time-consuming and resource-consuming for most supercomputer users.

(3) **In some cases, heuristic algorithms rely on current models to predict future executions, making it impractical to preprofile all potential models.** The majority of HPO/NAS algorithms are heuristic [57, 109, 138, 194], which implies that the models to be evaluated are not predetermined until the current models have completed their execution. Thus, users will not be able to provide accurate job runtime information, a situation that will lead to an invalid resource allocation plan and will largely downgrade the performance of the system.

To overcome these challenges, an intelligent online profiling mechanism is needed. Such a mechanism should accurately collect job runtime information while minimizing disruptions to the regular execution of jobs.

`MalleTrain` also employs MILP to do the allocation optimization but emphasizes practical deployment aspects in supercomputer clusters. `JPA` can be integrated seamlessly into the workflow, orchestrating automatic profiling and obviating the need for manual input. As a result, the profiling procedure becomes an inherent facet of the process, efficiently alleviating the user from the need to provide such details beforehand. This dynamic profiling process operates in real time, eliminating the need to halt any ongoing jobs. While the pro-

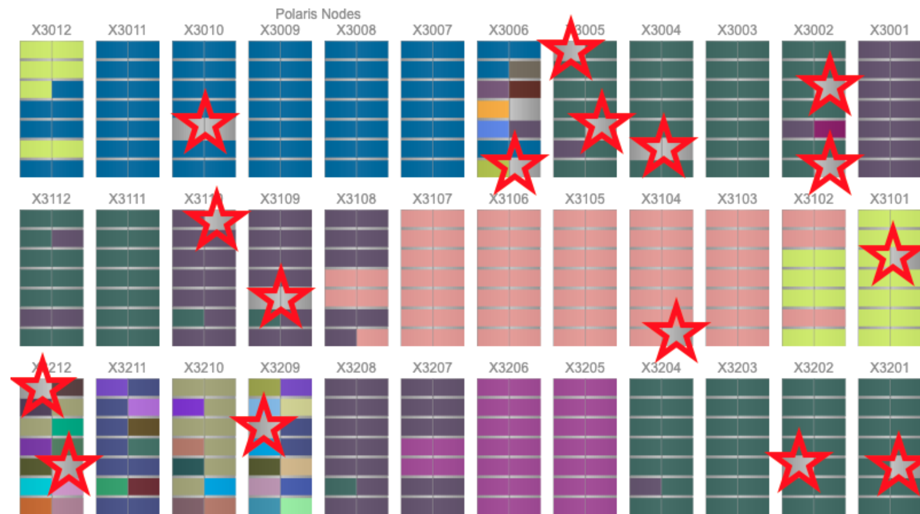
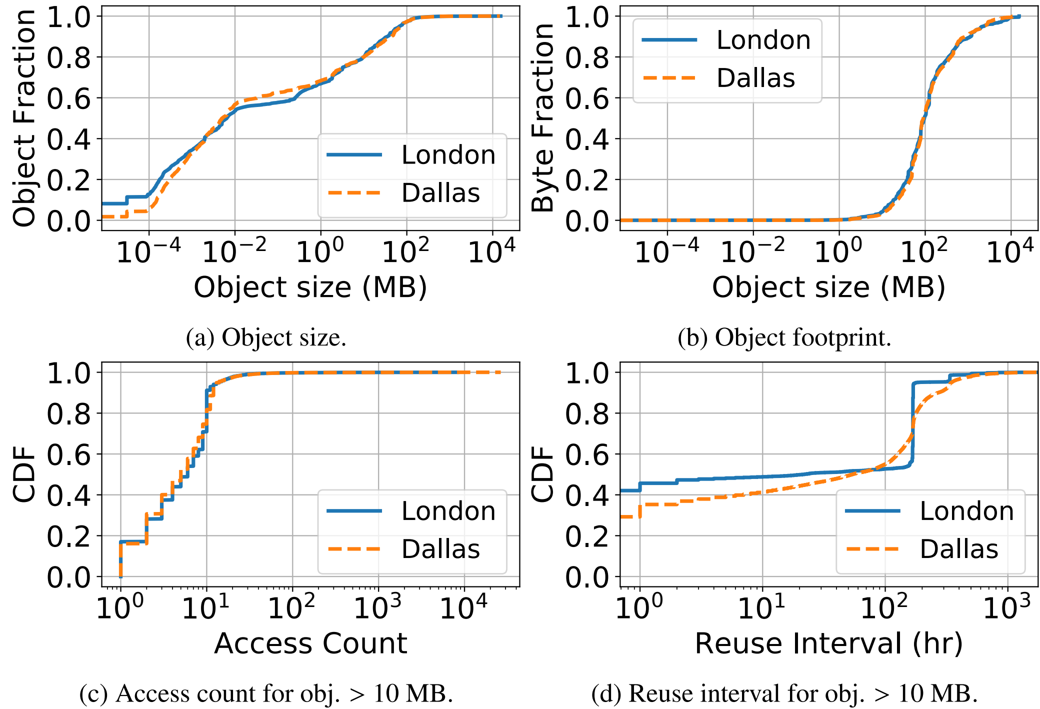


Figure 2.2: Example of fragment resources distribution on Polaris (27th in the TOP500 supercomputer list on Nov. 2023). Red stars mark fragmented idle resources scattered on the cluster. Note: For clarity in presentation, the figure depicts a majority of the cluster rather than its entirety.

filing phase may occasionally lead to suboptimal cluster performance, we mitigate potential overhead through the implementation of a carefully designed online profiling mechanism. Thus our design is able to obtain accurate profiling information without excessive operational costs.

2.1.4 Topology

The network topology in a supercomputer cluster plays an important role in facilitating efficient communication, seamless data transfer, and effective management of network resources. Today, the dragonfly [85] and fat-tree [28] topologies are widely utilized in supercomputer clusters due to their ability to deliver high bandwidth and low latency. These features make them adept at meeting the demanding requirements of modern high-performance computing environments. The Polaris cluster and upcoming Aurora cluster in the ALCF both use the dragonfly network topology, and the Summit cluster uses fat-tree. A major concern for fragmented idle resources in a supercomputer is that such resources



(a) Object size. (b) Object footprint.

(c) Access count for obj. > 10 MB. (d) Reuse interval for obj. > 10 MB.

Figure 2.3: Characteristics of object sizes and access patterns in the IBM Docker registry production traces.

will often be scattered and distant from each other, as shown in Figure 2.2. Each color represents a job; the nodes with same color were allocated to the same job. To fully utilize the inter connection bandwidth and reduce the latency, schedulers tend to assign the nodes into the same group or make them close to each other. For fragmented resources, however, usually the nodes are scattered into different topology groups. This scattering will have two major impacts. First, long distance usually means more hops are needed, which means the connections could have a higher fluctuation and cause a downgrade in the DNN training performance. Second, long distance could increase the end-to-end latency and cause more network resource contentions. We perform extensive evaluation and show that the topology is not a critical bottleneck for the design of `MalleTrain`.

2.2 Efficient Caching on Serverless Computing

Large-scale web applications are characterized by increasingly complex storage workloads. Many of these applications adopt a microservice architecture, comprising hundreds to thousands of individual modules [63]. These modules vary significantly in their object size distributions and request patterns. For instance, a Docker image registry service may use Redis to manage small-sized container metadata (e.g., manifests), while larger container images are stored in an object store [34, 105]. Although in-memory caching for small objects has been extensively studied within the context of large-scale web applications, the management and optimization of cloud caches for large objects remain underexplored. This oversight presents additional challenges and opportunities for enhancing cloud cache efficiency.

To obtain a better understanding of large object caching, we analyze production traces from an IBM Docker registry collected in 2017 from two datacenters (one in London, UK, and the other in Dallas, US) [34]. The goal is to reveal patterns that enable us to make realistic assumptions for the design of InfiniCache.

Extreme Variability in Object Size. We first analyze the object size distributions. As shown in Figure 2.3a, we find that object sizes span over nine orders of magnitude, and that more than 20% of objects are larger than 10 MB in size. This observation highlights the extreme variability and heterogeneity of real-world object store workloads, which further increases the complexity of cloud IMOC management.

Tension between Small and Large Objects. Efficiently managing both small and large objects in an IMOC is challenging due to two performance-cost tradeoffs. First, with limited cache capacity, large objects occupy a large amount of memory and would cause evictions of many small objects that might be reused in the near future, thus hurting performance. This is evidenced by Figure 2.3b, where large objects (with size larger than 10 MB) occupy more than 95% of the total storage footprint. Second, large object requests typically consume significant network bandwidth resources, which may inevitably affect the latencies of small objects.

On one end, to prevent large objects from consuming too much memory and starving small object requests, an object size threshold is defined to not admit objects larger than the threshold [17, 37]. On the other end, system administrators can simply provision more memory (and thus more servers) to increase the capacity of the cache. However, this would increase the total cost of ownership (TCO) with reduced resource utilization. In fact, according to our analysis of the production Docker registry workloads, for the busiest deployment among seven datacenters, the average throughput of requests with object sizes greater than 10MB is below $3,500$ `GETs` per hour.

Caching Large Objects Matters. While large object caching is challenging, it can provide significant benefit as large object workloads exhibit strong data locality. Figure 2.3c plots the access frequency distribution for all objects larger than 10 MB. About 30% of large objects are accessed at least 10 times, and the object popularity shows a long-tail distribution, with the most popular objects absorbing more than 10^4 accesses. Figure 2.3d shows the temporal reuse patterns of the large object workloads. Around 37%–46% large objects are reused within 1 hour since the last time they were accessed. The strong temporal locality patterns underscore the benefit for caching large objects for web applications.

2.3 Efficient Machine Learning Training on Serverless Computing

In this section, we give an overview of the latest ML workflows and how these workflows utilize cloud resources. We further motivate our work by discussing the state-of-the-practice and state-of-the-art cloud ML systems and identifying their limitations.

2.3.1 Modern Machine Learning Workflows

Distributed Machine Learning. Distributed data parallel training is typically employed to train modern ML models, which can contain up to billions of learning parameters [173], and require terabytes of training data [42]. Each worker node trains the model on a partition of the training data, and synchronizes weight updates frequently with other nodes. The choice of the synchronization approach can have a big impact on the training speed. In practice, Parameter Servers (PS) [7, 26, 93] and Ring-allreduce [38, 42, 47, 146] are typically used to coordinate the synchronization of weights across all worker nodes.

Machine Learning with Dynamic Resource Requirements. The life cycle of an ML model development includes pre-processing the training data, designing an appropriate model, tuning hyperparameters, and training. The resource requirements across these phases as well as within each phase can vary significantly and often require dynamic resource provisioning. For example, during the model design, engineers usually employ the autonomic design technique known as Neural Architecture Search (NAS) [55, 56] to find an optimized model architecture. During this phase, different architectures are crafted, trained, and evaluated according to the evaluation results of the previous trials, resulting in varying resource requirements for the new candidate model. Unless these new require-

ments are met by scaling the underlying memory and communication resources, the optimal execution cannot be achieved in terms of throughput and monetary cost.

As new architectures with different resource demands (e.g., a larger candidate model is more memory and communication demanding) are generated based on previous evaluation results, the underlying resources also need to be scaled to achieve the optimal execution in terms of both throughput and monetary cost. compute resource demand varies significantly. NAS automates the network architecture engineering and parameter tuning via deploying models of various architectures and hyper-parameters in parallel to learn a network that can achieve a best performance on a certain task.

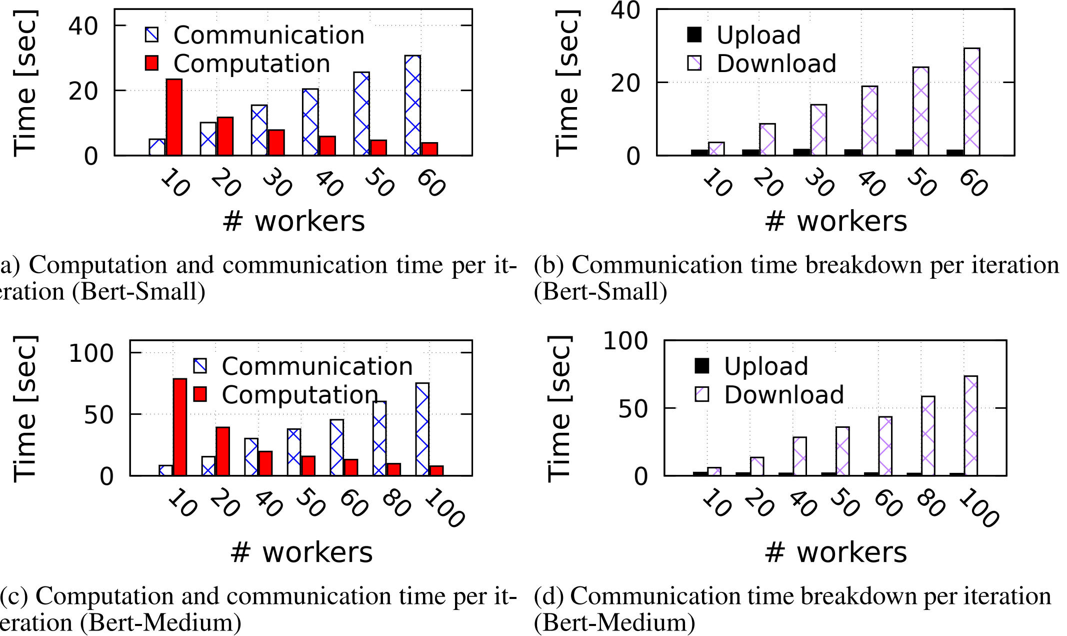
Similarly, a model training task often employs dynamic batching [182] to improve the training speed and model accuracy, such that the batch size can change between iterations or epochs during training. As a result, memory requirements may need to be adjusted. In addition, a larger batch size presents opportunities for intra- and inter-node parallelization, such that having more workers can benefit the training process. With dynamic batching, the batch size of different iterations or epochs changes during the training process. This in turn changes the memory demands as well as scaling potential of the training task, e.g., larger batch size takes more advantage of both intra- and inter-node parallelization and hence could benefit from having more workers. For these reasons, modern ML workflows often have dynamic resource demands and require effective auto-scaling for optimal execution. `SMLT` is designed to handle such dynamic resource requirements, and we demonstrate its effectiveness after applying it to NAS and dynamic batching systems in Section 5.3.

2.3.2 Machine Learning on the Cloud

Many existing ML systems for training large-scale models require a significant amount of computational resources that may not be available in a user's own infrastructure. Cloud computing and MLaaS systems can address this problem [139]. Hereafter, we briefly describe some well-known solutions.

Infrastructure-as-a-Service (IaaS). Cloud providers offer virtual machines (VMs) with a wide range of computational capabilities. This scheme is known as Infrastructure-as-a-Service (IaaS). IaaS provides on-demand resources with a time-based cost model. In an IaaS environment, users are fully responsible for deploying, managing, and provisioning the compute resources based on their requirements. Although it is a flexible option for practitioners, the extensive resource management overhead and limited scaling support make IaaS a less ideal candidate for training modern ML models, especially for ML practitioners with limited system expertise.

Machine-Learning-as-a-Service (MLaaS). To address the challenges of running ML tasks in an IaaS environment, cloud providers offer Machine-Learning-as-a-Service (MLaaS) to provide additional support for ML tasks. Many commercial MLaaS systems are still based on a pool of VMs with varying computational and communication capabilities, require a shared storage for training data, and offer some degree of scalability. Although presenting a plethora of options to choose from [6, 10, 15], these platforms require users to manually configure the required resources. Furthermore, to cover dynamic resource requirements during ML tasks and provide robustness against failures (e.g., out of memory), they typically require users to over-provision the configured resources, incurring inefficient resource utilization and high monetary cost. These problems are exacerbated when the ML



(a) Computation and communication time per iteration (Bert-Small)

(b) Communication time breakdown per iteration (Bert-Small)

(c) Computation and communication time per iteration (Bert-Medium)

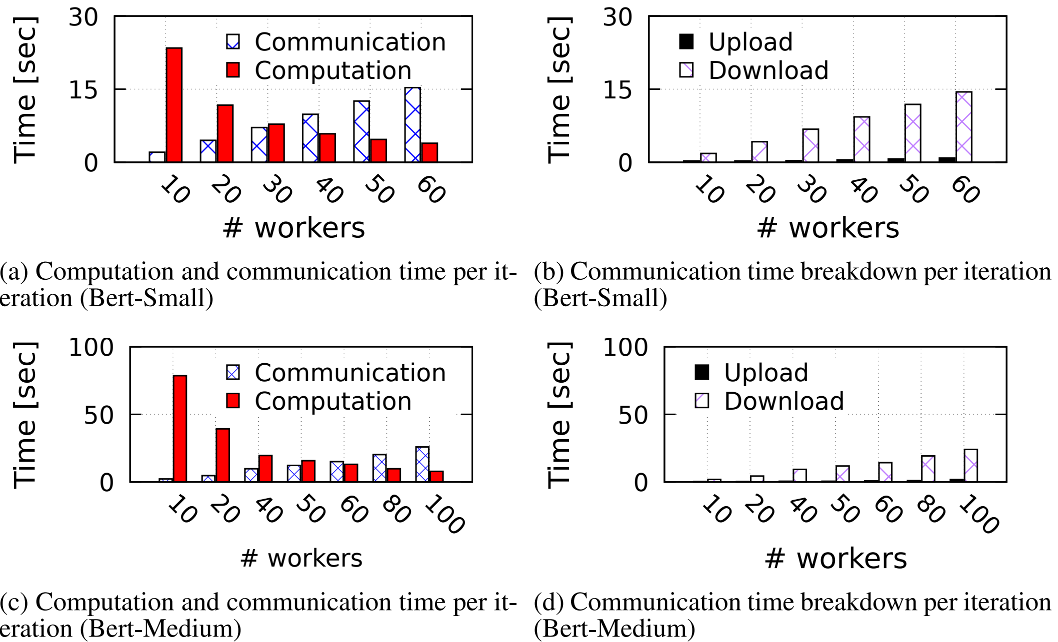
(d) Communication time breakdown per iteration (Bert-Medium)

Figure 2.4: Scalability of Bert-Small and Bert-Medium using Siren [161].

workflows include continuous learning and training with continuously incoming training data. As a result, ML practitioners still lack a fully-automated MLaaS solution to achieve dynamic resource allocation for various ML workflow phases with both performance and cost optimizations.

Machine Learning with Serverless Computing. Function-as-a-Service (FaaS), along with the Container-as-a-Service (CaaS) with a full-automation capability, are widely-realized forms of serverless computing and have been gaining popularity in both industry and academia. The elastic nature and the fine-grained resource management make these serverless realizations attractive to many application developers, including ML design and training developers. Besides removing the burden of resource management and scaling from ML developers, they also offer a flexible billing model and the continuously-improved execution startup delays [27, 121, 124].

In fact, FaaS has been actively studied for ML model serving [30, 38] and ML train-



(a) Computation and communication time per iteration (Bert-Small)

(b) Communication time breakdown per iteration (Bert-Small)

(c) Computation and communication time per iteration (Bert-Medium)

(d) Communication time breakdown per iteration (Bert-Medium)

Figure 2.5: Scalability of Bert-Small and Bert-Medium using Cirrus [44].

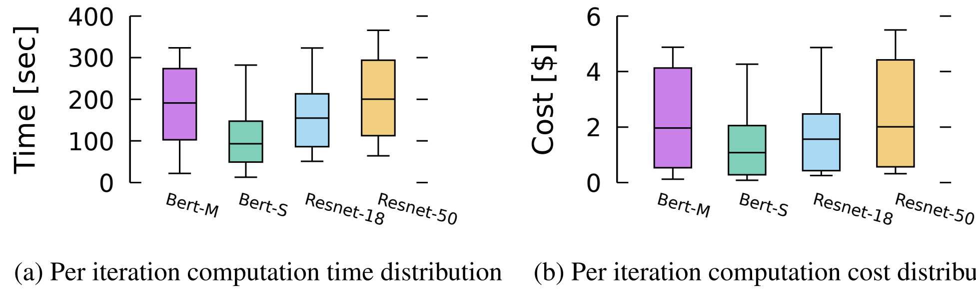
ing [43, 44, 60, 80, 161]. Although it is easy to see that model serving can benefit from FaaS' advantages (e.g., scalability, billing flexibility), the benefits for ML training are still not clear. Cirrus [44] is a specialized ML framework that addresses the resource limitations (e.g., memory, storage) in FaaS environments and applies it to linear regression. Siren [161] showcases that cost efficiency and training speed can be improved by adjusting the number of workers during different phases of training

Although these frameworks demonstrate that FaaS can be a viable alternative for ML training, their evaluation with small models and limited resources raises questions about the feasibility of training large-scale models with FaaS and serverless computing. It would not be surprising that increasing the number of training workers to process large ML models will significantly increase the communication overhead, the training time and monetary cost. For example, in a recent study, Jiang et al. [80] compare IaaS and FaaS platforms, and highlight issues in serverless settings that affect the performance and cost of ML training.

To confirm these questions, we evaluated two natural language models (i.e., BERT-small [142] and BERT-medium [155]) using these two approaches.[1] Figure 2.4 shows that the computation time decreases with increasing number of workers in Siren [161], but so does the communication time due to the use of S3 for storing intermediate parameter updates. With more than 20-40 workers, the total training time increases due to the communication overhead. Figure 2.5 shows a similar behavior for Cirrus [44]. These experiments show that communication can be a major bottleneck for training complex ML models in a serverless environment.

We further evaluated the performance of four models with varying number of workers between 10 to 200 and with three different sizes of memory allocation to these workers (3 GB, 6 GB and 10 GB). Figure 2.6a and 2.6b show the training time distribution and the cost distribution per iteration, respectively. These results highlight that an incorrect selection of workers and inefficient resource allocation to them can have significant impacts on training time and cost in a public serverless environment. Furthermore, the high variance in these values shows that it is not trivial to find the 'right' values for these parameters. Cirrus [44], Siren [161] and LambdaML [80], however, assume that the users know these values *before* running the training. In other words, these systems do not support dynamic resource allocation during training, leading to sub-optimal training time and causing higher monetary costs. In summary, these systems lack support for the needs of modern ML workflows with large models, dynamic resource requirements and user-centric training goals.

[1] Note that Siren's source code is not available, and Cirrus does not support the ML frameworks for these models. Nevertheless, we replicated these approaches to the best of our abilities.



(a) Per iteration computation time distribution (b) Per iteration computation cost distribution

Figure 2.6: Per iteration computation time and cost distributions with varying deployment configurations for Bert-Medium, Bert-Small, Resnet-18 (Tensorflow), as well as Resnet-50 (MXNet).

2.4 AI for Climate Application

Deep learning for super resolution Deep learning has transformed the field of image super-resolution (SR), a technique that enhances the resolution of images. Unlike traditional SR methods, deep learning employs neural networks to reconstruct high-resolution images from low-resolution inputs, delivering superior quality and efficiency. One of the pioneering deep learning models in super-resolution is the SR convolutional neural network (CNN) [53], which utilizes a three-layer neural network to learn an end-to-end mapping between low and high-resolution images. The layers in this model are designed for patch extraction and representation, non-linear mapping, and reconstruction, respectively. The SRCNN [53] demonstrated significant improvements over traditional methods like bicubic interpolation, setting a new benchmark for the quality of super-resolved images. Enhanced SR generative adversarial network (ESRGAN) [166] utilizes a generative adversarial network (GAN) framework to produce more realistic and sharper high-resolution images. Built upon the foundational concept of generative adversarial networks, ESRGAN employs a generator and a discriminator: the generator is tasked with creating high-resolution images, while the discriminator assesses their quality against actual high-resolution images, guiding the generator towards producing more accurate results. Wang et al. 2021 [163] used ESRGAN to successfully downscale precipitation from 50 km to 12km with

reasonable details over complex terrain, and demonstrated the ESRGAN can produce more realistic results than all previous SR methods and interpolation approaches. Stengel eta l. 2020 [150] used ESRGAN to downscale future projections of wind and solar resources from earth system models at a spatial resolution of 100 km to 2km.

Diffusion models The SR3 [141] is a diffusion model designed specifically for image super-resolution, it leverages the Denoising Diffusion Probabilistic Models (DDPM) [73] to enhance low-resolution images into high-resolution. Starting from a noisy low-resolution image, the model progressively denoises the image step-by-step, each time refining the details and clarity to achieve a high-resolution output. SR3 could condition on the low-resolution input while generating the high-resolution output. This conditioning helps guide the denoising process, ensuring that the generated details are consistent with the original image context. In the training phase, SR3 learns to reverse the process of adding noise to high-resolution images (the forward process). In the inference phase, it uses the learned reverse process starting from a low-resolution image, gradually reconstruct the details and lead to a high-resolution image.

Diffusion models in climate Diffusion models have been increasingly applied in climate science, offering more accurate and efficient data generation compared to GAN frameworks [e.g., 46]. These models are particularly relevant to climate data in three key aspects. First, diffusion models were originally developed for image generation tasks [73]. Climate data at each time step can be conceptualized as images, where the data patterns vary significantly between time steps. Unlike images of human faces or animals, climate data lack a consistent prior spatial distribution and often exhibit sharp gradients, making them more challenging to learn. Diffusion models excel in handling such complexities. Second, unlike deterministic downscaling methods, diffusion models can generate a large ensemble of realizations by sampling from a probability distribution, enabling uncertainty quantification

in downscaling. For example, Ling et al. (2024) [104] employed a diffusion probabilistic downscaling model to create a 180-year dataset of monthly surface variables for East Asia. This dataset, which enhances the resolution of global climate data from 1° to 0.1°, provides detailed local-scale insights into climate changes over the past centuries, crucial for regional adaptation and planning. Finally, the step-by-step Markov process and probabilistic framework of diffusion models offer a robust approach to data assimilation.

CHAPTER 3

DEEP NEURAL NETWORKS TRAINING ON UNFILLABLE SUPERCOMPUTER NODES

3.1 System Design and Realization

`MalleTrain` manages the residual resources of a supercomputer cluster, in other words, those that at any particular moment have not been allocated directly by the main scheduler. Two major challenges for `MalleTrain` arise in utilizing such residual resources: (1) their availability varies dynamically, and (2) they are preemptible. The `MalleTrain` design enables these resources to be utilized fully for parallel DNN training. `MalleTrain` seamlessly integrates with mainstream schedulers such as Slurm or PBS on supercomputer clusters. It operates without impacting the main scheduler, exclusively controlling the non-trivial, dynamic, residual resources that the main scheduler cannot utilize.

3.1.1 System Architecture Overview

Figure 3.1 shows the `MalleTrain` architecture and its five primary components, which we describe in the following:

Scavenger detects and collects idle nodes from the main job scheduler for `MalleTrain`. Two primary approaches could be employed: an event-driven mechanism, whereby the

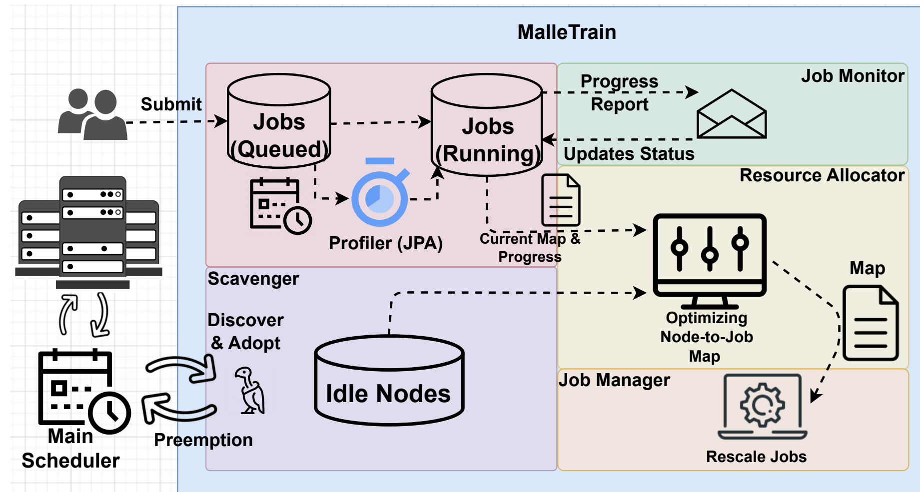


Figure 3.1: Schematic of the MalleTrain architecture. Scavenger adopts idle nodes, Resource Allocator determines a map of nodes to jobs, Job Manager rescales jobs according to the map, Job Monitor tracks job progress, and Job Profiling Advisor manages the online profiling process.

main scheduler alerts MalleTrain to idle nodes, or a proactive strategy, in which Scavenger periodically polls to find available unused (but preemptible when the main scheduler needs them) resources. The latter approach, preferred for its autonomy, requires no additional action from the main scheduler, ensuring seamless and efficient use of idle nodes by MalleTrain.

Resource Allocator maps nodes to DNN jobs in such a way as to optimize a given metric such as throughput or scaling efficiency. The allocation task can be formulated as a mathematical programming problem. In this chapter we adopt the formulation of [112] for resource allocation. The Resource Allocator is event-driven, with four types of events being considered: new nodes joining MalleTrain, nodes being recalled by the batch scheduler (i.e., the corresponding jobs are preempted), arrival of new MalleTrain jobs, and MalleTrain jobs completing.

Job Manager manages all jobs and implements the jobs-to-nodes mapping made by the Resource Allocator.

Job Monitor tracks job progress by consuming (current global batch size, timestamp) records generated by DNN training jobs via one line of `MalleTrain`-supplied code added to the training loop. The Monitor module then computes the current throughput as well as the cost incurred for each rescale operation and updates that information in a job records table to be used by the Resource Allocator.

Job Profiling Advisor manages the online profiling process, as described in subsection 3.1.3. The `JPA` is an independent component that starts work before the job entering the Resource Allocator.

When nodes cannot be backfilled by the main scheduler, they are redirected to the Scavenger for utilization. Jobs submitted by users to `MalleTrain` await the availability of nodes. As nodes become available, the jobs at the front of the queue commence execution. The running jobs transmit progress updates to the Job Monitor via a socket client. The system's architecture ensures continuous reporting of both cluster node statuses and job execution information to the Resource Allocator. The Allocator then employs MILP based on the current job distribution and number of nodes in the Scavenger. The MILP algorithm devises a strategic plan, which is represented by a map and subsequently conveyed to the Job Manager. The Job Manager then implements this plan to adjust resources accordingly. The events described in subsection 3.1.2 will trigger the Resource Allocator to run MILP and generate a new adjustment plan.

Users are provided with the option to explicitly indicate whether their job requires profiling. If so, the `JPA` consults with the Resource Allocator to assess the availability of necessary node resources for profiling. Should resources be insufficient, the jobs are returned to the queue. Conversely, if adequate resources are available for profiling, the job proceeds through the profiling process. This process uniquely involves an inverse order

of node numbers; further details are given in subsection 3.1.3. When the profiling process is done, the profiled job information will be an input to the MILP to find the optimal allocation.

3.1.2 Event-Driven Resource Adjustment

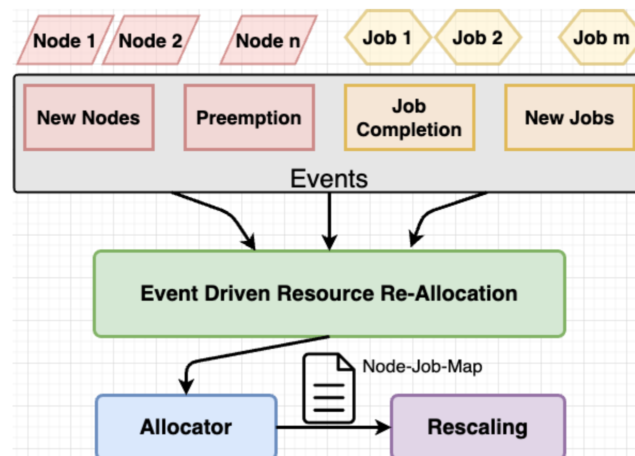


Figure 3.2: Event-driven resource allocation process.

Our event-driven resource management architecture is shown in Figure 3.2. There are four types of events:

New Nodes indicates that one or more nodes have become available to `MalleTrain`.

Preemption can be initiated at any time by the main scheduler without any prior notification. The jobs being run by `MalleTrain` on the preempted nodes are terminated and the nodes returned to the main batch scheduler.

Job Completion. `MalleTrain` picks a maximum of the top (first come, first serve, FCFS) jobs from its queue to prevent excessive hunger of low-priority jobs (e.g., low-throughput jobs when sample processed per second is the target to optimize). All the selected jobs are

launched by `MalleTrain` via spawning a process using a `subprocess` module of Python in a nonblocking fashion. The exit/completion of a job is thus notified from the Job Monitor module of `MalleTrain`.

A **New Jobs** event can trigger resource allocation only when the number of currently running jobs, Nj_{run}, is less than the jobs number threshold allowed in `MalleTrain`, Pj_{max}. When more than one job is submitted as a batch (e.g., grid search of a hyperparameter search), $Pj_{max} - Nj_{run}$ jobs will be added to the running list as a batch to reduce the rescaling cost. When the number of arrving jobs Nj_{arrive} is larger than $Pj_{max} - Nj_{run}$, the $Nj_{arrive} - (Pj_{max} - Nj_{run})$ jobs will be put into the FCFS queue for future execution.

Table 3.1
Example jobs-to-nodes map, as determined by MILP. Each row corresponds to a job, with scale given by the sum of the cells in the row; each column corresponds to a node, with at most one cell in the column with value 1 indicating the job to which the node is allocated.

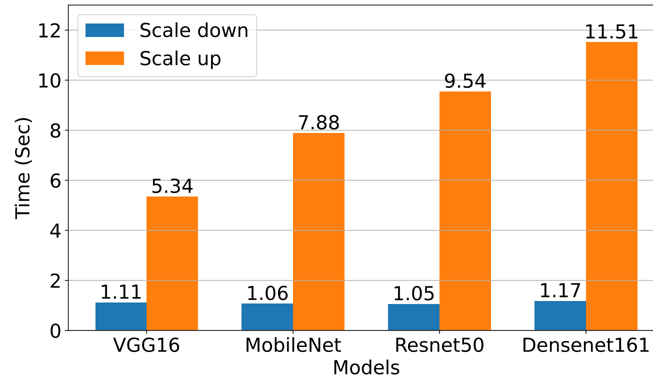
	N_1	N_2	N_3	N_4	N_5	N_6	N_7	...	...	N_n
J_1	0	0	1	0	0	0	1	0	0	0
J_2	0	0	0	0	0	0	0	0	1	1
...	1	0	0	1	1	0	0	0	0	0
J_4	0	1	0	0	0	1	0	1	0	0

The node-job map shows the allocation plan, and Table 3.1 demonstrates an example map of the allocation plan. The MILP optimizer takes the input and gives a new node-job map to the Allocator to do the reallocation. We give more details in subsection 3.1.3.

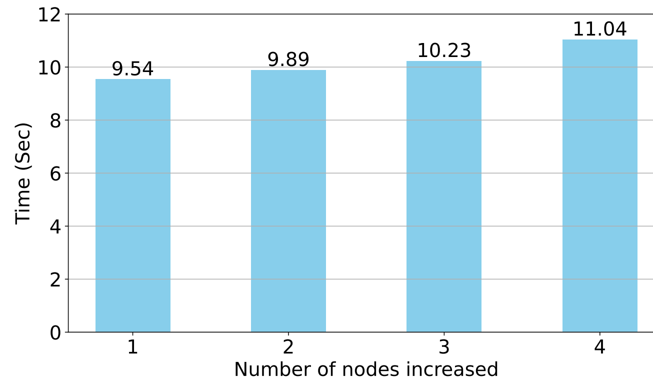
3.1.3 Job Profiling Advisor

In contrast to traditional profiling methods that necessitate dedicated resources, our online profiling process is integrated into the training process. This approach ensures the uninterrupted operation of worker processes during profiling. The strategic design of node adjustment sequences, as depicted in Figure 3.4, to avoid scale-up operations, effectively

minimizes additional overhead. Each job is equipped with a lightweight reporter (socket client), responsible for reporting job progress to the Job Monitor (socket server). This approach facilitates the automatic aggregation by the Job Monitor of the training process information that is then used for optimization purposes. Consequently, the `JPA` is enabled to make precise and timely adjustments, thereby maximizing resource utilization.



(a) 1-node scale-up and scale-down costs.



(b) Scale-up times vs. number of nodes: ResNet-50.

Figure 3.3: Rescaling overhead costs on Polaris A100 GPU nodes: (a) Time to scale up and down a single node, for different models; (b) Time to scale up different numbers of nodes, for ResNet-50 model.

We noted in subsection 2.1.3 the necessity for online profiling in order to permit accurate MILP solutions and to handle tasks for which profile information is not available before their execution. Here we shift focus to an in-depth examination of the design elements of `JPA`. In our proposed design the profiling function runs concurrently with jobs. Thus we want it to be:

Prompt, meaning that it processes profiling events rapidly so as to ensure efficient utilization of profiling information, and furthermore completes rapidly so as to minimize overhead and limit disruption to other tasks;

Fair, meaning that its design incorporates principles of fairness, and that in instances where job interruption is unavoidable, a Least Recently Used (LRU) strategy is employed to ensure equitable distribution of interruptions; and

Efficient, meaning that it prioritizes minimal disruption to other tasks, adhering to two key principles: (1) avoiding the interruption of multiple jobs simultaneously and (2) preventing the complete cessation of any single job.

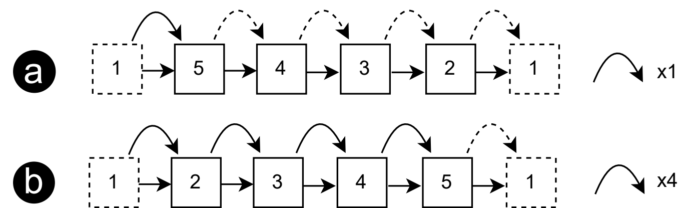


Figure 3.4: Inverse-order rescaling sequence. The solid curve represents scale-up and the dashed curve scale-down. JPA aims to minimize the number of scale-up operations in order to reduce overhead.

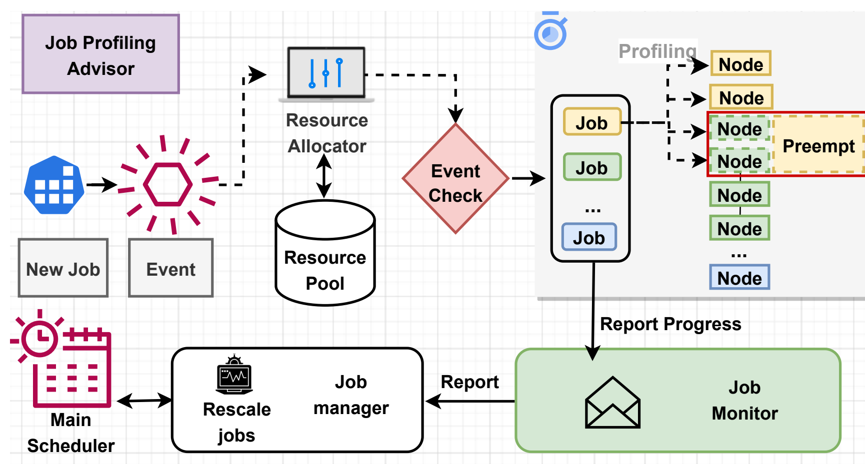


Figure 3.5: Online profiling process.

Accurate MILP requires that we know, or can rapidly determine, the time that will be required to run any training mini-task on any possible number of nodes. While obtaining this information may sound intractable, in practice the regular nature of DNN computations makes it feasible to obtain good estimates. As in FreeTrain, we assume a fixed per-node minibatch size (when training, we employ a learning rate scheduler to adjust learning rate according to the global batch size [67, 177]). We then need simply to measure the time per epoch for that minibatch size on different numbers of nodes, from a specified minimum to a specified maximum.

A useful optimization when performing those measurements derives from the observation that, as shown in Figure 3.3a, the cost of scaling up is consistently multiple times greater than that of scaling down. Furthermore, Figure 3.3b illustrates that the overhead incurred during scale-up remains relatively constant regardless of the number of nodes involved; even as the number of nodes increases, the increase in scale-up time is marginal. Consequently, in our profiling of the rescaling process, we should minimize the need for scaling up and prioritize scaling down wherever feasible. As an example, consider the two situations illustrated in Figure 3.4. If the initial number of nodes is 1 and the objective is to profile nodes 2, 3, 4, and 5, we may either: (a) scale up directly to 5 nodes and then scale down to 1, thereby gathering scalability data for all nodes using a single scale-up operation, or (b) incrementally scale up from 1 to 5, which requires four separate scale-up operations. The first approach is significantly more efficient than the first, since it requires only one scale-up operation.

The `JPA` architecture (Figure 3.5) resembles that of `MalleTrain` but with several distinctions: (1) `JPA` exclusively processes new job events, since only these require profiling; in contrast, the trainer instance accepts multiple events, as described in subsection 3.1.2. (2) The node adjustment in `JPA` is decided by our profiling algorithm instead of by the MILP

program. Users retain the discretion to decide whether their jobs undergo profiling. Upon receiving a profiling request from a user, a profiling event is triggered, which initiates a process whereby the Resource Allocator assesses the availability of sufficient resources for profiling. If resources are deemed adequate, a profiling job is started, temporarily preempting nodes from other jobs. Upon completion of profiling, the MILP process is engaged to make adjustments based on the newly collected information. The gathered scale information is then reported and recorded by the job manager, contributing to future optimization efforts.

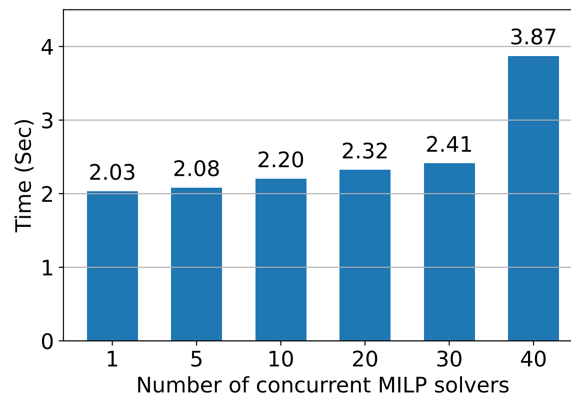


Figure 3.6: Average time taken for an example MILP computation as the numbers of concurrent MILP computations performed on a single 32-core head node scales from 1 to 40.

3.1.4 Cluster Configuration

MILP is an NP-hard problem and the cost of the MILP computation required to determine a mapping of jobs to idle nodes scales rapidly with the number of runnable jobs and available nodes. Thus, it can be preferable to partition a supercomputer into disjoint subsets and run multiple trainers in parallel, one per subset. This approach restricts the maximum number of nodes to which any one job can scale, but has the advantages of reducing delays due to training and of permitting different trainers to optimize for different metrics appropriate for different task types, such as computer vision models and language models.

With multiple trainers, the question arises of whether it is advisable from a performance perspective to run more than one on a single node. Our preliminary investigation into the effects of running multiple MILP processes concurrently on the same node revealed that the processing time begins to increase only when the number of concurrent trainers exceeds the number of cores, as illustrated in Figure 3.6. This suggests that deploying multiple trainers and running the associated MILP processes concurrently on a single head node can diminish overheads without adversely affecting the performance of standard jobs.

3.2 Evaluation and Discussion

We conducted an extensive experimental evaluation to validate the effectiveness and robustness of our framework with real logs of supercomputer clusters. We also validated `MalleTrain` on a small cluster in a real production environment.

3.2.1 Experiment Setup

We examined trace logs from two supercomputers listed in the TOP500 as of November 2023: Summit, ranked 7th, and Polaris, ranked 27th [22]. The Summit log spans 14 days from February 10 to February 24, 2021, while the Polaris log covers a 7-month duration, from January 1 to July 28, 2023.

Figure 3.7 depicts event traces from the Summit and Polaris supercomputers. We see that Polaris has more shorter gaps than Summit, with indeed over 50% of its event gaps being shorter than 10 seconds. A key factor contributing to this difference is Summit's

policy favoring large jobs. Such jobs generally have longer durations, leading to fewer but more extended resource occupations. Conversely, without a similar policy favoring large jobs, Polaris experiences more frequent, shorter gaps between events due to the prevalence of smaller jobs. However, because of the unavailability of idle node data for the Polaris cluster, we focus on Summit trace data in our log replay simulation evaluation. Figure 3.8, which shows idle nodes on Summit over a two-week period, shows that the number of idle nodes varies significantly over time.

While plugging `MalleTrain` into the batch scheduler of a real supercomputer would permit accurate evaluation in a real system, we would lose the ability to reproduce the same trace with different strategies, including the baseline allocation policy, for comparative research. Therefore, we instead generate representative traces and replay them on the real system for our experimental evaluation. In contrast to the simulation-based evaluation, experiments here do not rely on any performance modeling: they run the DNN training task on real supercomputer nodes.

A challenge for `MalleTrain` is to optimally utilize fragmented node×time resources to meet a user-specified metric (e.g., throughput in terms of samples processed per second, resource utilization/scaling efficiency). We synthesize traces that are independent and identically distributed with real traces from supercomputers. Figure 3.9 compares node idle gap lengths from real Summit scheduler logs vs. our synthetic traces. We see that the distribution of synthetic traces is close to those of the real logs, confirming the representativeness of our synthetic traces.

3.2.1.1 Workload

NASBench101 [174] is a neural architecture search (NAS) benchmark dataset created to permit systematic, reproducible, and accessible evaluation of NAS algorithms. It was introduced to address the challenges associated with the high computational cost of evaluating NAS algorithms, which traditionally require training thousands of neural network architectures from scratch to find the most efficient one for a given task. We conducted our experiment within the search space of NASBench101, which comprises 423,624 computationally unique neural architectures. The image size for our training is 224×224×3. We use randomly generated tensors instead of the real dataset to remove the potential I/O impact on our experiments. We note that our focus here is not on the accuracy of the models but rather on assessing throughput and scalability. Varieties of deep learning models that do the HPO tasks were also evaluated in the same context as the NAS workload; the models were randomly selected from models listed in Figure 3.12.

3.2.1.2 Testbed

We conducted experiments on a 32-node cluster in which each node is equipped with four A100 GPUs. The GPUs are interconnected via NVLink within each node, and nodes are connected via InfiniBand. The synthesized traces, as depicted in Figure 3.9, were instrumental in simulating the preemptive actions undertaken by the main scheduler.

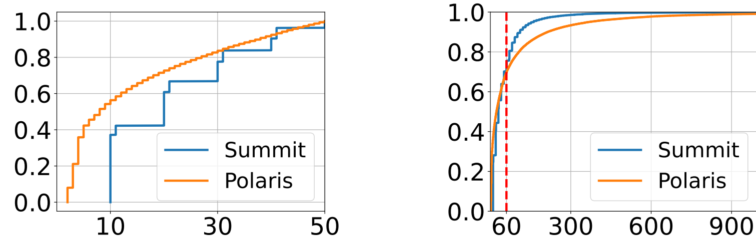


Figure 3.7: Cumulative histograms of idle gap counts on Summit and Polaris, for short gaps (0–50 secs: left) and longer gaps (0–3600 secs: right). Polaris has more shorter gaps (≤60 secs) while Summit has more gaps in the range from 60 to 600 secs.

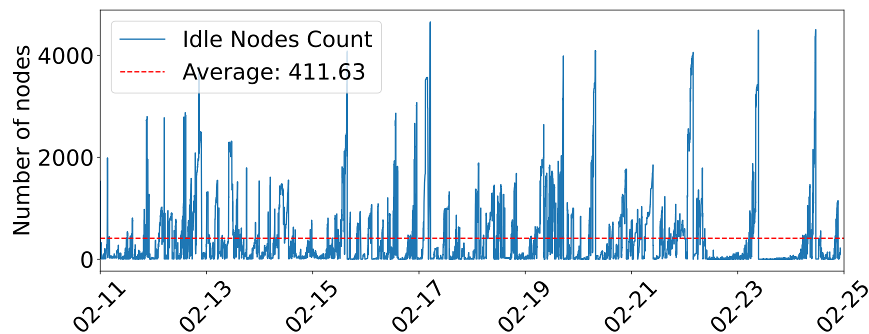


Figure 3.8: Idle nodes on Summit over two-week period.

3.2.2 Performance Evaluation

We conducted experiments to benchmark our system against the FreeTrain framework for preemptible resource allocation on HPC clusters. Our evaluation comprised NAS and HPO training workloads. Notably, the NAS workload exhibited more variability in training speed and scalability compared with HPO tasks.

Our primary metric for comparison was the overall training throughput of the system. We ran both frameworks under identical workloads to ensure a fair comparison. For the NAS model sampling process, we randomly selected models. To maintain consistency, we set the same seed value for both frameworks, ensuring that the sequence of model training remained identical across the experiments. We conducted the simulation with the two-week log and conducted the experiments for 12 hours with the synthetic trace. The average

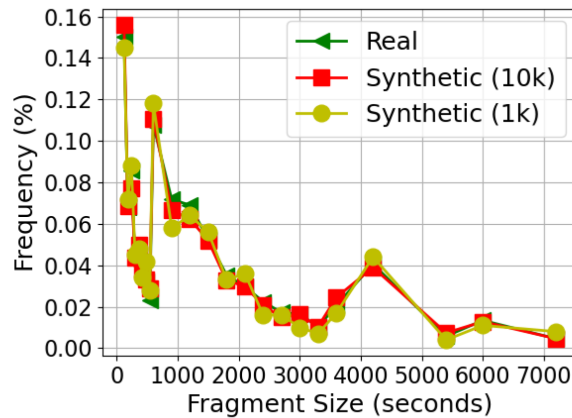
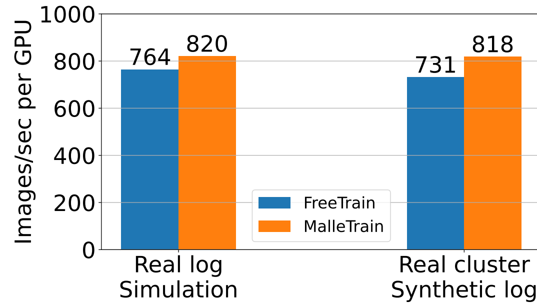


Figure 3.9: Comparison histogram of fragment length between real logs and synthetic. Synthetic (10k) shows the statistics for 10k fragments, and Synthetic (1k) shows the statistics for 1k fragments. The synthetic traces keep the same distribution as that of the real log.

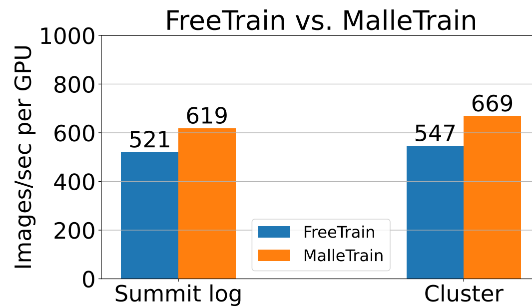
throughput is shown in Figure 3.10 with the NAS workload and HPO workload. We see that `MalleTrain` outperforms FreeTrain in various settings.

3.2.3 Topology Impact Analysis

The dynamic and randomly scattered nature of fragmented resources across the cluster raises concerns about potential declines in the overall performance of training jobs. To address these concerns, we conducted experiments on the Polaris cluster with the dragonfly network topology. Our study involved comparing the performance of nodes confined within a single dragonfly group versus those distributed across multiple dragonfly groups. Figure 3.11 shows that the physical distribution of nodes, whether scattered or closely situated, has minimal impact on NAS/HPO DNN training speed. Figure 3.12 indicates robust scalability of models even at the 32-node level, each node equipped with 4 NVIDIA A100 GPUs, encompassing a total of 128 A100 GPUs.

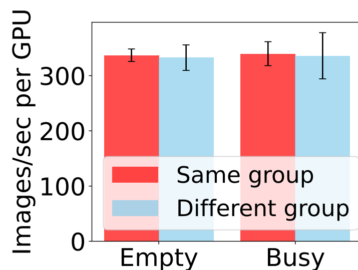


(a) Neural architecture search

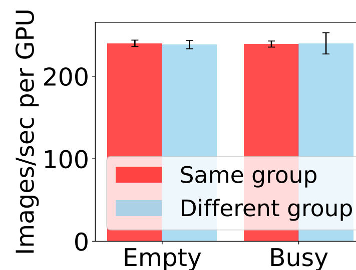


(b) Hyperparameter optimization

Figure 3.10: FreeTrain vs. `MalleTrain` performance for the NAS and HPO applications, as measured both with real logs on a simulator and synthetic logs on a real cluster.



(a) ResNet-50



(b) VGG-19

Figure 3.11: We analyzed training performance for sample `MalleTrain` jobs under four different scenarios: *Same Group, Empty* (where all nodes are located within the same Dragonfly group and the cabinet is empty), *Same Group, Busy* (where all nodes are within the same Dragonfly group but are collocated with other jobs), *Different Group, Empty* (where nodes are distributed across two Dragonfly groups with the two cabinets empty), and *Different Group, Busy* (where nodes are distributed across two Dragonfly groups and collocated with other jobs). The results demonstrate consistent training speeds for both models across all scenarios. The error bars for the *Different Group, Busy* scenario reveal higher variances in training speed, indicating fluctuations occur primarily in this scenario. However, the average training speed remains consistent despite these fluctuations.

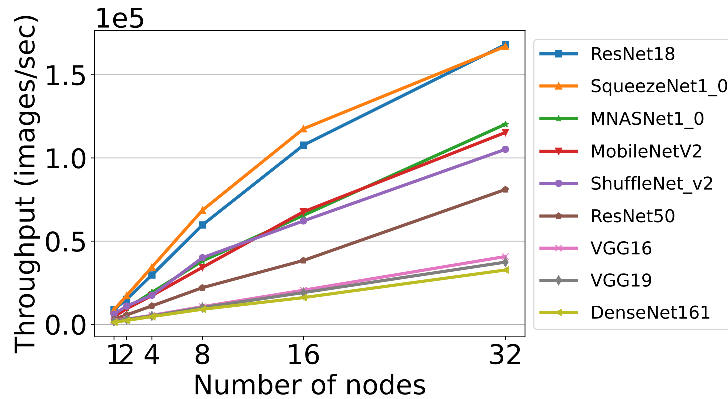


Figure 3.12: Trend analysis of model scalability on 32 Polaris nodes, each with 4 A100 GPUs.

The underlying reasons for these observations are multifaceted. First, leadership-class supercomputer clusters are typically outfitted with high-performance network devices. For instance, Polaris is equipped with the HPE Slingshot 11 interconnect, offering up to 200 Gb/s point-to-point bandwidth. Second, the networking infrastructure in these clusters is often highly overprovisioned, mitigating network contention among applications running on different nodes. Third, modern distributed deep learning frameworks, such as PyTorch [128] and Horovod [146], effectively overlap computing and communication tasks. This overlapping functionality reduces the network's impact on training speed, thereby diminishing the sensitivity to network conditions.

3.3 Related Work

We have already referred to the pioneering work of Liu et al. on FreeTrain [112], while noting also that certain assumptions and strict requirements make it fall short in the real production environment. FreeTrain heavily relies on users to provide accurate runtime information from training jobs, which increases the burden to the users, making it imprac-

tical for use. Indeed, in some widely used heuristic NAS/HPO algorithms, FreeTrain has to guess a configuration or provide information solely based on user experience; the inaccurate or out-of-date information might largely downgrade the overall performance of the MILP optimization algorithm. In contrast, `MalleTrain` integrates automatic profiling components into the process and doing the profiling automatically.

Pollux [133] is a resource-adaptive DNN training and scheduling framework designed to efficiently rearrange distributed deep learning processes, particularly in dynamic-resource environments such as shared clusters and cloud infrastructures. This framework employs Kubernetes for efficient scheduling, rescaling, and reconfiguring of job batch sizes and learning rates, thus maximizing training performance and optimizing resource utilization. Pollux operates on a fixed-size cluster, however, whereas `MalleTrain` can handle dynamically varying cluster sizes.

3.4 Summary

In this chapter, we have introduced `MalleTrain`, a system that we demonstrate can employ idle fragmented nodes on batch-scheduled HPC systems for large-scale DNN training. `MalleTrain` defines a workable architecture for efficient use of such idle nodes, and via its job profiling advisor, which efficiently gathers accurate job execution data at runtime with minimal interference to ongoing tasks, enables idle nodes to be employed efficiently even for dynamic workloads such as neural architecture search and hyperparameter optimization. Detailed performance studies involving both simulations and experiments validate the effectiveness of the approach and show that `MalleTrain` achieves >20% more training throughput than was reported, on the basis of simulation studies alone, for a precursor sys-

tem. `MalleTrain` thus opens up the feasibility of both improving the utilization of large HPC systems and increasing the resources delivered to DNN applications.

Moreover, the methodologies developed in this study have potential applications beyond their current scope. They could be adapted, for example, to infrastructure management tasks, such as scheduling in Kubernetes clusters and other cloud computing platforms.

CHAPTER 4
EXPLOITING EPHEMERAL SERVERLESS FUNCTIONS TO BUILD A COST-EFFECTIVE MEMORY CACHE

4.1 InfiniCache Design

InfiniCache has three components: an InfiniCache client library, a proxy, and a Lambda function runtime used to implement cache nodes[1]. As shown in Figure 4.1, an InfiniCache deployment consists of a cluster of Lambda cache nodes, which are logically partitioned and managed by multiple proxies. Each proxy orchestrates a *Lambda cache pool*. Applications interact with InfiniCache via a client library that is responsible for cache invalidation upon an overwrite and cache insertion upon a read miss assuming a read-only, write-through cache; the client library encodes and decodes the objects using erasure coding (EC) and interfaces with a proxy serving as a rendezvous that streams the EC-encoded object chunks between a client library and the Lambda nodes.

InfiniCache introduces a proxy primarily because a Lambda node cannot run in server mode due to banned inbound connections. Thus a client library has to rely on an intermediate server (the proxy) for accepting connection requests from Lambda nodes. In InfiniCache, the client library and proxy are logically separated as they have clearly partitioned functionality, but in deployment they can be physically co-located on the same machine. To enable data sharing across different Lambda cache pools, a client can communicate with any proxy (see Figure 4.1).

[1] We use Lambda cache node and Lambda function (runtime) interchangeably in different contexts.

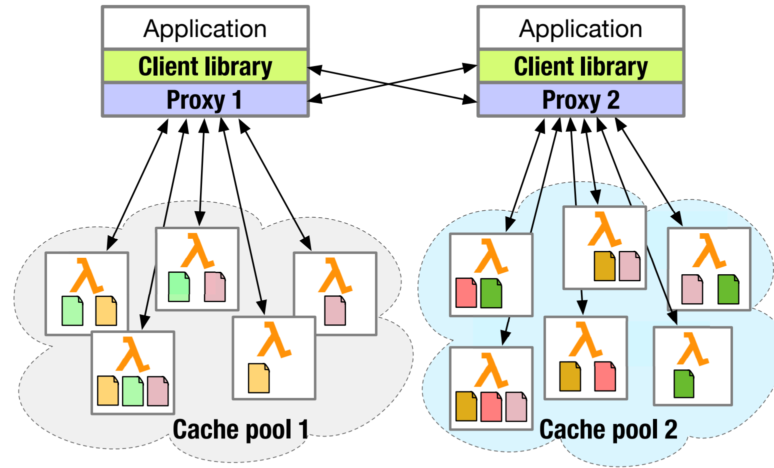


Figure 4.1: InfiniCache architecture overview. Icon denotes EC-encoded object chunks. Chunks with same color belong to the same object.

4.1.1 Client Library

InfiniCache's client library exposes to the application a clean set of `GET(key)` and `PUT(key, value)` APIs (see Figure 4.2). The client library is responsible for: (1) transparently handling object encoding/decoding using an embedded EC module, (2) load balancing the requests across a distributed set of proxies, and (3) determining where EC-encoded chunks are placed on a cluster of Lambda nodes.

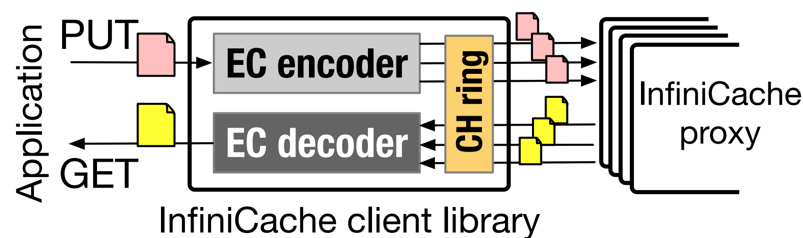


Figure 4.2: InfiniCache client library (CH: consistent hashing).

Erasure Coding Processing. In our initial design, we observed that adding EC processing to the proxy would stall the chunk streaming pipeline (section 4.1.2) and significantly impact the overall data transfer performance. Hence we made a design choice to move the

computation-heavy EC part from the proxy to the client library.

The PUT Path. Assume that we have a multi-proxy deployment in which each proxy manages a separate Lambda node pool with shared access among clients. For a PUT request, InfiniCache's client library first determines the destination proxy (and therefore its backing Lambda pool) by using a consistent hashing-based load balancing approach. The client library then encodes the object with a pre-configured EC code ($(d+p)$ using a Reed-Solomon (RS) code) and produces a number of object chunks, each with a unique identifier ID_{obj_chunk} (computed as a concatenation of the object key and the chunk's sequence number). To handle extremely large objects, InfiniCache can encode them with more aggressive EC code (e.g., (20 + 4)). Next, the client decides which Lambda nodes to store the chunks on by randomly generating a vector of non-repetitive ID_λ. Each encoded chunk with its piggybacked ¡$ID_{obj_chunk}, ID_\lambda$¿ is sent to the destination proxy, which streams the data to the destination Lambda nodes and remembers the locations in the Lambda pool where the chunks are cached.

The GET Path. A GET request is first sent to the proxy by using consistent hashing; the proxy then consults its mapping table, which records the chunk to Lambda node association and fetches the object chunks from the associated Lambda nodes (see section 4.1.2). Once the chunks arrive at the client, the client library decodes the chunks, reconstructs the original object, and returns the object to the application.

Eliminating Lambda Contention. Lambda functions are hosted by EC2 Virtual Machines (VMs). A single VM can host one or more functions. AWS seems to provision Lambda functions on the smallest possible number of VMs using a greedy binpacking heuristic [164]. This could cause severe network bandwidth contention if multiple network-

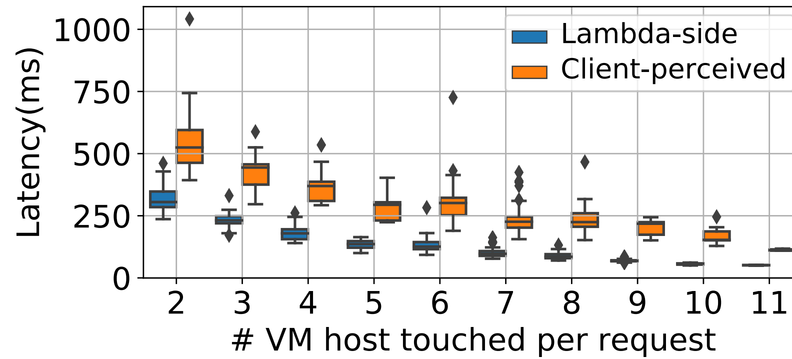


Figure 4.3: The box-and-whisker plot of latencies as a function of the number of VM hosts touched per request.

intensive Lambda functions get allocated on the same host VM.

We conduct an empirical study to verify this. In our study setup, each Lambda function has 256 MB memory. We use an RS code of (10 + 1) to split a 100 MB object into 10 data chunks and 1 parity chunk, and place each chunk on a Lambda node randomly selected from a fixed sized Lambda node pool. We measure the latency of `GET` requests by scaling-up the pool from 20 to 200 Lambda nodes. As a result, the number of host VMs that the 11-chunk object spans varies proportionally as the Lambda node pool scales up and down[2]. Figure 4.3 shows the latency distribution as a function of the number of underlying host VM touched per request. With a larger Lambda node pool (where the request is more likely to be spread across more host VMs), we observe a decreasing trend in the latency on the Lambda-side (the time that each Lambda node spends serving the chunk request) as well as the client-perceived (end-to-end) latencies.

These results stress the need to minimize resource contention among multiple Lambda functions sharing the same VM host. While over-provisioning a large Lambda node pool with many small Lambda functions would help to statistically reduce the chances of Lambda co-location, we find that using relatively bigger Lambda functions largely eliminates Lambda co-location. Lambda's VM hosts have approximately 3 GB memory. As such, if we use

[2]We run command `uname` in Lambda to get the underlying host VM's IP.

Lambda functions with ≥ 1.5 GB memory, every VM host is occupied exclusively by a single Lambda function, assuming InfiniCache's cache pool consists of Lambda functions with the same configuration[3].

4.1.2 Proxy

Each InfiniCache proxy (Figure 4.4) is responsible for: (1) managing a pool of Lambda nodes, and (2) streaming data between clients and the Lambda nodes. Each Lambda node proactively establishes a persistent TCP connection with its managing proxy.

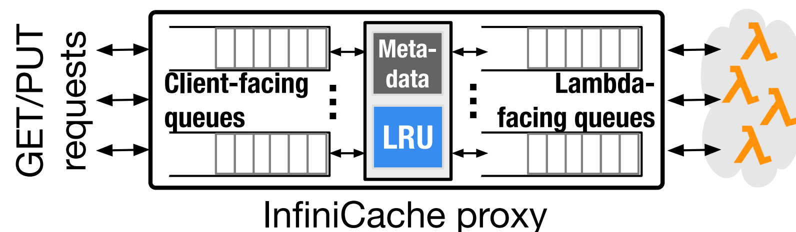


Figure 4.4: InfiniCache proxy.

Pool Management. Each proxy manages a pool of Lambda nodes, and also maintains the metadata to record the mapping between object chunks and Lambda nodes. To achieve fault tolerance, the proxy also serves as a coordinator to coordinate data migration and delta sync. Each proxy tracks the memory usage of every Lambda node in the pool. The proxy starts to evict objects as long as there is not enough free memory in the Lambda pool using a CLOCK based [61] LRU policy. The LRU module operates at the object granularity at the proxy. After the eviction process, the proxy updates the mapping metadata, and inserts the new data.

[3] AWS does not allow sharing Lambda-hosting VMs across tenants [35].

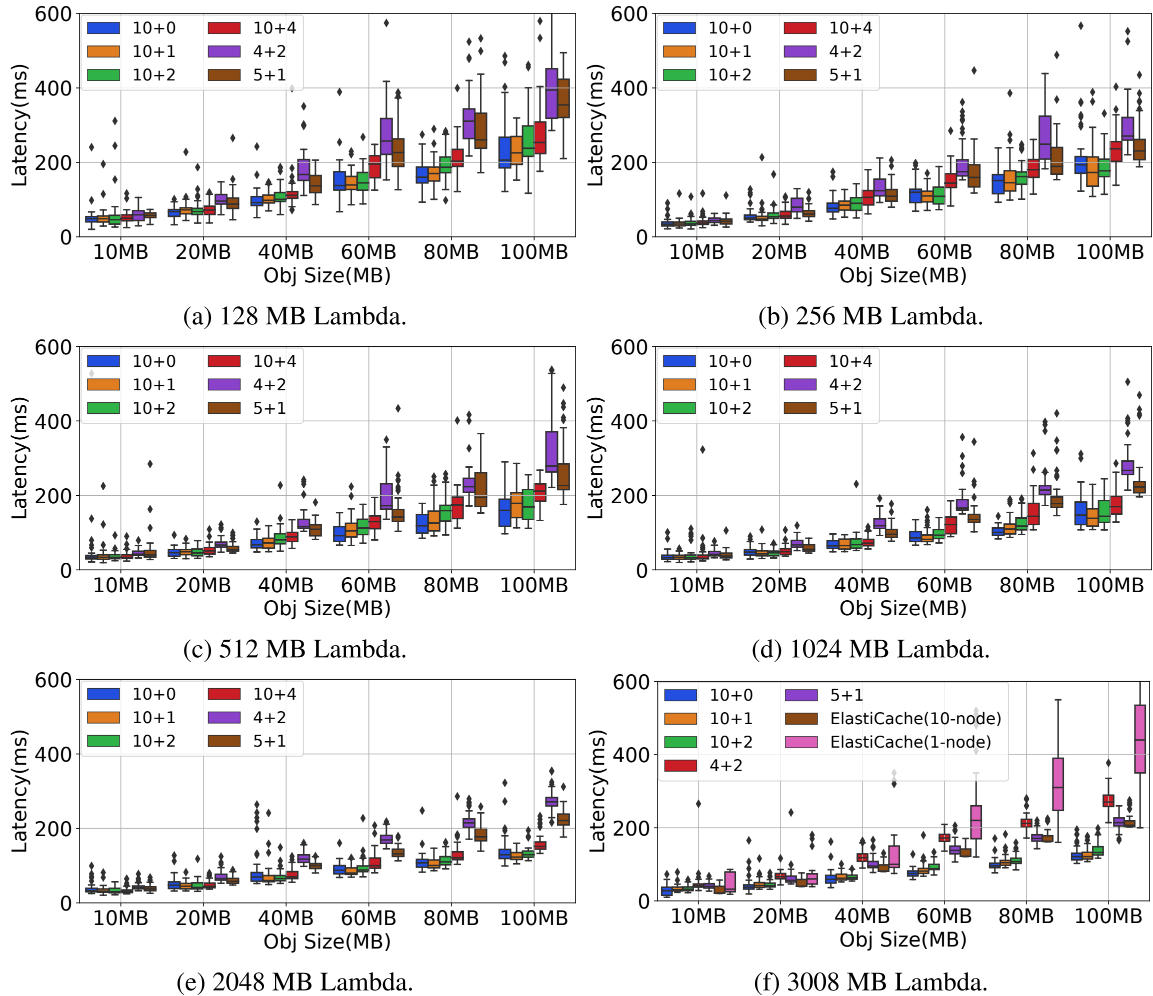
First-d based Parallel I/O. The proxy sends and receives object chunks in parallel by utilizing I/O parallelism to maximize network bandwidth utilization. To mitigate the Lambda straggler problem, the proxy directly streams the first d out of $(d+p)$ encoded object chunks to the client. Though accepting the first-d arrived chunks may likely result in an EC decoding process at the client library, as we show in section 4.2.1, the performance benefit of the optimization outweights the EC decoding overhead with reduced tail latency for `GET` requests.

4.2 Evaluation

In this section, we evaluate InfiniCache on AWS Lambda using microbenchmarks.

Implementation. We have implemented a prototype of InfiniCache using 5,340 lines of Go (460 LoC for the client library, 3,447 for the proxy, and 1,433 for the Lambda runtime). The EC module of the client library is implemented using the Golang reedsolomon lib [16], which uses Intel's AVX-512 for accelerating EC computation.

Setup. Our experiments use AWS Lambda functions with various configurations. Unless otherwise specified, we deploy the client (with InfiniCache's client library) and proxy on `c5n.4xlarge` EC2 VM instances. The Lambda functions are in the same Amazon Virtual Private Cloud (VPC) as the EC2 instances and are equipped with a 10 Gbps network connection. The Lambda functions' network bandwidth increases with its memory amount; we observed a throughput of 50–160 MBps (from the smallest memory amount of 128 MB to the largest memory amount of 3008 MB) between a `c5n.4xlarge` EC2 instance and a



(a) 128 MB Lambda.
(b) 256 MB Lambda.
(c) 512 MB Lambda.
(d) 1024 MB Lambda.
(e) 2048 MB Lambda.
(f) 3008 MB Lambda.

Figure 4.5: Microbenchmark performance.

Lambda function using `iperf3`.

4.2.1 Microbenchmark Performance

We first evaluate the performance of InfiniCache under synthetic `GET`-only workloads generated using a simple benchmark tool. With the microbenchmarking tests, we seek to understand how different configuration knobs impact InfiniCache's performance. The evaluated configuration knobs include: EC RS code (we compare (10 + 1), (10 + 2), (4 + 2),

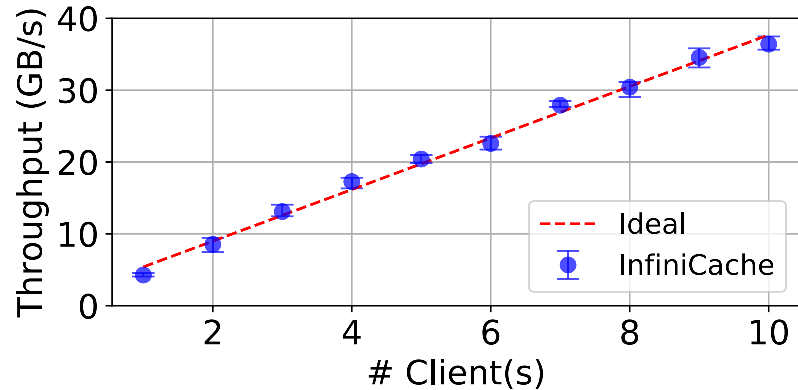


Figure 4.6: Scalability of InfiniCache.

(5 + 1), with a (10 + 0) baseline, which directly splits an object into 10 chunks without EC encoding/decoding), object sizes (10–100 MB), and the Lambda function's resource configurations (128–3008 MB).

Figure 4.5 shows the distributions of end-to-end request latencies seen under different configuration settings. Invoking a warm Lambda function takes about 13 ms on average (with the Go AWS SDK API), which is included in the end-to-end latency results. We observe that the (10 + 1) code performs best compared to other RS code configurations. This is due to two reasons. First, (10 + 1) results in a maximum I/O parallelism factor of 10 (first-k parallel I/O is described in section 4.1.2), and second, it keeps the EC decoding overhead at a minimum (the higher the number of parity chunks, the longer it takes for RS to decode). The caveat of using (10 + 1) is that it trades off fault tolerance for better performance.

Another observation is that the $(10+0)$ case does not seem to lead to a better performance than that of $(10+1)$ and in several cases even sees higher tail latencies. This is due to the fact that $(10+0)$ suffers from Lambda straggler issues, which outweighs the performance gained by fully eliminating the EC decoding overhead. In contrast, $(10+1)$'s first-d approach adds

redundancy and this request-level redundancy helps mitigate the impact of stragglers.

A Lambda function's resource configuration has a great impact on InfiniCache's latency. For example, $(10+1)$ achieves latencies in the range of 110–290 ms (Figure 4.5c) with 512 MB Lambda functions for objects of 100 MB, whereas with 2048 MB Lambda functions, latencies improve to 100–160 ms (Figure 4.5e). In addition, latency improvement hits a plateau for Lambda functions equipped with more than 1024 MB memory because larger Lambda functions eliminate the network bottleneck for large chunk transfers.

To compare InfiniCache with an existing solution, we choose ElastiCache (Redis) and deploy it in two modes, a `1-node` deployment using a `cache.r5.8xlarge` instance, and a scale-out `10-node` deployment using `cache.r5.xlarge` instances. As shown in Figure 4.5f, InfiniCache outperforms the `1-node` ElastiCache for all object sizes, as Redis is single-threaded and cannot handle concurrent large I/Os as efficiently. For larger object sizes, InfiniCache with (10 + 1) and (10 + 2) consistently achieves lower latencies compared to the `10-node` ElastiCache, thanks to InfiniCache's first-d based data streaming optimization. These results show that InfiniCache's performance is competitive as an IMOC.

Scalability. In this test, we setup a multi-client deployment to simulate a realistic use case in which a tenant has multiple microservices that concurrently read from and write to InfiniCache. To do so, we vary the number of clients from 1 to 10. We also deploy a 5-proxy cluster where each proxy manages a 50-node Lambda pool (and each Lambda function has 1024 MB memory). Each client uses consistent hashing to talk to different proxies for shared data access (see Figure 4.1).

Figure 4.6 shows the throughput in terms of GB/s. We observe that InfiniCache's through-

put scales linearly as the number of clients increases. Ideally, InfiniCache can scale linearly as long as more Lambda nodes are available for serving GET requests.

4.2.2 Summary

With web applications becoming increasingly storage-intensive, it is imperative to revisit the design of in-memory object caching in order to efficiently deal with both small and large objects. We have presented a novel in-memory object caching solution that achieves high cost effectiveness and good availability for large object caching by building InfiniCache on top of a popular serverless computing platform (AWS Lambda). For the first time in the literature, InfiniCache enables request-driven pay-per-use elasticity at the cloud storage level with a serverless architecture. InfiniCache does this by synthesizing a series of techniques including erasure coding and a delta-sync-based data backup scheme. Being serverless-aware, InfiniCache intelligently orchestrates ephemeral cloud functions and improves cost effectiveness.

CHAPTER 5
ENABLING SCALABLE AND ADAPTIVE MACHINE LEARNING TRAINING VIA SERVERLESS COMPUTING ON PUBLIC CLOUD

5.1 SMLT Design

In lieu of the aforementioned shortcomings to support dynamic resource requirements, user-centric deployment goals and large-scale ML models, we propose SMLT. In this section, we explain the rationale behind the SMLT design, present our goals, and then describe the building blocks to achieve them.

Our general aim is to design a generic framework that provides the easy-to-use functionality for ML practitioners to design and train their ML models efficiently and economically on a public cloud serverless platform. The framework should allow ML practitioners to focus on business logic while at the same time truly realizing the transparency and scalability promises of serverless computing. In particular, we design our building blocks to target three main goals.

- We aim to offer an overarching view of dynamic ML workflows to enable adaptive and efficient serverless scaling on public cloud platform.
- We aim to support user-centric deployment goals (e.g., training deadlines and budget limits) for running ML workflows on public serverless platforms.
- We aim to achieve scalability by enabling the handling of large and sophisticated ML

workflows while abstracting out the limitations of the underlying public serverless platforms.

5.1.1 Overarching View and Dynamic Adaptation

The underlying primitives in existing serverless platforms can support individual ML training tasks well. However, for modern dynamic ML workflows, such as dynamic batching [45] and NAS [56], today's serverless systems often fall short. The fundamental reason is the lack of a component that maintains an overarching view of the training dynamics of ML tasks. For example, when the batch size or model size changes over the training process, it may inevitably change the scalability and resource demands of the training task. Not adjusting the underlying serverless resources accordingly may cause performance and/or cost issues. The stateless nature of public serverless platforms, and hence their inability to carry over the training dynamics across function invocations, creates a challenge for supporting such an adaptation mechanism.

To solve this challenge, we propose a *training dynamics aware* design to enable an overarching view of ML workflows. To keep track of the training progress, we propose a monitoring component (called *Task Scheduler* in Section 5.2.1) to collect the training information (e.g., time taken to complete one iteration, batch size changes), and maintain it across function invocations. The task scheduler continuously monitors for changes in training information, and upon detecting change, activates an optimizer to determine the new resource allocation to meet the optimization targets. The scheduler then allocates resources according to the newly optimized resource decisions, such that the training can continue with new resources (e.g., number of workers, memory configuration).

5.1.2 User-centric Deployment and Execution

To guide the above process, we employ a user-centric deployment and execution approach. ML practitioners often have different goals in ML model training, such as meeting deadlines or staying within a monetary budget. These desired goals are not supported by today's MLaaS platforms (see Section 2.3). The serverless computing paradigm creates an unprecedented opportunity to provide these goals because of its advantages in scaling and resource management. By embracing serverless computing, `SMLT` alleviates the burden of reserving and scheduling resources from ML practitioners, and allows them to focus on their ML model development, while guaranteeing their time and budget constraints are met.

Definitions and Terminology. Our approach takes a user's requirements as input to optimize the deployment and execution of ML tasks. Without loss of generality, we use dynamic batching as an example to illustrate our approach. Let $\mathcal{B} = \{b_1, b_2, \cdots, b_n\}$ denote the batch scheduler, where b_i denotes the batch size at the ith epoch. Let $C = \{c_1, c_2, \cdots, c_n\}$ denote the system configuration across epochs as a tuple of scale-out factor (number of workers c_i^{worker}) and scale-up factor (memory size c_i^{memory}), i.e., $c_i = \langle c_i^{\text{worker}}, c_i^{\text{memory}} \rangle$. Let $T_{\mathcal{B}}(C)$ denote the training time using configuration C for a given batch scheduler $\mathcal{B}$, and $S_{\mathcal{B}}(C)$ denote the monetary cost using configuration C for a given batch scheduler $\mathcal{B}$. In addition, let T_{max} denote the user-specified training deadline and S_{max} denote the user-specified monetary budget for training.

Example Scenario. We give an example scenario where a user wants to meet the training deadline while minimizing the cost. This scenario can be modeled as an optimization

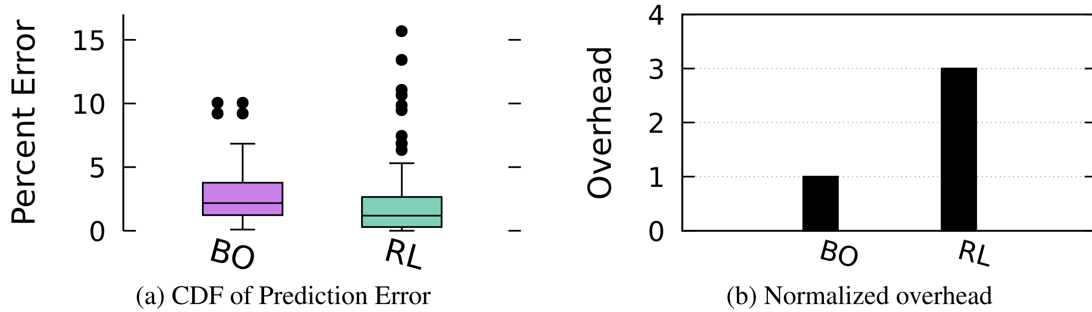
problem as follows:

$$\begin{aligned} \text{minimize} \quad & S_{\mathcal{B}}(C) \\ \text{subject to} \quad & T_{\mathcal{B}}(C) \leq T_{max}. \end{aligned} \tag{5.1}$$

To obtain the training cost $S_{\mathcal{B}_j}(C_i)$ of a particular deployment configuration C_i under a given batch scheduler $\mathcal{B}_j$, we need to first profile the training time per iteration (including the training throughput and communication time), and then use the cloud cost model to calculate the monetary cost. However, to exhaustively search the training cost of all possible deployment configurations to identify the minimum cost is prohibitively expensive as we need $n(C)$ number of profiling experiments. Therefore, we need an efficient optimization algorithm to explore the search space to reduce the optimization overhead. While there are plenty of optimization algorithms in the literature which could potentially be adopted here, we propose a customized Bayesian optimizer, inspired by [173], to efficiently solve this optimization problem.

There are other example scenarios with user-centric goals. For example, users may want to minimize training time with a budget, or users may simply want to finish training as fast as possible.

Bayesian Optimization. There are a few classic optimizers (e.g., Bayesian optimization and reinforcement learning) to solve the aforementioned optimization problems. We compare their performance in terms of accuracy and training overhead in Figure 5.1. We use the reinforcement learning model architecture proposed in [136] . It can be observed that, for the same prediction accuracy demonstrated in Figure 5.1a, reinforcement learning incurs 3× overhead compared to Bayesian optimization. In light of this result, we choose Bayesian optimization over reinforcement learning [44] to speed up the optimization process because it is much lighter-weight, and importantly, does not require additional training. This makes



(a) CDF of Prediction Error

(b) Normalized overhead

Figure 5.1: Comparison of Bayesian optimization and reinforcement learning in terms of accuracy and training overhead.

it more suitable for our goals in improving training speed and increasing cost-effectiveness.

The Bayesian Optimizer searches for the optimal deployment by profiling the throughput of the system under randomly chosen configurations. Based on the profiled throughput results, the Bayesian optimizer estimates the most beneficial configuration to profile next. This search is done in an iterative manner until the expected improvement is small enough or the predefined max number of iterations is reached, thus finding the optimal configuration. In doing so, `SMLT` exploits the serverless paradigm and its 'pay-as-you-go' model with fine-grained resource allocations. `SMLT`'s approach differs from the approach described in [173], whereby the Bayesian algorithm can run only once before the start of the ML task due to its high monetary cost associated with profiling in a VM-based cloud infrastructure, thus leaving little budget for the actual training tasks.

In its iterative configuration search, `SMLT` uses a two-dimensional search space where the worker memory size varies from 128 MB to 10 GB with an increment of 128 MB [13] and the number of workers varies depending on the model size and training parameters. The cost function for a particular deployment configuration is set as C_i, and this cost function is specified as part of user requirements like the example scenarios aforementioned.

We employ the widely-used Gaussian Process Regression [62] to calculate the posterior distribution. For the acquisition function, we use the Estimation Improvement (EI) [125] since it requires no hyperparameter tuning. Specifically, our EI is defined as: $EI(C_i) = (y_{max} - \mu(C_i))\beta(\gamma(C_i)) + \delta(C_i)\theta(\gamma(C_i))$, where C_i is a deployment configuration with m memory sizes and n number of workers, and y_{max} is the current lowest value from all explored tuples. μ and δ are the predictive mean and standard deviation functions, with the β and θ being the predictive cumulative distribution function of standard normal and the probability density function of standard normal, respectively.

Based on the output of the Bayesian optimizer, our task scheduler deploys the training task accordingly with the optimized number of workers (scale-out) and the optimized worker memory configuration (scale-up). It is worth noting that, different stages of a training task may have different performance and resource requirements (Section 2.3). Therefore, our optimizer is designed to consider these dynamic resource requirements during training to improve the overall performance and optimize the resource usage during training. These optimizations are made possible by our user-centric deployment module and the on-demand scalability, as well as the pay-as-you-go billing model provided by serverless computing; thus, enabling us to provide cost-efficient solutions that honor user-specified training deadline and budget requirements.

Altogether, `SMLT` provides significant advantages over existing MLaaS platforms and other serverless ML training frameworks, and provides user-centric deployments and executions in a fully-automated, scalable, and adaptive fashion. As a result, `SMLT` not only improves training performance in a public serverless environment while meeting user requirements, but more importantly, allows users to focus on the design and logic of ML algorithms, thus enhancing their productivity.

5.1.3 End-to-end Scalability

Using serverless primitives to design an ML training platform presents unique challenges for scalability. Here, we briefly describe these challenges and our approaches to address them. We will elaborate upon the end-to-end design of `SMLT` in Section 5.2.

Challenge 1: Communication Overhead. Serverless frameworks are designed for stateless, short-duration compute tasks. For example, serverless function instances such as AWS Lambda do not allow inbound connections [158]. More specifically, the two unique characteristics of a serverless platform are the stateless nature of a serverless function and the limited communication performance across functions. These characteristics force serverless ML platform designers to utilize an external storage for model synchronization. Hence, the communication load/pattern generated by the chosen model synchronization scheme as well as the choice of the storage service play a crucial role. If not chosen properly, the communication overhead can easily overshadow the gains achieved via splitting computation across a large number of serverless workers, as demonstrated in Figures 2.4 and 2.5.

Approach: Hybrid Storage Enabled Hierarchical Model Synchronization. We observed from our experimentation that a hybrid storage service combined with a hierarchical synchronization mechanism scales well for large ML models. Even though the synchronization is built atop of the commonly used scatter reduce approach (which is also adopted by LambdaML [80]), our design considers the unique system challenges of public cloud serverless: 1) missing a fast shared intermediate storage for storing intermediate data in serverless function, which is critical for the model aggregation since the gradient partitions need to be stored and retrieved quickly; 2) limited network bandwidth of individual serverless function, which is too slow for transferring the gradients between serverless function

and shared intermediate storage.

There is a large amount of data generated and used in ML tasks. Such data falls broadly into two types according to their sensitivity to latency. 1) The access to the first type of data is latency sensitive, e.g., the metadata containing the gradient-worker mapping information, the gradients produced during each training iteration, etc. For this type of data, we employ a fast storage medium to satisfy the latency-sensitive demands of the synchronization scheme. In particular, an in-memory key-value store (e.g., Redis) can be used as a *parameter store* to store this type of data. 2) The access to the second type of data is much more infrequent. For example, the training code and dataset are accessed only few times during an epoch or after every epoch. To strike a balance between performance and cost, we use a cloud-based object store (e.g., AWS S3). We present the details of our hybrid storage in Section 5.2.3.

Figure 5.2 presents our hierarchical synchronization mechanism. After each training iteration, the hierarchical synchronization mechanism takes the model gradients generated by each worker as input. The *shard generator*, residing in each of the n workers, divides the input gradients into m equal-sized shards ❶. These shards are uploaded to the *parameter store* which acts as a communication intermediary between the stateless serverless workers ❷. Each serverless worker also acts as a shard aggregator. Each shard aggregator is responsible for downloading and aggregating its assigned shards generated by all workers ❸. For simplicity, we assume that n equals m.[1] As a result, the 'shard aggregator 1' in Figure 5.2 is responsible for aggregating the first shard from all workers to perform a mean operation. The resulting value, *aggregated shard*, is then uploaded to the parameter store by each shard aggregator ❹. Finally, the *global aggregator* residing in each worker down-

[1] If m is greater than n, then each worker is responsible for aggregating multiple shards. Choosing m less than n will cause some workers to be idle during aggregation, which will affect performance.

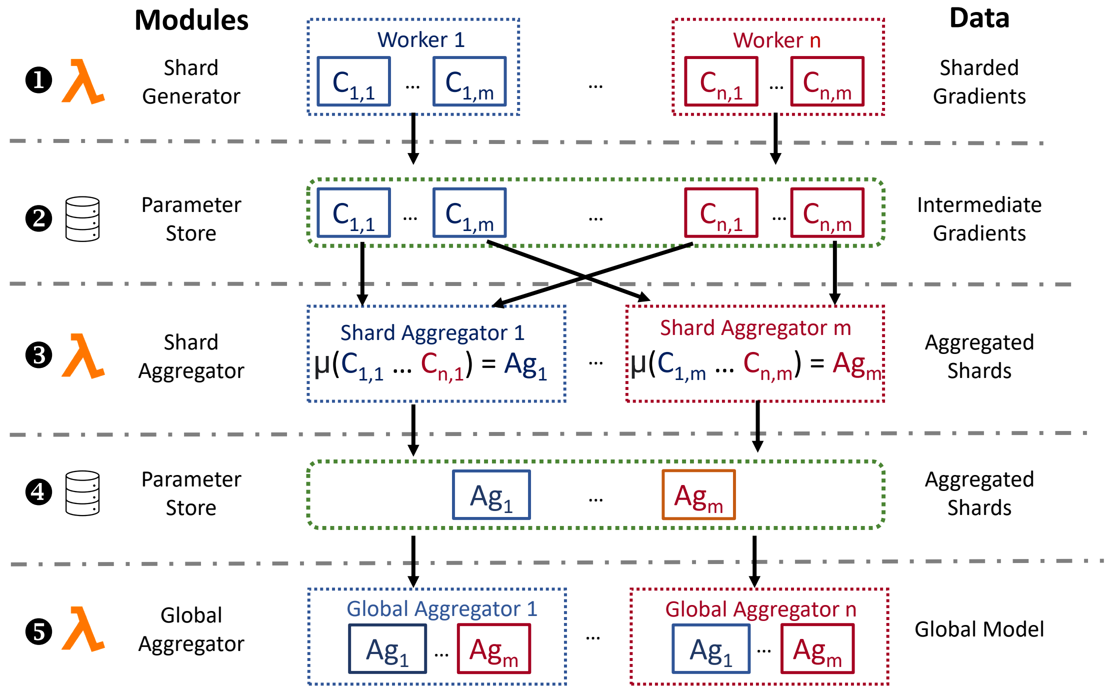


Figure 5.2: Hierarchical model synchronization mechanism.

loads all the aggregated shards, and reconstructs the updated model for the next iteration ❺. We describe in detail when and how this scheme is used in Section 5.2.2.

Challenge 2: Public serverless Platform Quirks. Serverless platforms typically limit the maximum duration of each executing function (e.g., 15 minutes in AWS Lambda [2]). Getting around this limitation as well as recovering from faults during function execution require keeping track of function states at regular intervals and restoring state during a restart. In addition, each function invocation may have non-trivial initialization overheads, making it important to minimize the number of restarts a function encounters. For example, initialization overheads could arise due to model loading within the function code itself. Other overheads include anomalous serverless platform behavior (which we encountered during our experiments on AWS Lambda), such as high invocation delays while invoking functions asynchronously and while invoking via the 'Map' state in AWS Step Functions [4]. Section 5.2.1 presents additional details.

Table 5.1
MODULES OF SMLT AND THEIR FUNCTIONALITIES.

Module	Submodule	Functionality
① End Client	(1a) Artifact Manager	Uploads user's training code and training data to the cloud-based object store (3a).
	(1b) Resource Manager	Allocates resources for training workers ② based on user-centric requirements (e.g., number of workers, and memory per worker).
	(1c) Task Scheduler	Invokes workers ② for training, keeps track of training progress, and restarts failed workers when necessary.
② Worker	(2a) Data Iterator	Loads training data to the local storage of the function from the object store (3a).
	(2b) Minibatch Buffer	Keeps track of the training data for every iteration used by the trainer (2c).
	(2c) Trainer	Runs the user-defined training code using the training data from the minibatch buffer (2b).
	(2d) Hierarchical Aggregator	Updates user model parameters after every iteration through the parameter store (3b).
③ Storage	(3a) Object Store	Stores training code and training data.
	(3b) Parameter Store	Stores intermediate training parameters of all workers ② after every iteration.

Approach: Task Scheduler. This challenge motivates us to create a coordination mechanism to provide the maximum flexibility and public cloud serverless platform independence, while invoking and monitoring function instances. To enable this coordination, we use a component called *task scheduler* in our approach. The task scheduler monitors the progress of each function instance, enables function instances to run for the maximum allowed execution duration, and performs the checkpointing for restarting functions (either due to faults or due to execution duration limits). We detail the task scheduler in Section 5.2.1.

5.2 SMLT Framework

In this section, we describe the details of the SMLT framework. Broadly speaking, SMLT adopts an architecture similar to a parameter server based distributed training using a fleet of stateless serverless functions. Although some existing frameworks [44, 80, 161] use a similar architecture, SMLT additionally offers the capability of intelligently and dynamically determining resource allocation (i.e., number of workers, memory sizes) that enables efficient and user-centric training of the state-of-the-art ML models with billions of learning parameters.

Table 5.1 gives an overview of SMLT, which consists of three main system modules. The *end client* is responsible for code deployment, resource allocation based on user-centric deployment and execution configurations, and ML task scheduling. The *workers* are mainly responsible for training and updating the global model. In *storage*, following our hybrid approach, the parameter store maintains the frequently-accessed intermediate model updates between workers, and the object store maintains the infrequently-accessed training code and data. Figure 5.3 shows the interactions between these different modules.

5.2.1 End Client

The end client provides an interface for users to interact with the cloud infrastructure. In particular, the end client is responsible for three tasks: 1) ML model and code deployment, 2) resource management which configures a deployment (e.g., number of workers, per-worker memory size) based on user-centric requirements and optimizations, and 3) scheduling tasks during training. Each user has her own end client, which can be deployed

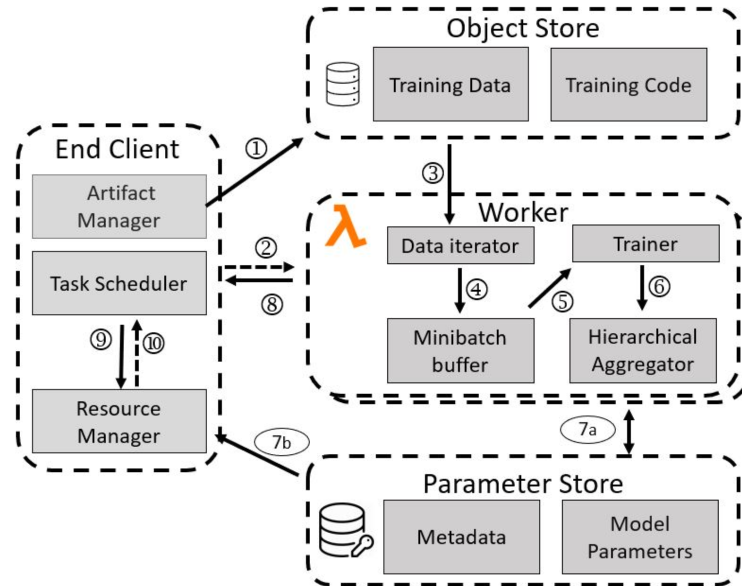


Figure 5.3: Workflow of `SMLT`. Solid lines indicate data transfer, and dashed lines indicate control signals.

either on a regular VM in the cloud or on a user's local machine.

The end client has three submodules: artifact manager, resource manager, and task scheduler.

Artifact Manager. The artifact manager is responsible for packaging the training code and data supplied by the user as input to `SMLT`, and uploading them to the object store (Step ① in Figure 5.3).

Resource Manager. The resource manager uses the training configuration (e.g., batch size) along with user-centric deployment requirements as input, and will dynamically adapt the resources for optimal training (Step ⑨).[2] Specifically, the resource manager controls the number of workers involved in a training phase, as well as the memory allocated to each worker. The appropriate values in different phases of the training are provided by

[2]Except for the initial configuration, where the training resources are arbitrarily assigned within the allowed bounds.

the Bayesian optimizer according to user-centric goals and the training progress (see Section 5.1.2), and then shared with the task scheduler (Step ⑩). The user can also override these values manually. Note that, commercial public serverless platforms typically provide a single control parameter for resources. For example, in AWS Lambda, other resources (e.g., number of computing cores, network bandwidth) are proportionally assigned by the allocated memory of a function instance.

The optimal training depends on the user's training requirements, training dataset, model size, and the cloud infrastructure's profile. `SMLT` is capable of optimizing the resources during the hyper-parameter search, dynamic batching and online learning for speed and cost-effectiveness. The optimization and adaptation of resources are triggered every time there is a change in training configuration obtained from the worker output (e.g., batch size or hyper-parameter change).

Task Scheduler. After the resources are determined by the resource manager, the task scheduler starts the training workers (Step ② in Figure 5.3) and will track their progress (Step ⑧). When the workers finish an iteration, their results are collected and fed back to the resource manager to be used in the Bayesian optimizer (Steps ⑧ and ⑨).

Tracking the progress of the workers by a separate task scheduler is important for three reasons: First, due to the stateless nature of the serverless workers, the used ML model needs to be obtained, and the ML framework used in the training code needs to be initialized by the worker. This initialization overhead depends on the model size and the specific ML framework, and can take multiple seconds (e.g., 4 seconds for Resnet-18 on Tensorflow) that quickly accumulate with function restarts. Therefore, a separate task scheduler can maximize the utility of each training worker by running it close to the limit of the function execution duration enforced by the underlying public cloud serverless platform (e.g.,

15 minutes on AWS Lambda) and across multiple iterations; therefore, the initialization overheads are amortized. After a worker is stopped, the task scheduler ensures that a new one is started and continues from the last iteration checkpoint.

The second reason for tracking the progress of each worker by a task scheduler is fault tolerance. A worker failure can derail the progress of the training. The task scheduler, with the help of the resource manager, detects failures by observing each worker's output (Step (7b)): The resource manager checks the output of each worker to determine whether a failure occurred during the training iteration. A successful update of the gradient parameters by a worker sets a flag in the output. The lack of this flag signals a worker failure, causing the task scheduler to restart it.

Finally, there may be hidden overheads specific to the underlying public serverless platforms. For example, when we experimented with serverless functions invoking other serverless functions asynchronously in AWS Lambda as utilized in LambdaML [80], we observed undocumented invocation delays for the new functions. Another example of such a hidden overhead is the limit on the concurrency of a function within the 'Map' state of a state machine in AWS Step Functions, even when the concurrency is set to be 'infinite' [12].[3] A separate and independent task scheduler in `SMLT` overcomes these limitations by treating every function invocation as a separate invocation and tracking them accordingly, and forcing the serverless platform to do the same.

[3]Since the writing of this chapter, the documentation has been updated to reflect that it is not guaranteed to achieve a requested concurrency.

5.2.2 Serverless Worker

The main responsibility of each SMLT worker is threefold: 1) obtain the training data, 2) run the training code, and 3) update the model. Each worker consists of the following four submodules that handle these tasks.

Data Iterator. Due to the limited storage space within serverless functions, similar to the existing work [44, 161], SMLT stores the training data in external cloud storage like AWS S3 or EFS. The data iterator submodule within each worker is responsible for fetching the appropriate subset of the training data from the external storage at the beginning of every training epoch and storing it within the worker's local disk storage (Step ③ in Figure 5.3). Furthermore, the data iterator also tracks which training data points have been processed by a worker within an epoch, in case the worker needs to resume training after a restart (due to either a failure or an execution time limit).

Minibatch Buffer. The minibatch buffer is responsible for loading a minibatch of training data from the local storage to the memory of the worker during each training iteration (Step ④). The minibatch size varies based on the number of workers and the global batch size during the training.

Trainer: The trainer submodule runs the user-defined training code over the minibatch of the training data (Step ⑤). The training code carries out the forward and backward passes to generate the gradients for the current iteration. These gradients are then aggregated with the gradients from other workers via SMLT's hierarchical model synchronization (Step ⑥).

Hierarchical Aggregator. The hierarchical aggregator submodule at each worker is responsible for synchronizing the gradients among different workers. As described in Section

5.1.3, the hierarchical aggregator at each worker first divides the gradients produced by the worker into shards, and uploads them to the parameter store (e.g., Redis). Each worker downloads the shards it is responsible for from the parameter store, aggregates them into a single shard, and uploads the aggregated shard back to the parameter store. Finally, the aggregator at each worker downloads all aggregated shards and produces the updated model for the next iteration (Step (7a)).

5.2.3 Hybrid Storage

Due to the restriction on inter-communication between Lambda functions, a shared storage solution is required. The choice of the storage system plays a crucial role because it is the main communication mechanism among serverless workers. In SMLT, we employ a hybrid approach that uses a cloud object store, e.g., AWS S3, for infrequently accessed data (few times or after every epoch), and an in-memory key-value store, e.g., Redis, for frequently accessed data (after each iteration). This hybrid approach matches the training phase to a cost-effective object store for bulk data access, while at the same time matching the model update phase to a fast in-memory key-value store for frequent data access.

Object Store. The object store hosts the training code and training data provided by the user and uploaded by the end client. The training code uploaded by the end client is accessed a few times at most (e.g., at the beginning of training, or after restarts from failures). Similarly, training data, which is accessed relatively infrequently after every epoch, is also hosted in the object store. Using the object store provides elasticity and cost efficiency for storing such types of infrequently-accessed data.

Parameter Store. In contrast, workers access gradient data much more frequently (i.e., after every iteration). During the hierarchical model synchronization, the speed of access to the updated and sharded gradients becomes critical for training speed. We use an in-memory key-value store (e.g., Redis) for this purpose. To avoid the extra cost of running the parameter store unnecessarily during the entire training, we use the light-weight containers from a cloud service (e.g., AWS Fargate, ECS), and keep them only alive during the model synchronization phase.

5.3 Evaluation

In this section, we evaluate `SMLT`'s capability to enable scalable and adaptive ML training via serverless computing on public cloud. We first demonstrate the effectiveness of the hierarchical model synchronization. Next, we present two different user-centric deployment scenarios. We then validate the performance of `SMLT` for dynamic batching and online learning. Finally, we show how `SMLT` can alleviate the burden of dynamic resource provisioning from ML engineers and practitioners during model design and building via an autonomic technique like neural architecture search (NAS).

5.3.1 Experimental Setup

We evaluate `SMLT` using three popular ML training frameworks: Tensorflow 2.0 [23], MXNet 1.7.0 [47] with gluon-cv 1.4.0, and Pytorch 1.8.0 [127]. For benchmarking tasks, we use the following ML models:

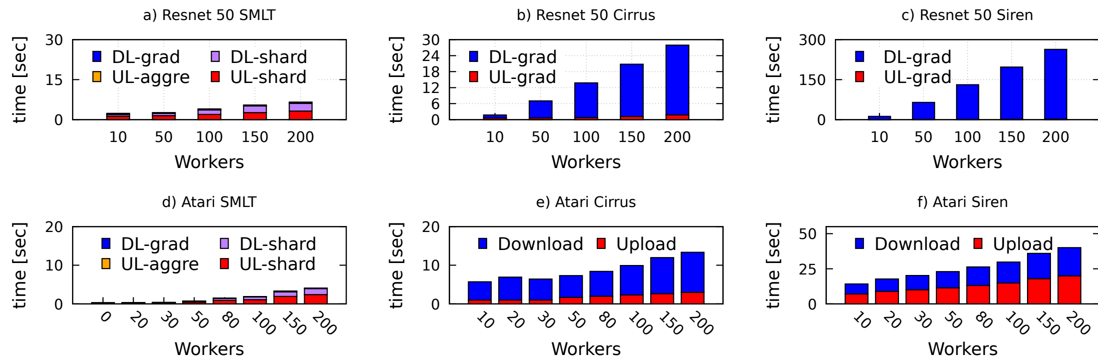


Figure 5.4: Communication time breakdown comparison of SMLT, Cirrus and Siren.

- ResNet-18 (18 layers, 11 million parameters) and ResNet-50 (50 layers, 23 million parameters) [71] with residual functions for image classification.
- Bert-Small [142] (66 million parameters) and Bert-Medium [155] (110 million parameters) for natural language processing.
- Atari break out game (50 million frames [83]) for reinforcement learning (RL).

In all experiments, we split the training dataset into smaller sets (up to 250 MB in size) and store them on AWS S3, which serves as our object store. We vary the memory allocated to training workers for each experiment based on the model size being evaluated. Unless otherwise noted, for hierarchical model synchronization in SMLT, we use the in-memory Redis key-value store hosted on AWS ECS, which serves as our parameter store. The AWS S3, AWS ECS, and the Lambda functions for workers are all deployed in the same AWS region, i.e., us-east-1. For cost calculation, we use the AWS cost formula along with the cost of object store, and parameter store cost associated with each method.

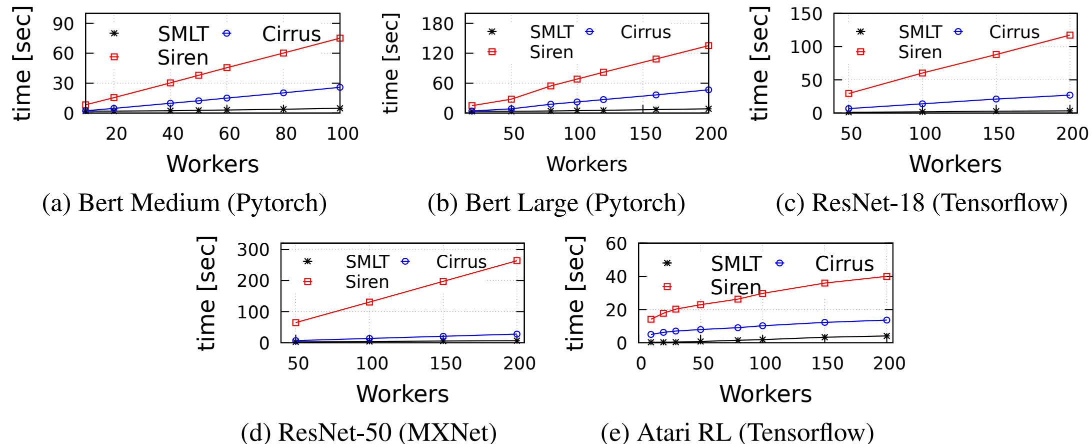


Figure 5.5: Per iteration communication time comparison between SMLT, Cirrus and Siren.

5.3.2 Effectiveness of Hierarchical Model Synchronization

We evaluate the effectiveness of SMLT's proposed hierarchical model synchronization by comparing the communication time between SMLT and the two baselines, i.e., Siren [161] and Cirrus [44]. Figure 5.5 shows the communication time as a function of the number of workers (scale-out) per iteration for all 5 benchmarks, respectively. We observe that for all three systems the communication time increases linearly as the number of training workers increases. This increase is due to the fact that more workers will produce more gradients that need to be transferred from the workers to the aggregator via the external storage. However, the increase in overhead for SMLT is substantially lower than Siren and Cirrus, thanks to the hierarchical aggregation mechanism along with the fast parameter store, which together significantly reduce the communication overhead in a serverless environment.

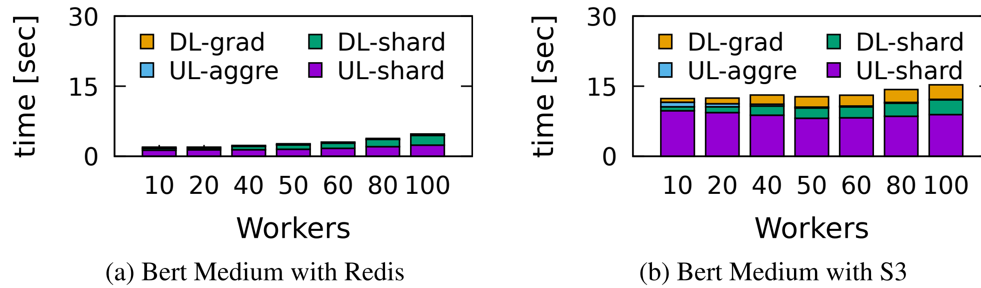
For 2 representative benchmarks, we further breakdown the communication time into individual communication steps during each training iteration, as shown in Figure 5.4. While using SMLT, we denote *UL-Shard* as the time taken for the shard generator to split

the gradients and upload them to the parameter store, *DL-Shard* as the time for the shard aggregator to download and aggregate the shards to form the aggregated shards, *UL-aggr* as the time taken for the shard aggregator to upload the aggregated shards, and *DL-grad* as the time for the global aggregator to download the aggregated shards (see Figure 5.2 for the detailed process description). For Siren and Cirrus, *UL-grad* refers to the time taken for each worker to upload gradients to the cloud storage, and *DL-grad* refers to the time taken for workers to download the parameters of all other workers for updating the model after every iteration. From Figure 5.4, one can see that, for both Siren and Cirrus, the main bottleneck often is the *DL-grad* step. On the other hand, `SMLT`'s sharding approach for uploading and downloading gradients results in a significant reduction of the *DL-grad* overhead.

It is worth noting that, in the case of the RL-based Atari model, the size of the uploaded data is larger than the Resnet-50 model. The longer uploading time is due to the large simulation data shared by each worker after every iteration. We observe the impact of larger uploaded data in Figure 5.4[d-f]. In the case of Siren, this impact is more pronounced, reflecting the limitation of a centralized parameter server.

5.3.3 Intermediate Storage: S3 vs Redis

Existing methods such as Cirrus [44] and λDNN [168] propose to use S3 as storage system for storing model parameters during the intermediate steps for model aggregation. However, proper selection of the cloud storage systems is critical to the scalability and cost effectiveness. With the increase in number of worker the communication overhead can contribute significantly to overall training time. In this section To demonstrate the impact



(a) Bert Medium with Redis

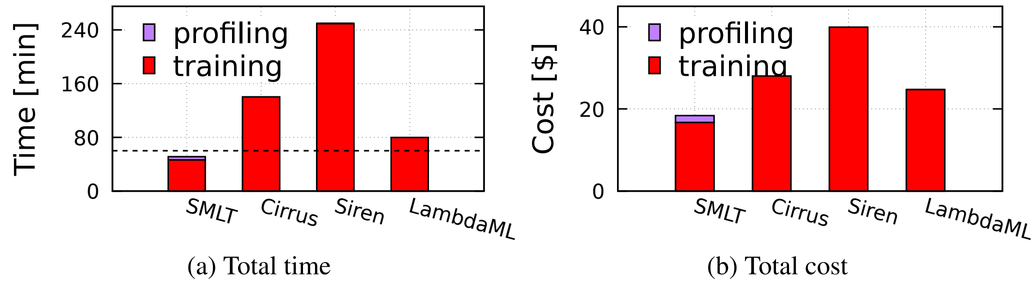
(b) Bert Medium with S3

Figure 5.6: Performance Comparison of Bert Medium Redis vs S3 as an external Storage.

of storage system we compare implementation of SMLT using S3 and Redis key-value store respectively as the underlying storage system. We deploy Redis on a cloud container service AWS ECS using using *C4.4xlarge* instances type. To reduce the randomness of the network delay, we place the ECS containers and our worker function instances in the same virtual private cloud. Figure 5.6 shows the comparison results. We can see that using Redis can significantly reduce the communication time compared to using S3 as storage system. In the best case scenario with 10 workers Redis based storage system improves the overall communication time by 8× and with 100 workers it achieves almost 6× improvement over S3. This further improves the overall training time and cost, as cost of serverless functions are directly correlated with the total run time. Similarly, pay-per-use cost model of AWS ECS also contrinbutes towards cost effectiveness.

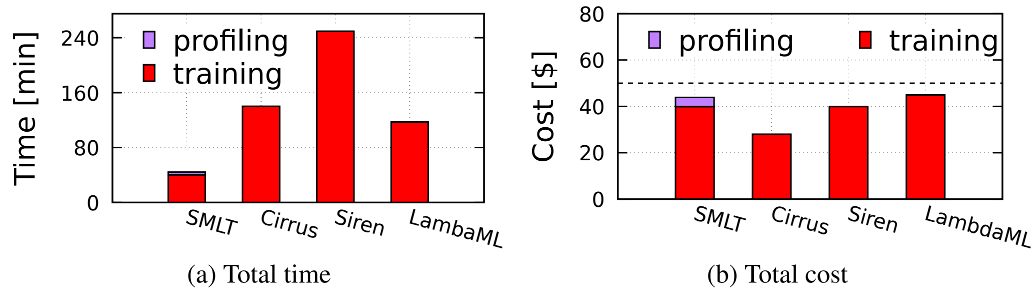
5.3.4 User-centric Deployments

We use the following two scenarios (exemplified in Section 5.1.2) to evaluate our user-centric deployment and execution approach. The scenario 1 is to, given a user-specified training time limit, optimize the monetary cost. The scenario 2 is to, given a user-specified monetary budget, minimize the training time. The results are shown in Figure 5.7 and Figure 5.8, respectively.



(a) Total time (b) Total cost

Figure 5.7: (Scenario 1) Minimize the monetary cost with a 1-hour training time limit for Bert-Medium with Pytorch. Note that, `SMLT` has profiling time and cost while other frameworks do not. For a fair comparison, we also demonstrate the profiling time and cost in `SMLT`.

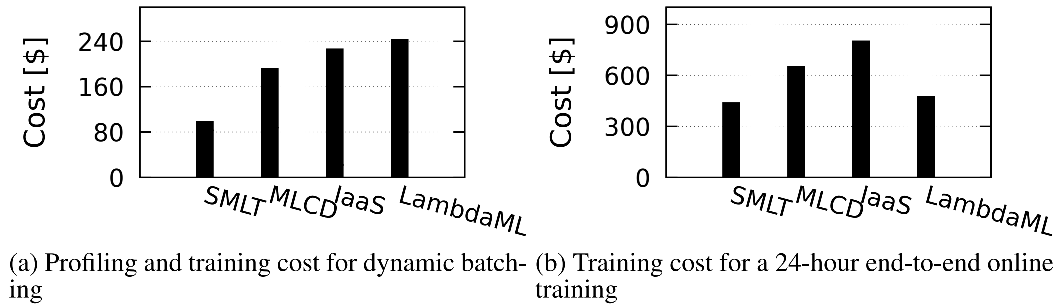


(a) Total time (b) Total cost

Figure 5.8: (Scenario 2) Minimize the training time with a $50 training cost budget for Bert-Medium with Pytorch. Note that, `SMLT` has profiling time and cost while other frameworks do not. For a fair comparison, we also demonstrate the profiling time and cost in `SMLT`.

In Scenario 1, only `SMLT` meets the user-specified time limit of completing 50 epochs within an hour, as Siren and Cirrus do not consider such user requirements. `SMLT` also has the lowest cost, thanks to its optimized deployment and execution. Figure 5.7 also shows that `SMLT` would have been able to achieve the best accuracy with the most number of epochs at the lowest cost, if we had stopped training at the time limit (i.e., 1 hour). In Scenario 2, as shown in Figure 5.8, all three frameworks meet the user-specified monetary budget of 50$ for completing 50 epochs, although Siren and Cirrus achieve it by coincidence. `SMLT` achieves significantly lower training time compared to the other two frameworks, because of its optimizations to match the user-specified budget.

These results validate that `SMLT` can meet the user-specified time and budget requirements while optimizing the monetary cost and training time, respectively. This capability provides



(a) Profiling and training cost for dynamic batching

(b) Training cost for a 24-hour end-to-end online training

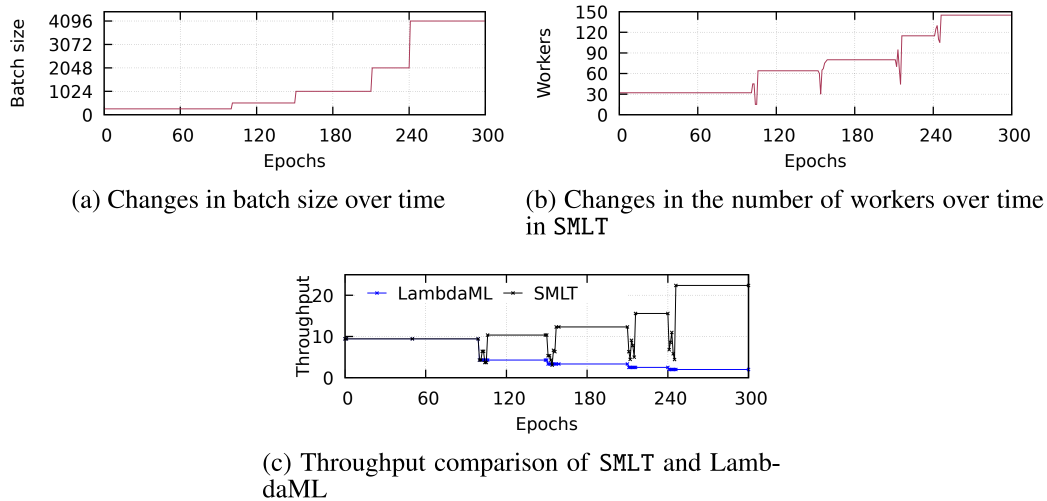
Figure 5.9: Cost comparison of profiling and training via our Bayesian optimizer, for dynamic batching and online learning, using Resnet-50 with Pytorch. Experiments using Resnet-18 with Tensorflow produced similar results.

important practical benefits to ML practitioners compared to other frameworks that are oblivious to user requirements.

5.3.5 Dynamic Batching and Online Learning

To show how our task scheduler can exploit serverless computing's 'pay-as-you-go' pricing model using the Bayesian optimizer, we conduct experiments for two cases: *dynamic batching* and *online learning*. In these experiments, we compare `SMLT` with MLCD (a VM-based MLaaS platform [173]), LambdaML [80], and the IaaS setup as described in [80], we employ the training of a single ResNet50 model on the CIFAR-10 dataset, consistent with the settings used in LambdaML [80].

In the case of dynamic batching, the memory size and the number of workers can be scaled without restarting the worker functions. As shown in Figure 5.9a, the profiling overhead, and the cost associated with it, in `SMLT` is significantly lower than in MLCD. Although LambdaML may not perform as well as the IaaS setup, `SMLT` showcases that an ML training platform using public serverless computing can provide a cost-effective and



(a) Changes in batch size over time

(b) Changes in the number of workers over time in `SMLT`

(c) Throughput comparison of `SMLT` and LambdaML

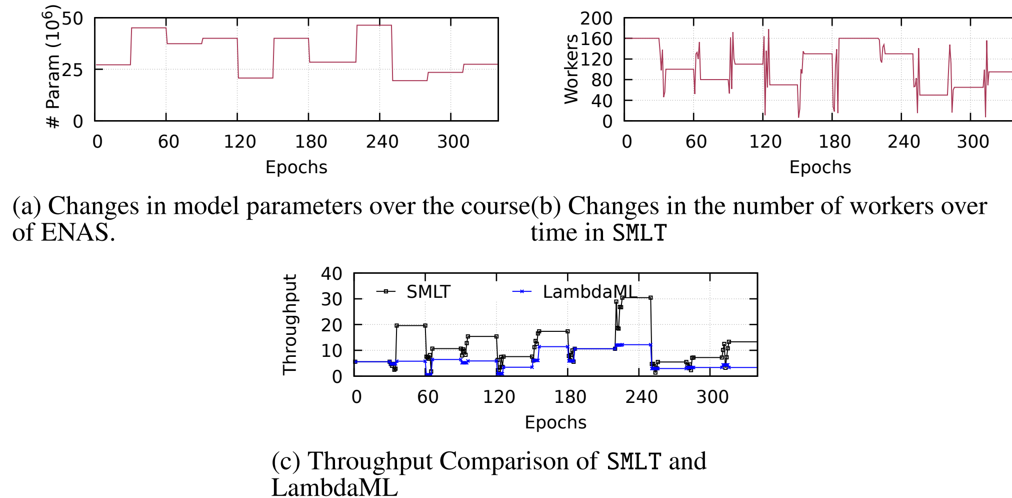
Figure 5.10: Throughput comparison for dynamic batching.

scalable solution.

In the case of online learning, due to the non-deterministic training time and the continuous resource provisioning, the cost for MLCD is higher than `SMLT` as shown in Figure 5.9b. Similarly, IaaS has high costs due to its continuously running, but at times idle, VM resources. On the other hand, LambdaML and `SMLT` showcase how an ML training platform can exploit the cost-effectiveness of public serverless computing.

Note that, `SMLT` offers an adaptive approach for dynamic workload conditions, whereas LambdaML does not. To demonstrate this crucial difference, we conduct another experiment and compare `SMLT`'s task scheduler to LambdaML (with a randomly-assigned, fixed resource allocation scheme).

Figure 5.10a illustrates how the batch size changes over time. Figure 5.10b depicts the corresponding worker changes that adaptively adjust in response to the batch size changes. Furthermore, Figure 5.10c presents a comparison of training throughput, measured as processed data points per second, over time between our SMLT method and LambdaML.



(a) Changes in model parameters over the course of ENAS.

(b) Changes in the number of workers over time in SMLT

(c) Throughput Comparison of SMLT and LambdaML

Figure 5.11: Throughput comparison for ENAS.

We assume that the user optimizes the resource allocation for the initial training configuration (i.e., batch size). Therefore, SMLT and LambdaML achieve similar training throughput initially. However, when the batch size changes, LambdaML's user-defined configuration observes sub-par training performance. In contrast, SMLT adjusts to the varying training parameter dynamically. We also observe the variation in throughput when the batch size changes are detected by the task scheduler via monitoring the workers. Cost-wise, SMLT also reduces the training cost by over 30% compared with LambdaML thanks to its dynamic resource allocation via the task scheduler.

5.3.6 Neural Architecture Search

Neural architecture search (NAS) often deploys up to hundreds of training jobs in parallel or series to search for a well-performing model architecture. In such scenarios, each training job generally deploys a different model architecture as part of the exploration phase. As such, the amount of resources required is dynamic since it depends on the size of the model being deployed. We demonstrate the adaptive behaviour of SMLT for optimal resource allo-

cation for the widely-used NAS framework called ENAS [131]. Figures 5.11a and 5.11b display model size over time and the corresponding changes in number of workers, respectively, Figure 5.11c demonstrates the throughput comparison between SMLT and LambdaML. Similarly, We assume that the user optimizes resource allocation for LambdaML based on the first model. Therefore, both SMLT and LambadML achieve similar throughput at the beginning of the exploration. However, as the model size changes, SMLT optimizes the resource allocation accordingly, resulting in improved throughput. SMLT achieves 3× cost savings compared to LambdaML through its dynamic resource allocation scheme.

5.4 Related Work

Like in any infrastructure-related problem, achieving a scalable and cost-effective solution becomes important for running ML workflows in the cloud. Earlier designs for MLaaS focus on addressing scalability [11, 185, 188], privacy [77, 154], and ease-of-use [152, 170], but not on cost-effectiveness. Major cloud providers, like Microsoft Azure [6] and AWS [15], have extended their existing statically-priced cloud platforms to commercial MLaaS without offering user-centric deployment guarantees. Similarly, recent academic efforts [69, 91, 167, 183] utilize traditional cloud resources (e.g., VMs, containers) as their underlying platforms, but require extraneous management policies or understanding the underlying resources as pointed out in [88, 181].

Some systems attempt to tackle the dynamic resource demands of ML tasks for specific use cases. DynamoML [48] proposes a set of online dynamic management techniques that perform scaling, job preemption, workload-aware scheduling, and elastic GPU sharing for different parts of the ML development workflow consisting of modeling, training, and in-

ference jobs. However, DynamoML is designed for traditional cloud settings instead of serverless, which leads to coarse-grained resource allocation strategies that are not suitable for a serverless setting. In addition, DynamoML requires prior knowledge of the workloads to be able to schedule the resource allocation effectively, making it impractical for workflows with lots of diverse workloads and workloads without prior knowledge. Take the NAS workflow as an example, it contains many potential models with unknown resource demands. Schuler et al. [145] uses reinforcement learning for resource allocation in serverless, but it requires expensive training and tuning of the reinforcement learning models. Our experimental results show that the simpler and faster Bayesian based approach proposed in this chapter works equally well but much more lightweight, see Figure 5.1. λDNN [168] also aims at using serverless infrastructure for training ML models. However, this work focuses on training single static ML model instead of the dynamic workflow scenario that this chapter targets for. Barista [38] uses a workload characterization and prediction mechanism to maintain SLOs with minimal cost. However, Barista's mechanisms are limited to the inference phase that has relatively less variation in resource dynamics than for the training. In contrast to all of these systems, `SMLT` does not require prior knowledge of the workloads because of its adaptive capability, does not require extra tuning of its scheduler supported by the Bayesian optimizer, and covers ML training phases that have more dynamic resource requirements.

5.5 Summary

We believe serverless plays a critical role in future machine learning routines. `SMLT` is the first fully-automated serverless framework for scalable and adaptive ML design and training on public cloud. The key contributions of `SMLT` are to: 1) equip public serverless

platforms with an overarching view and dynamic adaptation capabilities for ML workflows, 2) offer user-centric deployment and execution of ML tasks on serverless platforms, and 3) provide an open-source end-to-end framework that provides fast gradient synchronization and addresses the key limitations of serverless ML platforms. Our extensive experimental evaluation demonstrates the effectiveness and robustness of `SMLT`, which outperforms the state-of-the-art approaches by up to 8× faster in training speed and 3× lower in monetary cost.

CHAPTER 6

DIFFUSION-BASED, DATA ASSIMILATION ENABLED WIND SUPER-RESOLUTION

6.1 Preliminary

6.1.1 Denoising Diffusion Probabilistic Models

Denoising Diffusion Probabilistic Models (DDPMs) [73] are a type of generative model that use a progressive denoising process to generate high-quality images from pure Gaussian noise. These models consist of two main processes: a forward process and a reverse process. In the forward process, noise is incrementally added to the data over a series of forward steps, ultimately transforming the original image into pure Gaussian noise. At each time step t, a small amount of noise is added, controlled by a predefined noise schedule β_t. This schedule determines the rate and extent of noise addition at each step. Over T time steps, the image is progressively corrupted by Gaussian noise until no recognizable features remain.

$$q(\mathbf{x}_t \mid \mathbf{x}_{t-1}) = \mathcal{N}\left(\mathbf{x}_t;\ \sqrt{1-\beta_t}\mathbf{x}_{t-1}, \beta_t \mathbf{I}\right) \tag{6.1}$$

In the reverse process, DDPMs undo the forward process step-by-step, starting from random noise and iteratively refining the sample to generate images. The reverse process is

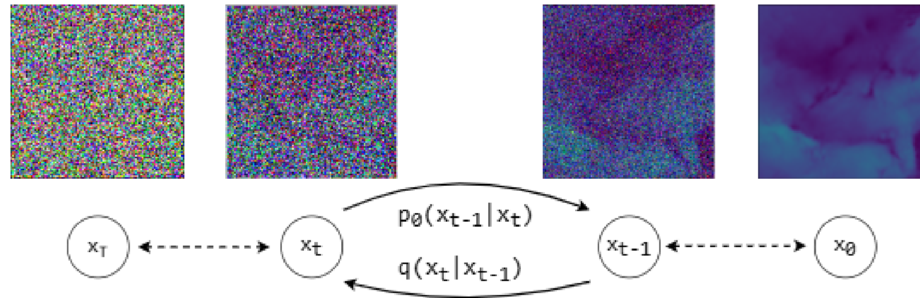
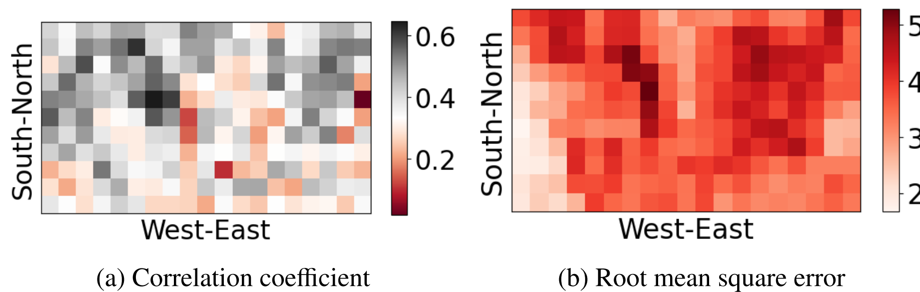


Figure 6.1: The figure demonstrates the DDPM's forward process of adding noise and the reverse denoising process.



(a) Correlation coefficient (b) Root mean square error

Figure 6.2: Correlation coefficient (left) and root mean square error (right) between WRF and HRRR data averaged over 5 days. Despite advances in NWP, significant model biases remain, especially at coarser resolutions and in areas where complex terrain makes parametrization difficult. Data driven approaches can rectify these differences without the need for computationally expensive physics.

guided by a deep neural network denoiser, typically a U-Net for image data. The neural network is trained to predict the noise added at each step in the forward process, gradually learning to denoise the image. The objective of training is to minimize a variational bound on the negative log-likelihood, which reduces to a weighted mean squared error (MSE) between the actual noise added in the forward process and the noise predicted by the network. By optimizing this loss (MSE), the model learns to effectively reconstruct the orginal image from noise, one step at a time.

$$p_\theta(\mathbf{x}_{t-1} \mid \mathbf{x}_t) = \mathcal{N}\left(\mathbf{x}_{t-1}; \mu_\theta(\mathbf{x}_t, t), \sigma_t^2 \mathbf{I}\right) \tag{6.2}$$

When doing training for the neural network model, the objective is to approximate the

reverse distribution such that we can effectively denoise the noisy samples, we show the forward and reverse diffusion process in Figure 6.1. The model is trained to predict the noise $\boldsymbol{\epsilon}$ that was added during the forward process. The training loss can be expressed as:

$$L_t = \mathbb{E}_{\mathbf{x}_0,\boldsymbol{\epsilon},t}\left[||\boldsymbol{\epsilon} - \epsilon_\theta(\mathbf{x}_t, t)||^2\right] \tag{6.3}$$

Where $\boldsymbol{\epsilon} \sim \mathcal{N}(0, \mathbf{I})$ represents the Gaussian noise, and $\epsilon_\theta(\mathbf{x}_t, t)$ is the model's predicted noise. The training objective minimizes the difference between the true noise $\boldsymbol{\epsilon}$ and the predict noise ϵ_θ. Through this training, the model learns to reverse the forward process, allowing it to generate and refine images by starting from random noise.

6.1.2 Data

The ultimate goal of this study is to produce high-resolution wind speed data at turbine rotor-height and at the wind farm-scale resolution. Ideally, we would use numerical model simulations with resolutions in the tens to hundreds of meters, supplemented by assimilating in-situ observations, to train our diffusion-based SR model. However, such high-resolution data are not widely available over a large geographic area. Numerical models are typically run at resolutions of a few kilometers, and hub-height in-situ observations are sparse and often not publicly accessible. To demonstrate our workflow, we utilize the state-of-the-art WIND Toolkit Long-Term Ensemble Dataset (WTK-LED; [54]), which provides wind data at a spatial resolution of 2 km and temporal resolution of 5 minutes. This dataset is used to train our SR model. We then incorporate high-resolution rapid refresh (HRRR) data as a proxy for ”ground truth” to assist with data assimilation, the HRRR data is at

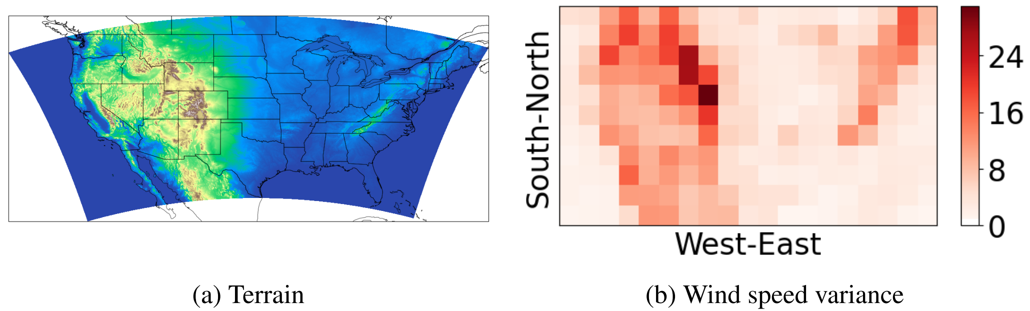
a 3 km spatial resolution. Once validated, our framework will be expanded to include ultra-high-resolution numerical simulations (e.g., large eddy simulations) and in-situ rotor-height observations from meteorological towers. We illustrate the relationship between the two datasets in Figure 6.2. The correlation coefficient and RMSE are calculated based on the average wind speed over five days.

High resolution data for training WTK-LED provides wind speed and direction at multiple altitudes (10 m to 500 m above ground level) as part of its Weather Research and Forecasting (WRF) model outputs. Detailed model configurations are described in [54]. The dataset spans three years (2018–2020), with 2 km spatial and 5-minute temporal resolutions, covering 2649 (west-to-east) × 1949 (south-to-north) grid cells across the U.S. and surrounding oceans. For training, we sampled 100 m wind speeds every three hours from 2018 to 2020. Terrain height (elevation) data from the WRF model were also included to account for the influence of terrain on wind patterns. For testing, we used 2020 data sampled at 1-hour intervals. Further details of this dataset are provided in Table 6.1.

Observational data for data assimilation HRRR data, developed by National Oceanic and Atmospheric Administration (NOAA), serves as our ”ground truth” for data assimilation. HRRR operates with a 3 km spatial resolution and hourly temporal resolution, incorporating radar data every 15 minutes to enhance output accuracy. HRRR has been extensively validated for wind resource assessment across various terrains [132], [49], [156]. Although HRRR performs well compared to in-situ observations, it still struggles in complex terrains, such as mountainous regions. While not a perfect ground truth, HRRR provides a flexible data source, allowing us to sample random locations and create in-situ data format. This flexibility is invaluable for demonstrating our workflow and testing model performance.

Data	WRF	HRRR	HGT
Source	WTK-LED	NOAA	WTK-LED
Type	Wind	Wind	Terrain
Spatial	2 km	3 km	2 km
Temporal	3 hr	1 hr	Constant
Years	2020	2020	N/A
HR	128×128	128×128	128×128
LR	16×16	16×16	16×16

Table 6.1
Dataset specifications for WRF, HRRR, and HGT



(a) Terrain (b) Wind speed variance

Figure 6.3: The figure illustrates the relationship between terrain complexity and the corresponding average wind speed variance over five days, showing that more complex terrain correlates with greater wind speed variance.

Given that the WTK-LED and HRRR datasets cover slightly different regions, we first crop the WTK-LED data to match the HRRR coverage. Next, we regrid the HRRR data onto the WRF grid, ensuring that all datasets are aligned on the same grid for training and data assimilation. Due to the large data size—comprising over five million grid cells—the entire domain is divided into smaller patches, each containing 168×168 grid cells. All results presented in this chapter are conducted at the patch level.

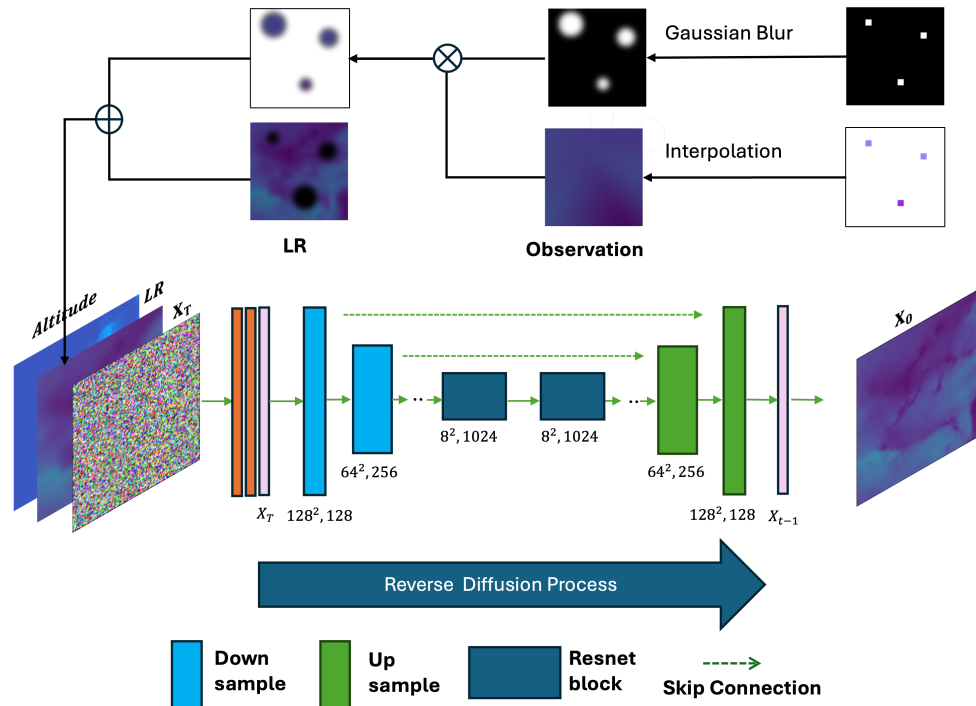


Figure 6.4: Image generation process incorporating both data assimilation and diffusion to downscale wind speed data. We interpolate sparse observation data over the inference grid then apply a soft bleed mask. The observation pixels are in-painted to the simulation results, and the composited image is used to guide the image generation process through the diffusion model, extra terrain information serve as condition during the reverse diffusion process.

6.2 Method

6.2.1 Overview

In our work, we integrate the data assimilation process into the diffusion-based downscaling workflow by inpainting sparse observational data into low-resolution images. These images are then processed by a diffusion model for downscaling. We employ the SR3 model as our primary downscaling tool. During training, terrain information is included as an additional condition alongside the initial condition x_t and low resolution image, enhancing the model's downscaling performance. During inference, to ensure the sparse observational data effectively influences the output, we have developed an method for dynamically adjusting the impact radius of the Gaussian kernel. This design considers both the terrain information and the variation in wind speed surrounding the ob-

servation points, optimizing the effectiveness of the observations.

6.2.2 Condition in Training

Wind speed is influenced by various factors including temperature and pressure differences, which drive atmospheric movements from high to low-pressure areas, altitude plays a role as wind generally speeds up at higher elevations due to reduced surface friction. Surface roughness from terrain and vegetation also affects wind by increasing friction and slowing it down. In our case, since we investigate the 100 meters height wind speed above ground, one of the important impact factors to the wind speed is the altitude. By comparing the wind speed image and the terrain information image, we can find that the wind speed is highly correlated with the terrain information, as showed in Figure 6.3, which inspire us to incorporate the terrain information as a condition to training the diffusion model.

Our backbone model for super-resolution tasks is the SR3 model, which employs a UNet architecture as shown in Figure 6.4. The SR3 model starts with a pure Gaussian noise image, x_t, and a low-resolution condition image. We enhance this setup by integrating terrain information directly into the condition. Specifically, we add terrain data as an extra channel using a 128x128 matrix, This matrix is concatenated with the interpolated low-resolution image and Gaussian noise image to form a comprehensive input consisting of the terrain channel, the low-resolution image, and the Gaussian noise image, these inputs are then processed by the UNet for denoising. Throughout this chapter, we use this enhanced model, which incorporates additional terrain conditions, as our pretrained model for downscaling tasks. Unless otherwise specified, this pretrained model with integrated terrain information serves as the backbone model for `WindSR`.

6.2.3 Data Assimilation in Inference

Following similar inpainting techniques in [75, 114, 149], we incorporate observations into the conditioning image prior to the inference phase. To simulate sparse observations, we randomly sample observation points from HRRR data. To quantify the effectiveness of the assimilation process more easily, we select either single or multiple observation points within 128x128 pixel areas. Given that the resolution in WRF data is 2 km, this corresponds to having single or multiple observation points across a 256 km x 256 km area. In real world, these observation points represent weather variable data recorded at observation towers, weather balloons or weather radar within the model simulation image domain. We use these sparse observation points to perform interpolation, scaling them to match the size of the simulation image. This interpolated image is then utilized to create a surrounding support area using the softmask in next step, the details are showed in Figure 6.4.

We adapt the softmask method from DiffDA [74] to blend observation data with WRF data using a Gaussian kernel softbleed function, creating a supportive area around each observation point. A key innovation in our approach is the dynamic adjustment of the impact radius d, influenced by the terrain variance and surrounding WRF wind speed data variance. This design accounts for the fact that wind speed changes can vary dramatically across different regions. For instance, in areas with complex terrain, wind speed can fluctuate rapidly, which diminishes the representational capacity of a single observation point. In such cases, we limit the impact radius of the observation point to a smaller area. Conversely, in flat regions where wind speeds are more consistent and show less variance, an observation point can effectively cover a larger area. As illustrated in Figure 6.3, for the Mountains region with complex terrain, exhibit higher wind speed variance compared to flat regions. Previous studies [74] have overlooked this variability, treating each observation point as having an equal impact on the simulation data, which has led to suboptimal performance.

Our method for determining the observation point's impact radius involves incrementally increasing the radius by 1 pixel at each step and calculating the variance within the terrain and the model

Algorithm 1 Dynamic Impact Radius

Input: Coordinate p, terrain deviation threshold T_1, wind speed deviation threshold T_2

```
1:  procedure DIR(p, T_1, T_2)
2:      Define min_radius ← 1, max_radius ← 6
3:      Initialize the impact radius r ← min_radius
4:      while r < max_radius do
5:          a = {(x, y) ∈ R^2 | (x − p_x)^2 + (y − p_y)^2 ≤ r^2}
6:          σ_h = sqrt( (1/N) Σ_{p̂∈a} (h_p̂ − h̄)^2 ), h_p̂ : terrain height at point p̂
7:          σ_s = sqrt( (1/N) Σ_{p̂∈a} (s_p̂ − s̄)^2 ), s_p̂ : wind speed at point p̂
8:          if σ_h < T_1 and σ_s < T_2 then
9:              r ← r + 1
10:         else
11:             break
12:     return the impact radius r
```

	PSNR (↑)	SSIM (↑)
SRCNN [53]	27.34	0.7005
ESRGAN [166]	28.16	0.7021
WindSR w/o Terrain	31.69	0.8105
WindSR w/ Terrain	**32.83**	**0.8207**

Table 6.2
THE AVERAGE SSIM AND PSNR ACROSS 200 RANDOM SAMPLED IMAGES FOR VARIOUS MODELS.

simulation data (WRF data) for the current radius covered region. We set a threshold for variance, once exceeded, we cease radius expansion and set this final radius as the definitive impact radius for this observation point, the radius boundaries are constrained between 1 and 6 pixels. This dynamic adjustment process is detailed in Algorithm 1, illustrating how we tailor the kernel size based on localized environmental conditions of the observation points.

Once the impact radius is determined, we use this kernel to mask out portions of the interpolated observation image and incorporate these segments into the model simulation image for downscaling. This composite image process can be formalized as shown in Equation 6.4, where m_s represents the weight of the observation values through the softbleed function, x_{HRRR} is the observation image with sparse observation point, and x_{WRF} is the model simulation image. The resulting composite image, x, is then processed through the pre-trained WindSR downscaling model, which integrates pure noise and terrain condition. Following the downscaling process, we obtain a high-resolution image with the observation data assimilated, effectively enhancing the detail and accuracy of the

	Radius 2		Radius 4		Radius 6		Dynamic	
	MAE (↓)	**RMSE (↓)**	**MAE (↓)**	**RMSE (↓)**	**MAE (↓)**	**RMSE (↓)**	**MAE (↓)**	**RMSE (↓)**
SRCNN [53]	1.78	2.17	1.70	2.11	1.69	2.10	1.69	2.09
ESRGAN [166]	1.92	2.34	1.82	2.25	1.80	2.23	1.81	2.24
WindSR w/o Ter.	1.81	2.20	1.74	2.15	1.72	2.13	1.71	2.12
WindSR	**1.76**	**2.15**	**1.70**	**2.08**	**1.68**	**2.12**	**1.64**	**2.01**

Table 6.3
Different models with different radius comparison.

simulation data.

$$x = (m_{weight} \odot x_{\text{HRRR}}) + ((1 - m_{weight}) \odot x_{\text{WRF}}) \tag{6.4}$$

The composite image x combines simulation model data (WRF data) with sparse observation data (HRRR data) and serves as the initial condition for the reverse diffusion process.

6.3 Experiment

We conduct extensive experiments to assess the downscaling and data assimilation processes, comparing them with state-of-the-art methods. The experiment setting is detailed in subsection 6.3.1, in subsection 6.3.2, we perform comparative experiments against various super-resolution models to evaluate the effectiveness of our downscaling approach. subsection 6.3.2 further explores the effectiveness of assimilation of our methods.

6.3.1 Implementation

Our WindSR downscaling model is trained on a dataset of 10,000 images, each with a resolution of 128x128 pixels. These images are randomly sampled from 2020 WRF wind speed data. For

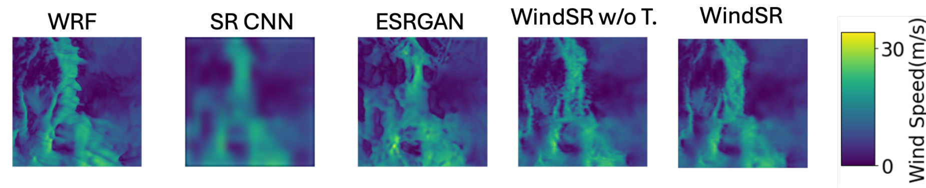


Figure 6.5: Visual comparison of different SR models

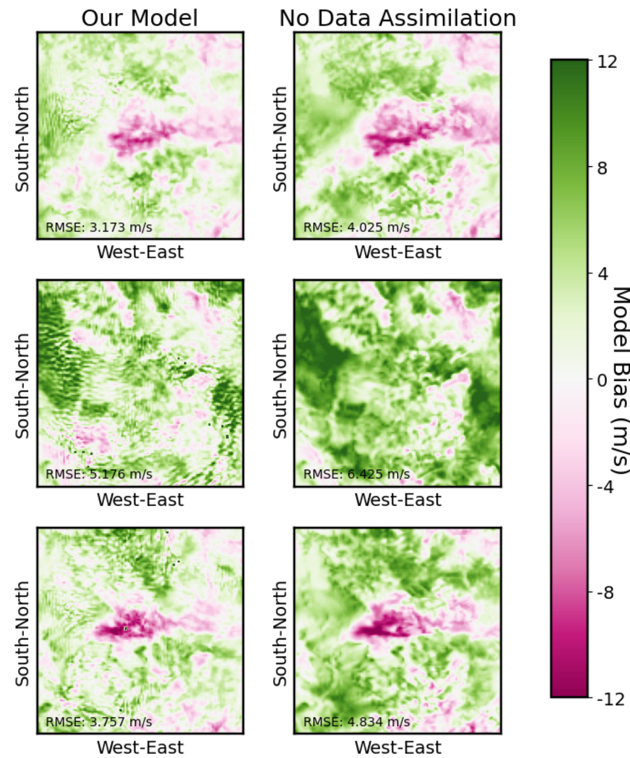
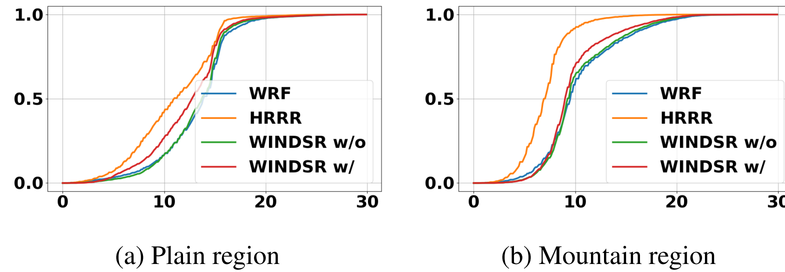


Figure 6.6: All six images represent the same geo-spatial location, and each row corresponds to images from the same time step. The images exhibit minimal model bias, indicated by white or near-white colors.

training, the images are downsampled to 16x16 pixels, creating pairs of high-resolution and low-resolution images. The training is conducted on a single node equipped with four A100 GPUs, spanning 300,000 iterations over six days. The parameter settings adhered to those specified in the original SR3 model. All inferences are performed on a single A100 GPU, with each image requiring approximately 90 seconds to process.



(a) Plain region (b) Mountain region

Figure 6.7: Cumulative density functions of wind speeds across different terrain regions of the US, comparing values recorded by HRRR, WRF, and our model. In both regions, data assimilation brings the distribution of wind speeds from our model closer to HRRR data.

6.3.2 Comparison of Downscaling

Various deep learning models are widely used in the field of image super-resolution, each with its own unique characteristics. The SRCNN [53] model is a straightforward CNN-based approach known for being lightweight yet effective. On the other hand, GAN models, particularly the ESRGAN [166], have shown significant advantages in super-resolution tasks. ESRGAN [166] leverages the principles of generative neural networks to produce high-resolution images with remarkable detail. The SR3 model, a vanilla implementation, has become one of the most popular choices for super-resolution tasks today. In our study, we compare the performance of these super-resolution models within the WindSR framework for downscaling tasks. We particularly highlight the advantages of our pretrained model, demonstrating its superior performance. Table 6.2 showcases a comparative analysis of average SSIM (Structural Similarity Index) and PSNR (Peak Signal-to-Noise Ratio) across 200 images for a variety of super-resolution models. The WindSR model with terrain data integration (WindSR w/ Terrain) demonstrates superior performance, achieving the highest PSNR of 32.83 and SSIM of 0.8207 among the listed models. In contrast, the SRCNN model has the lowest PSNR and SSIM values at 27.34 and 0.7005, respectively. Additionally, the WindSR model without terrain data (WindSR w/o Terrain) still outperforms the more traditional SRCNN and ESRGAN models with a PSNR of 31.69 and an SSIM of 0.8105. This suggests that the enhancements in the WindSR model, particularly with the inclusion of terrain data, significantly improve image quality metrics compared to the other models evaluated.

To visually demonstrate the performance differences between the models, we select a representative image tile for detailed visualization and comparison, this image is 128x128 pixel which covered 256 km x 256 km areas. Figure 6.5 showcases the visual comparison of different super-resolution (SR) models. From this figure, it is evident that while SRCNN [53] achieves relatively acceptable metrics compared to ESRGAN [166], it substantially lacks in capturing fine-grained details. In contrast, the WindSR model excels both in metrics and visual effects. When training the ESRGAN [166] model, initial challenges with convergence were encountered due to the high weight of content loss in the model's default setting, which is typically designed for objects with clear textures and contours. This approach was less effective for wind speed images, which do not have distinct contour patterns. To address this, we adjust the loss weights, reduce content loss while increasing pixel loss, enabling ESRGAN [166] to converge and capture more fine-grained details. Conversely, SRCNN [53], with its shallower layers, effectively simulates the interpolation process but lacks the ability to generate detailed visuals, its metric success is largely due to fitting pixel values.

6.3.3 Comparison of Data Assimilation

From the previous section, we verified the effectiveness of the diffusion model for downscaling wind speed data. In this section, we assess the data assimilation process and further validate the effectiveness of our design. Differing from the earlier section that utilized PSNR and SSIM to evaluate image super-resolution quality, we now focus on the pixel-level differences between the ground truth HRRR data and the generated images with and without data assimilation. Additionally, we evaluate our dynamic radius design against various fixed radius configurations to demonstrate the enhanced effectiveness of our approach.

Table 6.3 detailed the performance of various super-resolution models, including SRCNN [53], ESRGAN [166], WindSR with and without terrain conditioning (Ter.), measured by Mean Absolute Error (MAE) and Root Mean Square Error (RMSE) across different radius settings and a dynamic

radius scenario. The results show that the WindSR model with conditioning consistently outperforms the others, particularly in the dynamic radius setting where it achieves the lowest MAE (1.64) and RMSE (2.01) at the same time. This indicates that dynamic radius in the WindSR model substantially improves accuracy, reducing both the MAE and RMSE across all tested scenarios.

To visually demonstrate the effectiveness of our model, we provide Figure 6.6 to show a comparative analysis of model biases in two scenarios: one is WindSR which incorporates data assimilation, and a counterpart model without data assimilation. In each scenario, our model consistently shows lower Root Mean Square Error (RMSE) values (3.173 m/s, 5.176 m/s, 3.757 m/s) compared to the no data assimilation model (4.025 m/s, 6.425 m/s, 4.834 m/s), highlighting the effectiveness of data assimilation in reducing simulation model errors.

We further illustrate the effectiveness of our data assimilation techniques with the CDFs of wind speeds across different terrain regions of the United States, shows in Figure 6.7, specifically focusing on plain and mountainous areas. These plots compare wind speed values of HRRR data, WRF data, and our WindSR model. Our visualizations show that in both terrain types, the integration of data assimilation significantly enhances the alignment of wind speed distributions from our model with the HRRR data, which serves as the ground truth. This closer alignment validates the superior performance of our model in capturing and replicating accurate wind speed dynamics.

6.4 Summary

In this chapter, we present `WindSR`, an innovative wind super resolution method that synergistically combines sparse observational data with rich simulation data in the process of downscaling using advanced diffusion models. We introduce a dynamic radius technique that optimally merges observational and simulation data, establishing a novel condition for the diffusion process that enhances its effectiveness. Furthermore, terrain information is meticulously integrated during both the

training and inference stages to augment the data synthesis process. We rigorously evaluate our approach against conventional CNN and GAN-based models in terms of downscaling efficiency and data assimilation effectiveness. Our findings demonstrate that WindSR significantly outperforms these traditional methods, offering substantial improvements in precision and integration of diverse data sources.

CHAPTER 7

CONCLUSION AND FUTURE WORK

7.1 Conclusion

The primary objective of this dissertation is to explore the significant challenges that currently hinder the efficiency of AI for Science infrastructure, particularly in terms of resource utilization on large-scale supercomputers and cloud platforms. This research seeks to address the critical question of how to design high-performance infrastructures that can be seamlessly integrated with AI for Science workflows to enhance computing efficiency. To mitigate these challenges, this dissertation proposes several innovative frameworks and use cases.

First, we introduce `MalleTrain`, a novel framework designed to optimize the utilization of underused resources on supercomputer clusters. `MalleTrain` employs dynamic resource allocation method that adapt to the varying computational needs of AI applications, thereby maximizing throughput and minimizing idle times.

Second, the dissertation proposes a large object in-memory caching system that significantly improves the data retrieval process. This system is designed to reduce latency and increase data access speed, which is crucial for processing large data sets that are typically used in scientific research. By keeping large data objects readily accessible in memory, the system ensures that computational tasks receive data input without unnecessary delays, thereby streamlining the workflow.

Third, we introduce an adaptive and scalable machine learning training framework that operates atop serverless computing infrastructure, typically hosted on public cloud platforms. This frame-

work intelligently adjusts computing resources according to varying workloads, enabling efficient and scalable utilization of serverless computing for machine learning training.

Lastly, we present WindSR, an AI application for climate research, as a demonstrative example of how AI serves as a valuable tool in scientific fields. This framework integrates data assimilation into a diffusion-based super-resolution process, achieving significant improvement compared to traditional methods.

Through these contributions, this dissertation not only tackles the pressing issues of resource inefficiency but also sets the groundwork for future innovations in AI for Science infrastructure. These frameworks and systems collectively aim to revolutionize the integration of artificial intelligence into scientific computing, making it more robust, scalable, and efficient.

7.2 Future Work

There could be a lot of potential extensions to the methodologies and results in this dissertation for both efficient usage of supercomputers and efficient data storage on cloud. We highlight some directions for future research below.

7.2.1 Large languagus model traning and serving on unfilled supercomputer nodes

For future work, several promising directions emerge from the concept of training and serving large language models on unfilled supercomputer nodes. First, a deeper investigation into adaptive scheduling algorithms could further optimize the use of supercomputing resources by dynamically

allocating idle nodes to AI tasks based on real-time demands and workloads. Additionally, exploring advanced techniques for model partitioning and distribution across multiple nodes could enhance the efficiency and scalability of both training and serving processes. Another area of potential research involves developing more robust failover and recovery systems to ensure uninterrupted service and data integrity during the deployment of models on intermittently available resources.

7.2.2 Extend in memory caching system to persistent storage system on serverless platform

Extend the in-memory caching system to include a persistent storage system on a serverless platform could dramatically improve data handling and performance scalability. This initiative would aim to integrate the high-speed access of in-memory caches with the durability of persistent storage, providing a hybrid solution that ensures data persistence without sacrificing performance. Key areas of focus would include the development of innovative data synchronization techniques to maintain consistency between volatile and non-volatile storage layers, and the implementation of smart eviction policies that optimize storage use based on real-time demand and access patterns. Additionally, exploring the potential of serverless architectures to dynamically manage and scale these storage resources could further reduce operational overhead and enhance system responsiveness. This extended system would support more complex and data-intensive applications, broadening the capabilities and applications of serverless computing platforms.

BIBLIOGRAPHY

[1] Amazon sagemaker serverless inference. https://aws.amazon.com/about-aws/whats-new/2021/12/amazon-sagemaker-serverless-inference/. [Online; accessed 08-April-2024].

[2] Aws lambda – serverless compute. https://aws.amazon.com/lambda/. [Online; accessed 08-April-2024].

[3] Aws serverless endpoints. https://docs.aws.amazon.com/sagemaker/latest/dg/serverless-endpoints.html. [Online; accessed 08-April-2024].

[4] Aws step functions. https://aws.amazon.com/step-functions/. [Online; accessed 08-April-2024].

[5] Azure functions serverless architecture – microsoft azure. https://azure.microsoft.com/en-us/services/functions/. [Online; accessed 08-April-2024].

[6] Azure machine learning – microsoft azure. https://azure.microsoft.com/en-in/services/machine-learning/. [Online; accessed 08-April-2024].

[7] Common lambda application types and use cases. https://docs.aws.amazon.com/lambda/latest/dg/applications-usecases.html. [Online; accessed 08-April-2024].

[8] Docker Hub: Container Image Library. https://www.docker.com/products/docker-hub.

[9] Facebook's Top Open Data Problems. https://research.fb.com/blog/2014/10/facebook-s-top-open-data-problems/.

[10] Google cloud ai platform. [Online; accessed 08-April-2024].

[11] Hopswork. https://www.hopsworks.ai/. [Online; accessed 08-April-2024].

[12] Increase maxconcurrency in step functions latest 'map' feature. https://docs.aws.amazon.com/lambda/latest/dg/lambda-concurrency.html. [Online; accessed 08-April-2024].

[13] Lambda quotas. https://docs.aws.amazon.com/lambda/latest/dg/gettingstarted-limits.html. [Online; accessed 08-April-2024].

[14] Machine learning inference at scale using aws serverless. https://aws.amazon.com/blogs/machine-learning/machine-learning-inference-at-scale-using-aws-serverless. [Online; accessed 08-April-2024].

[15] Machine learning on aws. https://aws.amazon.com/machine-learning/. [Online; accessed 08-April-2024].

[16] Reed-Solomon Erasure Coding in Go. https://github.com/klauspost/reedsolomon.

[17] Varnish HTTP Cache. https://varnish-cache.org/.

[18] https://aws.amazon.com/ec2/spot/, 2023. Accessed: 2023-10-23.

[19] https://cloud.google.com/spot-vms, 2023. Accessed: 2023-10-23.

[20] https://azure.microsoft.com/en-us/products/virtual-machines/spot/, 2023. Accessed: 2023-10-23.

[21] https://www.alcf.anl.gov/polaris, 2023. Accessed: 2023-10-23.

[22] https://www.top500.org/lists/top500/2023/11/, 2023. Accessed: 2023-11-15.

[23] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. Tensorflow: A system for large-scale machine learning. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 265–283, 2016.

[24] Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, et al. Gpt-4 technical report. *arXiv preprint arXiv:2303.08774*, 2023.

[25] Bilge Acun, Abhishek Gupta, Nikhil Jain, Akhil Langer, Harshitha Menon, Eric Mikida, Xiang Ni, Michael Robson, Yanhua Sun, Ehsan Totoni, et al. Parallel programming with migratable objects: Charm++ in practice. In *SC'14: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 647–658. IEEE, 2014.

[26] Naman Agarwal, Ananda Theertha Suresh, Felix Yu, Sanjiv Kumar, and H Brendan McMahan. cpsgd: communication-efficient and differentially-private distributed sgd. In *Proceedings of the 32nd International Conference on Neural Information Processing Systems*, pages 7575–7586, 2018.

[27] Istemi Ekin Akkus, Ruichuan Chen, Ivica Rimac, Manuel Stein, Klaus Satzke, Andre Beck, Paarijaat Aditya, and Volker Hilt. Sand: Towards high-performance serverless computing. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 923–935, 2018.

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

[29] Ahsan Ali, Xiaolong Ma, Syed Zawad, Paarijaat Aditya, Istemi Ekin Akkus, Ruichuan Chen, Lei Yang, and Feng Yan. Enabling scalable and adaptive machine learning training via serverless computing on public cloud. *Performance Evaluation*, 167:102451, 2025.

[30] Ahsan Ali, Riccardo Pinciroli, Feng Yan, and Evgenia Smirni. Batch: machine learning inference serving on serverless platforms with adaptive batching. In *2020 SC20: International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, pages 972–986. IEEE Computer Society, 2020.

[31] William Allcock, Paul Rich, Yuping Fan, and Zhiling Lan. Experience and practice of batch scheduling on leadership supercomputers at Argonne. In *Workshop on Job Scheduling Strategies for Parallel Processing*, pages 1–24. Springer, 2017.

[32] 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*, pages 173–182. PMLR, 2016.

[33] Ganesh Ananthanarayanan, Ali Ghodsi, Andrew Warfield, Dhruba Borthakur, Srikanth Kandula, Scott Shenker, and Ion Stoica. Pacman: Coordinated memory caching for parallel jobs. In *Presented as part of the 9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*, pages 267–280, San Jose, CA, 2012. USENIX.

[34] Ali Anwar, Mohamed Mohamed, Vasily Tarasov, Michael Littley, Lukas Rupprecht, Yue Cheng, Nannan Zhao, Dimitrios Skourtis, Amit S. Warke, Heiko Ludwig, Dean Hildebrand, and Ali R. Butt. Improving docker registry design based on production workload analysis. In *16th USENIX Conference on File and Storage Technologies (FAST 18)*, pages 265–278, Oakland, CA, 2018. USENIX Association.

[35] AWS. Whitepaper: Security overview of AWS Lambda Security and compliance best practices. March 2019.

[36] Doug Beaver, Sanjeev Kumar, Harry C. Li, Jason Sobel, and Peter Vajgel. Finding a needle in haystack: Facebook's photo storage. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, OSDI'10, pages 47–60, Berkeley, CA, USA, 2010. USENIX Association.

[37] Daniel S. Berger, Ramesh K. Sitaraman, and Mor Harchol-Balter. Adaptsize: Orchestrating the hot object memory cache in a content delivery network. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, pages 483–498, Boston, MA, 2017. USENIX Association.

[38] Anirban Bhattacharjee, Ajay Dev Chhokra, Zhuangwei Kang, Hongyang Sun, Aniruddha Gokhale, and Gabor Karsai. Barista: Efficient and scalable serverless

serving system for deep learning prediction services. In *2019 IEEE International Conference on Cloud Engineering (IC2E)*, pages 23–33. IEEE, 2019.

[39] Kaifeng Bi, Lingxi Xie, Hengheng Zhang, Xin Chen, Xiaotao Gu, and Qi Tian. Accurate medium-range global weather forecasting with 3d neural networks. *Nature*, 619(7970):533–538, 2023.

[40] Tom B Brown. Language models are few-shot learners. *arXiv preprint arXiv:2005.14165*, 2020.

[41] Sebastian Buchwald, Manuel Mohr, and Andreas Zwinkau. Malleable invasive applications. In *Software Engineering (Workshops)*, pages 123–126, 2015.

[42] Kevin Canini, Tushar Chandra, Eugene Ie, Jim McFadden, Ken Goldman, Mike Gunter, Jeremiah Harmsen, Kristen LeFevre, Dmitry Lepikhin, Tomas Lloret Llinares, et al. Sibyl: A system for large scale supervised machine learning. *Technical Talk*, 1:113, 2012.

[43] Joao Carreira, Pedro Fonseca, Alexey Tumanov, Andrew Zhang, and Randy Katz. A case for serverless machine learning. In *Workshop on Systems for ML and Open Source Software at NeurIPS*, volume 2018, 2018.

[44] Joao Carreira, Pedro Fonseca, Alexey Tumanov, Andrew Zhang, and Randy Katz. Cirrus: A serverless framework for end-to-end ml workflows. In *Proceedings of the ACM Symposium on Cloud Computing*, pages 13–24, 2019.

[45] Chen Chen, Qizhen Weng, Wei Wang, Baochun Li, and Bo Li. Fast distributed deep learning via worker-adaptive batch sizing. In *Proceedings of the ACM Symposium on Cloud Computing*, pages 521–521, 2018.

[46] Lei Chen, Fei Du, Yuan Hu, Fan Wang, and Zhibin Wang. SwinRDM: Integrate SwinRNN with Diffusion Model towards High-Resolution and High-Quality Weather Forecasting. *arXiv e-prints*, page arXiv:2306.03110, June 2023.

[47] 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. *arXiv preprint arXiv:1512.01274*, 2015.

[48] Min-Chi Chiang and Jerry Chou. Dynamoml: Dynamic resource management operators for machine learning workloads. In *CLOSER*, pages 122–132, 2021.

[49] Ethan Collins, Zachary J Lebo, Robert Cox, Christopher Hammer, Matthew Brothers, Bart Geerts, Robert Capella, and Sarah McCorkle. Forecasting high wind events in the hrrr model over wyoming and colorado. part i: Evaluation of wind speeds and gusts. *Weather and Forecasting*, 39(5):705–723, 2024.

[50] Ewa Deelman, Karan Vahi, Mats Rynge, Rajiv Mayani, Rafael Ferreira da Silva, George Papadimitriou, and Miron Livny. The evolution of the Pegasus workflow management software. *Computing in Science & Engineering*, 21(4):22–36, 2019.

[51] Travis Desell, Kaoutar El Maghraoui, and Carlos A Varela. Malleable applications for scalable high performance computing. *Cluster Computing*, 10(3):323–337, 2007.

[52] Zhenhua Di, Juan Ao, Qingyun Duan, Jin Wang, Wei Gong, Chenwei Shen, Yanjun Gan, and Zhao Liu. Improving WRF model turbine-height wind-speed forecasting using a surrogate- based automatic optimization method. *Atmospheric Research*, 226:1–16, September 2019.

[53] Chao Dong, Chen Change Loy, Kaiming He, and Xiaoou Tang. Image super-resolution using deep convolutional networks. *IEEE transactions on pattern analysis and machine intelligence*, 38(2):295–307, 2015.

[54] Caroline Draxl, Jiali Wang, Lindsay Sheridan, Chunyong Jung, Nicola Bodini, Sarah Buckhold, Connor T Aghili, Kyle Peco, Rao Kotamarthi, Andrew Kumler, et al. Wtk-led: The wind toolkit long-term ensemble dataset. Technical report, National Renewable Energy Laboratory (NREL), Golden, CO (United States), 2024.

[55] Thomas Elsken, Jan Hendrik Metzen, and Frank Hutter. Neural architecture search. In *Automated Machine Learning*, pages 63–77. Springer, 2019.

[56] Thomas Elsken, Jan Hendrik Metzen, and Frank Hutter. Neural architecture search: A survey. *Journal of Machine Learning Research*, 20(55):1–21, 2019.

[57] Stefan Falkner, Aaron Klein, and Frank Hutter. BOHB: robust and efficient hyper-parameter optimization at scale. In *International Conference on Machine Learning*, pages 1437–1446. PMLR, 2018.

[58] Dror G Feitelson and Larry Rudolph. Toward convergence in job schedulers for parallel supercomputers. In *Workshop on Job Scheduling Strategies for Parallel Processing*, pages 1–26. Springer, 1996.

[59] Hanhua Feng, Vishal Misra, and Dan Rubenstein. PBS: a unified priority-based scheduler. In *Proceedings of the 2007 ACM SIGMETRICS international conference on Measurement and Modeling of Computer Systems*, pages 203–214, 2007.

[60] Lang Feng, Prabhakar Kudva, Dilma Da Silva, and Jiang Hu. Exploring serverless

computing for neural network training. In *2018 IEEE 11th international conference on cloud computing (CLOUD)*, pages 334–341. IEEE, 2018.

[61] Fernando J Corbato. A paging experiment with the multics system. Technical Report.

[62] Peter I Frazier. A tutorial on bayesian optimization. *arXiv preprint arXiv:1807.02811*, 2018.

[63] Yu Gan, Yanqi Zhang, Dailun Cheng, Ankitha Shetty, Priyal Rathi, Nayan Katarki, Ariana Bruno, Justin Hu, Brian Ritchken, Brendon Jackson, Kelvin Hu, Meghna Pancholi, Yuan He, Brett Clancy, Chris Colen, Fukang Wen, Catherine Leung, Siyuan Wang, Leon Zaruvinsky, Mateo Espinosa, Rick Lin, Zhongling Liu, Jake Padilla, and Christina Delimitrou. An open-source benchmark suite for microservices and their hardware-software implications for cloud & edge systems. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '19, pages 3–18, New York, NY, USA, 2019. ACM.

[64] Jonathan P Gardner, John C Mather, Mark Clampin, Rene Doyon, Matthew A Greenhouse, Heidi B Hammel, John B Hutchings, Peter Jakobsen, Simon J Lilly, Knox S Long, et al. The james webb space telescope. *Space Science Reviews*, 123:485–606, 2006.

[65] Mohammad Ghaderibaneh, Caitao Zhan, and Himanshu Gupta. Deepalloc: Deep learning approach to spectrum allocation in shared spectrum systems. *IEEE Access*, 2024.

[66] Parisa Golbayani, Dan Wang, and Ionut Florescu. Application of deep neural networks to assess corporate credit rating. *arXiv preprint arXiv:2003.02334*, 2020.

[67] Priya Goyal, Piotr Dollár, Ross Girshick, Pieter Noordhuis, Lukasz Wesolowski, Aapo Kyrola, Andrew Tulloch, Yangqing Jia, and Kaiming He. Accurate, large minibatch SGD: Training ImageNet in 1 hour. *arXiv preprint arXiv:1706.02677*, 2017.

[68] M J Grubb and N I Meyer. Wind energy: Resources, systems, and regional strategies. 12 1993.

[69] Runsheng Guo, Victor Guo, Antonio Kim, Josh Hildred, and Khuzaima Daudjee. Hydrozoa: Dynamic hybrid-parallel dnn training on serverless containers. *Proceedings of Machine Learning and Systems*, 4:779–794, 2022.

[70] Jiashu He, Mingyu Derek Ma, Jinxuan Fan, Dan Roth, Wei Wang, and Alejandro Ribeiro. Give: Structured reasoning with knowledge graph inspired veracity extrapolation. *arXiv preprint arXiv:2410.08475*, 2024.

[71] 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*, pages 770–778, 2016.

[72] Muhammad Hidayat, Siska Yulia Defitri, and Haim Hilman. The impact of artificial intelligence (ai) on financial management. *Management Studies and Business Journal (PRODUCTIVITY)*, 1(1):123–129, 2024.

[73] Jonathan Ho, Ajay Jain, and Pieter Abbeel. Denoising Diffusion Probabilistic Models. *arXiv e-prints*, page arXiv:2006.11239, June 2020.

[74] Langwen Huang, Lukas Gianinazzi, Yuejiang Yu, Peter D Dueben, and Torsten Hoefler. Diffda: a diffusion model for weather-scale data assimilation. *arXiv preprint arXiv:2401.05932*, 2024.

[75] Langwen Huang, Lukas Gianinazzi, Yuejiang Yu, Peter D. Dueben, and Torsten Hoefler. DiffDA: a Diffusion Model for Weather-scale Data Assimilation. *arXiv e-prints*, page arXiv:2401.05932, January 2024.

[76] Doug Hudgeon and Richard Nichol. Machine learning for business: using amazon sagemaker and jupyter, 2024.

[77] Tyler Hunt, Congzheng Song, Reza Shokri, Vitaly Shmatikov, and Emmett Witchel. Chiron: Privacy-preserving machine learning as a service. *arXiv preprint arXiv:1803.05961*, 2018.

[78] O. A. Jaramillo and Borja M. A. Wind speed analysis in La Ventosa, Mexico: a bimodal probability distribution case. *Renewable Energy*, 29(10):1613–1630, August 2004.

[79] Julie Jebeile, Vincent Lam, and Tim Räz. Understanding climate change with statistical downscaling and machine learning. *Synthese*, 199:1877–1897, 2021.

[80] Jiawei Jiang, Shaoduo Gan, Yue Liu, Fanlin Wang, Gustavo Alonso, Ana Klimovic, Ankit Singla, Wentao Wu, and Ce Zhang. Towards demystifying serverless machine learning training. In *Proceedings of the 2021 International Conference on Management of Data*, pages 857–871, 2021.

[81] Matthew D Jones, Joseph P White, Martins Innus, Robert L DeLeon, Nikolay Simakov, Jeffrey T Palmer, Steven M Gallo, Thomas R Furlani, Michael Shower-

man, Robert Brunner, et al. Workload analysis of Blue Waters. *arXiv preprint arXiv:1703.00924*, 2017.

[82] John Jumper, Richard Evans, Alexander Pritzel, Tim Green, Michael Figurnov, Olaf Ronneberger, Kathryn Tunyasuvunakool, Russ Bates, Augustin Žídek, Anna Potapenko, et al. Highly accurate protein structure prediction with alphafold. *nature*, 596(7873):583–589, 2021.

[83] Russell Kaplan, Christopher Sauer, and Alexander Sosa. Beating atari with natural language guided reinforcement learning. *arXiv preprint arXiv:1704.05539*, 2017.

[84] Julian Kates-Harbeck, Alexey Svyatkovskiy, and William Tang. Predicting disruptive instabilities in controlled fusion plasmas through deep learning. *Nature*, 568(7753):526–531, 2019.

[85] John Kim, Wiliam J Dally, Steve Scott, and Dennis Abts. Technology-driven, highly-scalable dragonfly topology. *ACM SIGARCH Computer Architecture News*, 36(3):77–88, 2008.

[86] Alexander Kirillov, Eric Mintun, Nikhila Ravi, Hanzi Mao, Chloe Rolland, Laura Gustafson, Tete Xiao, Spencer Whitehead, Alexander C Berg, Wan-Yen Lo, et al. Segment anything. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pages 4015–4026, 2023.

[87] Christopher Kleman, Shoaib Anwar, Zhengchun Liu, Jiaqi Gong, Xishi Zhu, Austin Yunker, Rajkumar Kettimuthu, and Jiaze He. Full waveform inversion-based ultrasound computed tomography acceleration using two-dimensional convolutional neural networks. *Journal of Nondestructive Evaluation, Diagnostics and Prognostics of Engineering Systems*, 6(4):041004, 2023.

[88] Yo-Seob Lee. Analysis on trends of machine learning-as-a-service. *International Journal of Advanced Culture Technology*, 6(4):303–308, 2018.

[89] Baolin Li, Rohan Basu Roy, Tirthak Patel, Vijay Gadepally, Karen Gettings, and Devesh Tiwari. Ribbon: cost-effective and qos-aware deep learning model inference using a diverse pool of cloud computing instances. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–13, 2021.

[90] Haoyuan Li, Ali Ghodsi, Matei Zaharia, Scott Shenker, and Ion Stoica. Tachyon: Reliable, memory speed storage for cluster computing frameworks. In *Proceedings of the ACM Symposium on Cloud Computing*, SOCC '14, pages 6:1–6:15, New York, NY, USA, 2014. ACM.

[91] Li Erran Li, Eric Chen, Jeremy Hermann, Pusheng Zhang, and Luming Wang. Scaling machine learning as a service. In *International Conference on Predictive Applications and APIs*, pages 14–29. PMLR, 2017.

[92] Liam Li, Kevin Jamieson, Afshin Rostamizadeh, Ekaterina Gonina, Jonathan Ben-Tzur, Moritz Hardt, Benjamin Recht, and Ameet Talwalkar. A system for massively parallel hyperparameter tuning. *Proceedings of Machine Learning and Systems*, 2:230–246, 2020.

[93] Mu Li, Li Zhou, Zichao Yang, Aaron Li, Fei Xia, David G Andersen, and Alexander Smola. Parameter server for distributed machine learning. In *Big Learning NIPS Workshop*, volume 6, page 2, 2013.

[94] Xinyi Li, Ti Zhou, Haoyu Wang, and Man Lin. Energy-efficient computation with

dvfs using deep reinforcement learning for multi-task systems in edge computing. *arXiv preprint arXiv:2409.19434*, 2024.

[95] Jiacheng Liang, Songze Li, Bochuan Cao, Wensi Jiang, and Chaoyang He. Omnilytics: A blockchain-based secure data market for decentralized machine learning. *arXiv preprint arXiv:2107.05252*, 2021.

[96] Jiacheng Liang, Ren Pang, Changjiang Li, and Ting Wang. Model extraction attacks revisited. In *Proceedings of the 19th ACM Asia Conference on Computer and Communications Security*, pages 1231–1245, 2024.

[97] Jiacheng Liang, Zian Wang, Lauren Hong, Shouling Ji, and Ting Wang. Waterpark: A robustness assessment of language model watermarking. *arXiv preprint arXiv:2411.13425*, 2024.

[98] Fudong Lin, Summer Crawford, Kaleb Guillot, Yihe Zhang, Yan Chen, Xu Yuan, Li Chen, Shelby Williams, Robert Minvielle, Xiangming Xiao, et al. Mmst-vit: Climate change-aware crop yield prediction via multi-modal spatial-temporal vision transformer. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pages 5774–5784, 2023.

[99] Fudong Lin, Kaleb Guillot, Summer Crawford, Yihe Zhang, Xu Yuan, and Nian-Feng Tzeng. An open and large-scale dataset for multi-modal climate change-aware crop yield predictions. *arXiv preprint arXiv:2406.06081*, 2024.

[100] Fudong Lin, Xu Yuan, Yihe Zhang, Purushottam Sigdel, Li Chen, Lu Peng, and Nian-Feng Tzeng. Comprehensive transformer-based model architecture for real-world storm prediction. In *Joint European Conference on Machine Learning and Knowledge Discovery in Databases*, pages 54–71. Springer, 2023.

[101] Zhengqi Lin and Andrzej Ruszczynski. Fast dual subgradient optimization of the integrated transportation distance between stochastic kernels. *arXiv preprint arXiv:2312.01432*, 2023.

[102] Zhengqi Lin and Andrzej Ruszczyński. An integrated transportation distance between kernels and approximate dynamic risk evaluation in markov systems. *SIAM Journal on Control and Optimization*, 61(6):3559–3583, 2023.

[103] Zhengqi Lin and Andrzej Ruszczyński. Stochastic kernel approximation by transportation distance method. In *2024 IISE Annual Conference and Expo*. IISE, 2024.

[104] Fenghua Ling, Zeyu Lu, Jing-Jia Luo, Lei Bai, Swadhin K Behera, Dachao Jin, Baoxiang Pan, Huidong Jiang, and Toshio Yamagata. Diffusion model-based probabilistic downscaling for 180-year east asian climate reconstruction. *npj Climate and Atmospheric Science*, 7(1):131, 2024.

[105] M. Littley, A. Anwar, H. Fayyaz, Z. Fayyaz, V. Tarasov, L. Rupprecht, D. Skourtis, M. Mohamed, H. Ludwig, Y. Cheng, and A. R. Butt. Bolt: Towards a scalable docker registry via hyperconvergence. In *2019 IEEE 12th International Conference on Cloud Computing (CLOUD)*, pages 358–366, July 2019.

[106] Chen Liu, Matthew Amodio, Liangbo L Shen, Feng Gao, Arman Avesta, Sanjay Aneja, Jay C Wang, Lucian V Del Priore, and Smita Krishnaswamy. Cuts: A deep learning and topological framework for multigranular unsupervised medical image segmentation. In *International Conference on Medical Image Computing and Computer-Assisted Intervention*, pages 155–165. Springer, 2024.

[107] Chen Liu, Danqi Liao, Alejandro Parada-Mayorga, Alejandro Ribeiro, Marcello DiStasio, and Smita Krishnaswamy. Diffkillr: Killing and recreating diffeo-

morphisms for cell annotation in dense microscopy images. *arXiv preprint arXiv:2410.03058*, 2024.

[108] Chen Liu, Ke Xu, Liangbo L Shen, Guillaume Huguet, Zilong Wang, Alexander Tong, Danilo Bzdok, Jay Stewart, Jay C Wang, Lucian V Del Priore, et al. Imageflownet: Forecasting multiscale image-level trajectories of disease progression with irregularly-sampled longitudinal medical images. *arXiv preprint arXiv:2406.14794*, 2024.

[109] Hanxiao Liu, Karen Simonyan, and Yiming Yang. DARTS: Differentiable architecture search. *arXiv preprint arXiv:1806.09055*, 2018.

[110] Xiaoqun Liu, Jiacheng Liang, Muchao Ye, and Zhaohan Xi. Robustifying safety-aligned large language models through clean data curation. *arXiv preprint arXiv:2405.19358*, 2024.

[111] Zhengchun Liu, Ahsan Ali, Peter Kenesei, Antonino Miceli, Hemant Sharma, Nicholas Schwarz, Dennis Trujillo, Hyunseung Yoo, Ryan Coffee, Naoufal Layad, et al. Bridging data center AI systems with edge computing for actionable information retrieval. In *2021 3rd Annual Workshop on Extreme-scale Experiment-in-the-Loop Computing (XLOOP)*, pages 15–23. IEEE, 2021.

[112] Zhengchun Liu, Rajkumar Kettimuthu, Michael E Papka, and Ian Foster. FreeTrain: a framework to utilize unused supercomputer nodes for training neural networks. In *IEEE/ACM 23rd International Symposium on Cluster, Cloud and Internet Computing (CCGrid)*, pages 299–310. IEEE, 2023.

[113] Zhengchun Liu, Hemant Sharma, J-S Park, Peter Kenesei, Antonino Miceli,

Jonathan Almer, Rajkumar Kettimuthu, and Ian Foster. Braggnn: fast x-ray bragg peak analysis using deep learning. *IUCrJ*, 9(1):104–113, 2022.

[114] Andreas Lugmayr, Martin Danelljan, Andres Romero, Fisher Yu, Radu Timofte, and Luc Van Gool. Repaint: Inpainting using denoising diffusion probabilistic models. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pages 11461–11471, 2022.

[115] Zhonghao Lyu, Yuchen Li, Guangxu Zhu, Jie Xu, H Vincent Poor, and Shuguang Cui. Rethinking resource management in edge learning: A joint pre-training and fine-tuning design paradigm. *arXiv preprint arXiv:2404.00836*, 2024.

[116] Lixian Ma, En Shao, Yueyuan Zhou, and Guangming Tan. Widepipe: High-throughput deep learning inference system on a cluster of neural processing units. In *2021 IEEE 39th International Conference on Computer Design (ICCD)*, pages 563–566. IEEE, 2021.

[117] Xiaolong Ma, Feng Yan, Lei Yang, Ian Foster, Michael Papka, Zhengchun Liu, and Rajkumar Kettimuthu. Malletrain: Deep neural networks training on unfillable supercomputer nodes. In *International Conference on Performance Engineering*. ACM/SPEC, 2024.

[118] Giovanni Mariani, Andreea Anghel, Rik Jongerius, and Gero Dittmann. Predicting cloud performance for hpc applications: A user-oriented approach. In *2017 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*, pages 524–533. IEEE, 2017.

[119] Amil Merchant, Simon Batzner, Samuel S. Schoenholz, Muratahan Aykol, Gowoon

Cheon, and Ekin Dogus Cubuk. Scaling deep learning for materials discovery. *Nature*, 2023.

[120] Julia Moemken, Mark Reyers, Hendrik Feldmann, and Joaquim G. Pinto. Future Changes of Wind Speed and Wind Energy Potentials in EURO-CORDEX Ensemble Simulations. *Journal of Geophysical Research (Atmospheres)*, 123(12):6373–6389, June 2018.

[121] Anup Mohan, Harshad Sane, Kshitij Doshi, Saikrishna Edupuganti, Naren Nayak, and Vadim Sukhomlinov. Agile cold starts for scalable serverless. In *11th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 19)*, 2019.

[122] Ahuva W. Mu'alem and Dror G. Feitelson. Utilization, predictability, workloads, and user runtime estimates in scheduling the IBM SP2 with backfilling. *IEEE Transactions on Parallel and Distributed Systems*, 12(6):529–543, 2001.

[123] Ingo Müller, Renato Marroquín, and Gustavo Alonso. Lambada: Interactive data analytics on cold data using serverless cloud infrastructure. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, pages 115–130, 2020.

[124] Edward Oakes, Leon Yang, Dennis Zhou, Kevin Houck, Tyler Harter, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. Sock: Rapid task provisioning with serverless-optimized containers. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 57–70, 2018.

[125] Michael A Osborne, Roman Garnett, and Stephen J Roberts. Gaussian processes for global optimization. In *3rd international conference on learning and intelligent optimization (LION3)*, pages 1–15, 2009.

[126] Saroj Kumar Panda, Man Lin, and Ti Zhou. Energy-efficient computation offloading with dvfs using deep reinforcement learning for time-critical iot applications in edge computing. *IEEE Internet of Things Journal*, 10(8):6611–6621, 2022.

[127] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. Automatic differentiation in pytorch. 2017.

[128] 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. *arXiv preprint arXiv:1912.01703*, 2019.

[129] Tirthak Patel, Zhengchun Liu, Rajkumar Kettimuthu, Paul Rich, William Allcock, and Devesh Tiwari. Job characteristics on large-scale systems: Long-term analysis, quantification and implications. In *2020 SC20: International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, pages 1186–1202. IEEE Computer Society, 2020.

[130] Erik Lundtang Petersen, Niels Gylling Mortensen, Lars Landberg, Jørgen Højstrup, and Helmut Paul Frank. Wind power meteorology. 1997.

[131] Hieu Pham, Melody Guan, Barret Zoph, Quoc Le, and Jeff Dean. Efficient neural architecture search via parameters sharing. In *International Conference on Machine Learning*, pages 4095–4104. PMLR, 2018.

[132] Yelena L Pichugina, RM Banta, T Bonin, WA Brewer, A Choukulkar, BJ McCarty, S Baidar, Caroline Draxl, HJS Fernando, J Kenyon, et al. Spatial variability of winds

and hrrr–ncep model error statistics at three doppler-lidar sites in the wind-energy generation region of the columbia river basin. *Journal of Applied Meteorology and Climatology*, 58(8):1633–1656, 2019.

[133] 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 *OSDI*, volume 21, pages 1–18, 2021.

[134] Aurick Qiao, Willie Neiswanger, Qirong Ho, Hao Zhang, Gregory R Ganger, and Eric P Xing. Pollux: Co-adaptive cluster scheduling for goodput-optimized deep learning. *arXiv preprint arXiv:2008.12260*, 2020.

[135] Heyang Qin, Samyam Rajbhandari, Olatunji Ruwase, Feng Yan, Lei Yang, and Yuxiong He. Simigrad: Fine-grained adaptive batching for large scale training using gradient similarity measurement. *Advances in Neural Information Processing Systems*, 34:20531–20544, 2021.

[136] Heyang Qin, Syed Zawad, Yanqi Zhou, Sanjay Padhi, Lei Yang, and Feng Yan. Reinforcement-learning-empowered mlaas scheduling for serving intelligent internet of things. *IEEE Internet of Things Journal*, 7(7):6325–6337, 2020.

[137] K. V. Rashmi, Mosharaf Chowdhury, Jack Kosaian, Ion Stoica, and Kannan Ramchandran. Ec-cache: Load-balanced, low-latency cluster caching with online erasure coding. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 401–417, Savannah, GA, November 2016. USENIX Association.

[138] Esteban Real, Alok Aggarwal, Yanping Huang, and Quoc V Le. Regularized evolution for image classifier architecture search. In *Proceedings of the aaai conference on artificial intelligence*, volume 33, pages 4780–4789, 2019.

[139] Mauro Ribeiro, Katarina Grolinger, and Miriam AM Capretz. Mlaas: Machine learning as a service. In *2015 IEEE 14th International Conference on Machine Learning and Applications (ICMLA)*, pages 896–902. IEEE, 2015.

[140] DA Sachindra, Khandakar Ahmed, Md Mamunur Rashid, S Shahid, and BJC Perera. Statistical downscaling of precipitation using machine learning techniques. *Atmospheric research*, 212:240–258, 2018.

[141] Chitwan Saharia, Jonathan Ho, William Chan, Tim Salimans, David J. Fleet, and Mohammad Norouzi. Image Super-Resolution via Iterative Refinement. *arXiv e-prints*, page arXiv:2104.07636, April 2021.

[142] Victor Sanh, Lysandre Debut, Julien Chaumond, and Thomas Wolf. Distilbert, a distilled version of bert: smaller, faster, cheaper and lighter. *arXiv preprint arXiv:1910.01108*, 2019.

[143] Klaus Satzke, Istemi Ekin Akkus, Ruichuan Chen, Ivica Rimac, Manuel Stein, Andre Beck, Paarijaat Aditya, Manohar Vanga, and Volker Hilt. Efficient gpu sharing for serverless workflows. In *Proceedings of the 1st Workshop on High Performance Serverless Computing*, HiPS '21, 2021.

[144] Justin T Schoof. Statistical downscaling in climatology. *Geography Compass*, 7(4):249–265, 2013.

[145] Lucia Schuler, Somaya Jamil, and Niklas Kühl. Ai-based resource allocation: Re-

inforcement learning for adaptive auto-scaling in serverless environments. In *2021 IEEE/ACM 21st International Symposium on Cluster, Cloud and Internet Computing (CCGrid)*, pages 804–811. IEEE, 2021.

[146] Alexander Sergeev and Mike Del Balso. Horovod: Fast and easy distributed deep learning in TensorFlow. *arXiv preprint arXiv:1802.05799*, 2018.

[147] Jawaad Sheriff, Peineng Wang, Peng Zhang, Ziji Zhang, Yuefan Deng, and Danny Bluestein. In vitro measurements of shear-mediated platelet adhesion kinematics as analyzed through machine learning. *Annals of biomedical engineering*, 49(12):3452–3464, 2021.

[148] Samuel L Smith, Pieter-Jan Kindermans, Chris Ying, and Quoc V Le. Don't decay the learning rate, increase the batch size. *arXiv preprint arXiv:1711.00489*, 2017.

[149] Yang Song, Jascha Sohl-Dickstein, Diederik P Kingma, Abhishek Kumar, Stefano Ermon, and Ben Poole. Score-based generative modeling through stochastic differential equations. *arXiv preprint arXiv:2011.13456*, 2020.

[150] Karen Stengel, Andrew Glaws, Dylan Hettinger, and Ryan N King. Adversarial super-resolution of climatological wind and solar data. *Proceedings of the National Academy of Sciences*, 117(29):16805–16815, 2020.

[151] Pei Sun, Henrik Kretzschmar, Xerxes Dotiwalla, Aurelien Chouard, Vijaysai Patnaik, Paul Tsui, James Guo, Yin Zhou, Yuning Chai, Benjamin Caine, et al. Scalability in perception for autonomous driving: Waymo open dataset. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 2446–2454, 2020.

[152] Ahmad P Tafti, Eric LaRose, Jonathan C Badger, Ross Kleiman, and Peggy Peissig. Machine learning-as-a-service and its application to medical informatics. In *International Conference on Machine Learning and Data Mining in Pattern Recognition*, pages 206–219. Springer, 2017.

[153] Jianping Tang, Xiaorui Niu, Shuyu Wang, Hongxia Gao, Xueyuan Wang, and Jian Wu. Statistical downscaling and dynamical downscaling of regional climate in china: Present climate evaluations and future climate projections. *Journal of Geophysical Research: Atmospheres*, 121(5):2110–2129, 2016.

[154] Stacey Truex, Ling Liu, Mehmet Emre Gursoy, Lei Yu, and Wenqi Wei. Demystifying membership inference attacks in machine learning as a service. *IEEE Transactions on Services Computing*, 2019.

[155] Iulia Turc, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Well-read students learn better: On the importance of pre-training compact models. *arXiv preprint arXiv:1908.08962v2*, 2019.

[156] David D Turner, Harvey Cutler, Martin Shields, Rebecca Hill, Brad Hartman, Yuchen Hu, Tao Lu, and Hwayoung Jeon. Evaluating the economic impacts of improvements to the high-resolution rapid refresh (hrrr) numerical weather prediction model. *Bulletin of the American Meteorological Society*, 103(2):E198–E211, 2022.

[157] Sathish S Vadhiyar and Jack J Dongarra. SRS: A framework for developing malleable and migratable parallel applications for distributed systems. *Parallel Processing Letters*, 13(02):291–312, 2003.

[158] Ao Wang, Jingyuan Zhang, Xiaolong Ma, Ali Anwar, Lukas Rupprecht, Dimitrios Skourtis, Vasily Tarasov, Feng Yan, and Yue Cheng. {InfiniCache}: exploiting

ephemeral serverless functions to build a {cost-effective} memory cache. In *18th USENIX conference on file and storage technologies (FAST 20)*, pages 267–281, 2020.

[159] Dan Wang, Zhi Chen, Ionuţ Florescu, and Bingyang Wen. A sparsity algorithm for finding optimal counterfactual explanations: Application to corporate credit rating. *Research in International Business and Finance*, 64:101869, 2023.

[160] Dan Wang, Tianrui Wang, and Ionuţ Florescu. Is image encoding beneficial for deep learning in finance? *IEEE Internet of Things Journal*, 9(8):5617–5628, 2020.

[161] Hao Wang, Di Niu, and Baochun Li. Distributed machine learning with a serverless architecture. In *IEEE INFOCOM 2019-IEEE Conference on Computer Communications*, pages 1288–1296. IEEE, 2019.

[162] Haoyu Wang, Xinyi Li, Ti Zhou, and Man Lin. Data-driven software-based power estimation for embedded devices. *arXiv preprint arXiv:2407.02764*, 2024.

[163] Jiali Wang, Zhengchun Liu, Ian Foster, Won Chang, Rajkumar Kettimuthu, and V Rao Kotamarthi. Fast and accurate learned multiresolution dynamical downscaling for precipitation. *Geoscientific Model Development*, 14(10):6355–6372, 2021.

[164] Liang Wang, Mengyuan Li, Yinqian Zhang, Thomas Ristenpart, and Michael Swift. Peeking behind the curtains of serverless platforms. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 133–146, Boston, MA, 2018. USENIX Association.

[165] Peineng Wang, Jawaad Sheriff, Peng Zhang, Yuefan Deng, and Danny Bluestein. A multiscale model for shear-mediated platelet adhesion dynamics: correlating in

silico with in vitro results. *Annals of Biomedical Engineering*, 51(5):1094–1105, 2023.

[166] Xintao Wang, Ke Yu, Shixiang Wu, Jinjin Gu, Yihao Liu, Chao Dong, Yu Qiao, and Chen Change Loy. Esrgan: Enhanced super-resolution generative adversarial networks. In *Proceedings of the European conference on computer vision (ECCV) workshops*, pages 0–0, 2018.

[167] Jiasi Weng, Jian Weng, Chengjun Cai, Hongwei Huang, and Cong Wang. Goldengrain: Building a secure and decentralized model marketplace for mlaas. *arXiv e-prints*, pages arXiv–2011, 2020.

[168] Fei Xu, Yiling Qin, Li Chen, Zhi Zhou, and Fangming Liu. λdnn: Achieving predictable distributed dnn training with serverless architectures. *IEEE Transactions on Computers*, 71(2):450–463, 2021.

[169] Ben Yang, Yun Qian, Larry K. Berg, Po-Lun Ma, Sonia Wharton, Vera Bulaevskaya, Huiping Yan, Zhangshuan Hou, and William J. Shaw. Sensitivity of Turbine-Height Wind Speeds to Parameters in Planetary Boundary-Layer and Surface-Layer Schemes in the Weather Research and Forecasting Model. *Boundary-Layer Meteorology*, 162(1):117–142, January 2017.

[170] Yuanshun Yao, Zhujun Xiao, Bolun Wang, Bimal Viswanath, Haitao Zheng, and Ben Y Zhao. Complexity vs. performance: empirical analysis of machine learning as a service. In *Proceedings of the 2017 Internet Measurement Conference*, pages 384–397, 2017.

[171] Qing Ye, Yuhao Zhou, Mingjia Shi, Yanan Sun, and Jiancheng Lv. Dbs: Dy-

namic batch size for distributed deep neural network training. *arXiv preprint arXiv:2007.11831*, 2020.

[172] Abbas Yeganeh-Bakhtiary, Hossein EyvazOghli, Naser Shabakhty, Bahareh Kamranzad, and Soroush Abolfathi. Machine learning as a downscaling approach for prediction of wind characteristics under future climate change scenarios. *Complexity*, 2022(1):8451812, 2022.

[173] Jun Yi, Chengliang Zhang, Wei Wang, Cheng Li, and Feng Yan. Not all explorations are equal: Harnessing heterogeneous profiling cost for efficient mlaas training. In *2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 419–428. IEEE, 2020.

[174] Chris Ying, Aaron Klein, Eric Christiansen, Esteban Real, Kevin Murphy, and Frank Hutter. NAS-Bench-101: Towards reproducible neural architecture search. In *International Conference on Machine Learning*, pages 7105–7114. PMLR, 2019.

[175] Andy B Yoo, Morris A Jette, and Mark Grondona. Slurm: Simple Linux utility for resource management. In *Workshop on job scheduling strategies for parallel processing*, pages 44–60. Springer, 2003.

[176] Haihang You and Hao Zhang. Comprehensive workload analysis and modeling of a petascale supercomputer. In *Workshop on Job Scheduling Strategies for Parallel Processing*, pages 253–271. Springer, 2012.

[177] Yang You, Zhao Zhang, Cho-Jui Hsieh, James Demmel, and Kurt Keutzer. ImageNet training in minutes. In *47th International Conference on Parallel Processing*, pages 1–10, 2018.

[178] Yinghao Yu, Renfei Huang, Wei Wang, Jun Zhang, and Khaled Ben Letaief. Sp-cache: Load-balanced, redundancy-free cluster caching with selective partition. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis*, SC '18. IEEE Press, 2018.

[179] Caitao Zhan, Mohammad Ghaderibaneh, Pranjal Sahu, and Himanshu Gupta. Deepmtl: Deep learning based multiple transmitter localization. In *2021 IEEE 22nd International Symposium on a World of Wireless, Mobile and Multimedia Networks (WoWMoM)*, pages 41–50. IEEE, 2021.

[180] Caitao Zhan, Mohammad Ghaderibaneh, Pranjal Sahu, and Himanshu Gupta. Deepmtl pro: Deep learning based multiple transmitter localization and power estimation. *Pervasive and Mobile Computing*, 82:101582, 2022.

[181] Chengliang Zhang, Minchen Yu, Wei Wang, and Feng Yan. Mark: Exploiting cloud services for cost-effective, slo-aware machine learning inference serving. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 1049–1062, 2019.

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

[183] Huaizheng Zhang, Yuanming Li, Yizheng Huang, Yonggang Wen, Jianxiong Yin, and Kyle Guan. Mlmodelci: An automatic cloud platform for efficient mlaas. In *Proceedings of the 28th ACM International Conference on Multimedia*, pages 4453–4456, 2020.

[184] Jingyuan Zhang, Ao Wang, Xiaolong Ma, Benjamin Carver, Nicholas John Newman, Ali Anwar, Lukas Rupprecht, Dimitrios Skourtis, Vasily Tarasov, Feng Yan,

et al. Infinistore: Elastic serverless cloud storage. *arXiv preprint arXiv:2209.01496*, 2022.

[185] Yang Zhang, Fangzhou Xu, Erwin Frise, Siqi Wu, Bin Yu, and Wei Xu. Datalab: a version data management and analytics system. In *Proceedings of the 2nd International Workshop on BIG Data Software Engineering*, pages 12–18, 2016.

[186] Ziji Zhang, Peng Zhang, Peineng Wang, Jawaad Sheriff, Danny Bluestein, and Yuefan Deng. Rapid analysis of streaming platelet images by semi-unsupervised learning. *Computerized Medical Imaging and Graphics*, 89:101895, 2021.

[187] Zikai Zhang and Rui Hu. Byzantine-robust federated learning with variance reduction and differential privacy. In *2023 IEEE Conference on Communications and Network Security (CNS)*, pages 1–9. IEEE, 2023.

[188] Shuai Zhao, Manoop Talasila, Guy Jacobson, Cristian Borcea, Syed Anwar Aftab, and John F Murray. Packaging and sharing machine learning models via the acumos ai open platform. In *2018 17th IEEE International Conference on Machine Learning and Applications (ICMLA)*, pages 841–846. IEEE, 2018.

[189] Qihua Zhou, Song Guo, Jun Pan, Jiacheng Liang, Jingcai Guo, Zhenda Xu, and Jingren Zhou. Pass: Patch automatic skip scheme for efficient on-device video perception. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 2024.

[190] Qihua Zhou, Song Guo, Jun Pan, Jiacheng Liang, Zhenda Xu, and Jingren Zhou. Pass: patch automatic skip scheme for efficient real-time video perception on edge devices. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 37, pages 3787–3795, 2023.

[191] Ti Zhou and Man Lin. Deadline-aware deep-recurrent-q-network governor for smart energy saving. *IEEE Transactions on Network Science and Engineering*, 9(6):3886–3895, 2021.

[192] Ti Zhou and Man Lin. Cpu frequency scheduling of real-time applications on embedded devices with temporal encoding-based deep reinforcement learning. *Journal of Systems Architecture*, 142:102955, 2023.

[193] Ti Zhou, Haoyu Wang, Xinyi Li, and Man Lin. Profiling and understanding cpu power management in linux. In *2023 IEEE Smart World Congress (SWC)*, pages 1–8. IEEE, 2023.

[194] Barret Zoph, Vijay Vasudevan, Jonathon Shlens, and Quoc V Le. Learning transferable architectures for scalable image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 8697–8710, 2018.