FasDL: An Efficient Serverless-Based Training Architecture With Communication Optimization and Resource Configuration

Xinglei Chen, Zinuo Cai, Hanwen Zhang, Ruhui Ma, *Member, IEEE*, and Rajkumar Buyya, *Fellow, IEEE*

***Abstract*—Deploying distributed training workloads of deep learning models atop serverless architecture alleviates the burden of managing servers from deep learning practitioners. However, when supporting deep model training, the current serverless architecture faces the challenges of inefficient communication patterns and rigid resource configuration that incur subpar and unpredictable training performance. In this paper, we propose FasDL, an efficient serverless-based deep learning training architecture to solve these two challenges. FasDL adopts a novel training framework `K-REDUCE` to release the communication overhead and accelerate the training. Additionally, FasDL builds a lightweight mathematical model for `K-REDUCE` training, offering predictable performance and supporting subsequent resource configuration. It achieves the optimal resource configuration by formulating an optimization problem related to system-level and application-level parameters and solving it with a pruning-based heuristic search algorithm. Extensive experiments on AWS Lambda verify a prediction accuracy over 94% and demonstrate performance and cost advantages over the state-of-art architecture LambdaML by up to 16.8% and 28.3% respectively.**

***Index Terms*—Serverless computing, deep learning, communication optimization, resource configuration.**

I. Introduction

THE past decade has witnessed the successful development of serverless computing [1], [2], [3], which started with the release of AWS Lambda [4], the first commercial serverless service, in 2014. Serverless computing, also known as Function as a Service (FaaS), has many advantages over conventional cloud computing paradigms like Infrastructure as a Service (IaaS) [5], Platform as a Service (PaaS) [6] and Software as a Service (SaaS) [7]. For cloud customers, serverless computing eliminates tedious operation and maintenance (O&M) because of its automatic scalability to handle burst requests. For cloud vendors, since serverless computing transfers the responsibility of system operation from cloud customers to vendors, they have more superiority in system scheduling to improve the utilization of the underlying infrastructures. Therefore, serverless computing has attracted attention from industry and academia and has wide application in all fields like video processing [8] and high-performance computation [9].

Due to the automatic scalability and cost-effectiveness, it has also attracted widespread attention from deep learning practitioners to deploy their training on serverless platforms [10]. Serverless computing benefits deep model training in two aspects. The lightweight nature of serverless functions accelerates the training procedure by effortlessly running a multitude of serverless functions in parallel, which is especially beneficial during hyperparameter tuning. Moreover, serverless computing provides a cost-efficient methodology for deploying deep learning training workflows because deep learning practitioners are billed only by the used resources with the "pay-as-you-go" billing mode. A multitude of works have arisen to optimize serverless-based deep learning training workflows [11], [12], [13], [14].

Since LambdaML [14] provides a comprehensive characterization of serverless-based deep model training, we follow its footstep and identify two critical challenges in practice.

1) Excessive Communication Overhead: The first challenge arises from the lack of peer-to-peer communication [1] because of the stateless nature of serverless computing. The absence of peer-to-peer communication in serverless computing impedes the implementation of traditional communication patterns such as Ring AllReduce and Decentralized Parallel [15] which depend on stable connections between workers. LambdaML implements two communication patterns, AllReduce and ScatterReduce, with external storage as the communication channel to address this challenge. However, the communication overhead can be at most 6× over the computation latency [13].

Received 25 March 2024; revised 24 September 2024; accepted 29 September 2024. Date of publication 23 October 2024; date of current version 20 January 2025. This work was supported in part by Shanghai Key Laboratory of Scalable Computing and Systems, in part by the National Key Laboratory of Ship Structural Safety, and in part by the Eighth Research Institute of China Aerospace Science and Technology Group Company, Ltd., under Grant USCAST2023-17 and Grant USCAST2023-21. Recommended for acceptance by W. Li. *(Xinglei Chen and Zinuo Cai contributed equally to this work.) (Corresponding author: Ruhui Ma.)*

Xinglei Chen, Zinuo Cai, Hanwen Zhang, and Ruhui Ma are with the School of Electronic Information and Electrical Engineering, Shanghai Jiao Tong University, Shanghai 200240, China (e-mail: chen1556805501@sjtu.edu.cn; kingczn1314@sjtu.edu.cn; rowerman@sjtu.edu.cn; ruhuima@sjtu.edu.cn).

Rajkumar Buyya is with the School of Computing and Information Systems, University of Melbourne, Melbourne, VIC 3010, Australia (e-mail: rbuyya@unimelb.edu.au).

Digital Object Identifier 10.1109/TC.2024.3485202

2) Challenging Parameter Configuration: The second challenge is that although serverless computing can alleviate O&M efforts, deep learning practitioners are still required to configure system-level and application-level parameters to balance training latency, model accuracy and monetary cost [16]. Our experiments in Section II-C reveal that the misconfiguration of dual-level parameters can result in either slow training or high cost.

To address the aforementioned two critical challenges, we propose **FasDL**, an efficient serverless-based deep learning training architecture, to enhance the performance of training deep learning models on serverless platforms and facilitate automatic configuration to alleviate the effort of finding the optimal parameter configuration.

In response to the first challenge, we revisit the design of the communication patterns AllReduce and ScatterReduce in LambdaML, finding that the rigid selection of the number of aggregators leads to the high communication overhead. AllReduce selects 1 worker as the aggregator, while all the workers participate in aggregation in ScatterReduce, which are two extreme cases in terms of the number of aggregators. We conduct experiments to compare the communication performance as we change the number of aggregators from 1 to that of workers in Section II-B. Our experimental results reveal that increasing the number of aggregators can not necessarily alleviate the communication overhead. The performance gain becomes negligible as the number of aggregators approaches that of the workers, and can even lead to an increase in communication overhead due to the decay of throughput caused by excessive partitioning of the model parameters. Based on the observation, we design an adaptive communication pattern named `K-REDUCE` in Section IV where we select the optimal K aggregators from all the workers for model aggregation to ensure the communication quality. Furthermore, we observe that workers other than the aggregators (referred to as non-aggregators) are idle during the aggregation step. Therefore, we design an unequal dataset partitioning scheme and a Hybrid Asynchronous Parallel protocol, as detailed in Section IV, to utilize the idle time of non-aggregators to further accelerate training.

Regarding the second challenge, we identify that we can predict the performance with unseen parameter configurations after characterizing serverless services and carefully formulating their relationship. Therefore, we develop a mathematical model between the dual-level parameters and overall performance in terms of training time, convergence efficiency, and monetary cost in Section V. Subsequently, we efficiently solve the configuration optimization with a pruning-based heuristic algorithm in Section VI.

Our contributions are as follows:

- Firstly, to address the issue of low efficiency in communication patterns, we propose `K-REDUCE` to mitigate the communication overhead in distributed learning (Section IV) with optimal aggregator number. It also exploits unequal dataset partitioning scheme and Hybrid Asynchronous Parallel (HAP) protocol to further expedite the training.

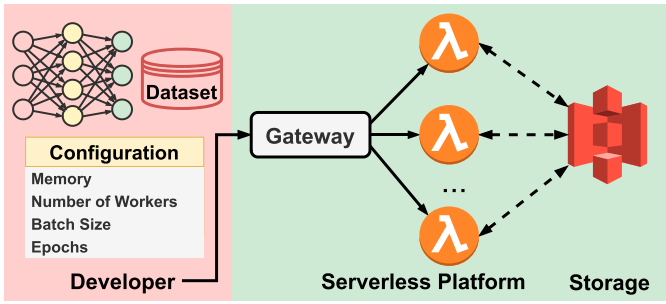


Fig. 1. Typical serverless-based distributed model training on AWS Lambda.

- Secondly, to overcome the difficulty in providing cost-efficient configurations, we formulate a model that relates the dual-level parameters with overall performance (Section V) to offer predictable performance. Then, we design a pruning-based heuristic algorithm to automate the parameter configuration for the `K-REDUCE` (Section VI).
- Finally, we implement a prototype of **FasDL** to realize the `K-REDUCE` framework atop AWS Lambda and AWS S3 (Section VII). Extensive experiments prove that **FasDL** can predict the training performance within 5.4% error, speed up the training by up to 16.8% and reduce the cost by up to 28.3% compared with LambdaML (Section VIII).

II. Background and Motivation

A. Serverless-Based Distributed Training

We show a typical workflow of training deep learning models on serverless computing platforms in Fig. 1. On the developer side, besides defining model architectures and pre-processing training datasets, developers also prepare a configuration file to describe the necessary system-level and application-level parameters. The system-level parameters include the number of serverless functions and their memory configuration that will influence the execution performance of the function, and the application-level ones are more relevant to the training procedure like the batch size. The developers trigger the function through the Gateway, and the serverless platform will execute functions, each corresponding to a worker in distributed training, in parallel. During the execution of the distributed training procedure, the functions will communicate with each other through storage services because of the lack of a peer-to-peer communication mechanism [1]. Therefore, although serverless computing can benefit distributed training because of its high lightweight parallelism, it also puts forward challenges of inter-function communication overhead and the dual-level parameter configuration.

B. Inter-Function Communication Overhead

There are two common implementation strategies to solve the lack of inter-function communication channels during the distributed training on serverless platforms, server-based and storage-based communication channel. The former, represented by λDNN [12], launches a long-live server with a fixed communication address, allowing serverless functions to communicate

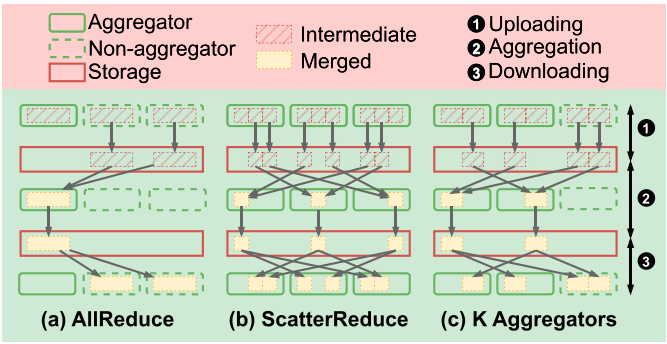


Fig. 2. Three serverless communication patterns under storage-based channels.

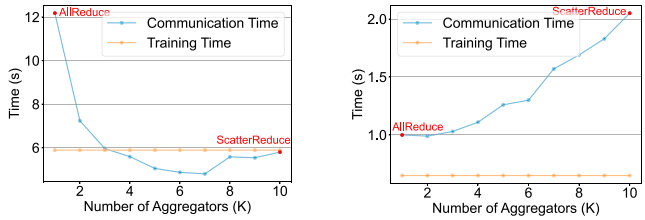


Fig. 3. Time breakdown in one iteration.

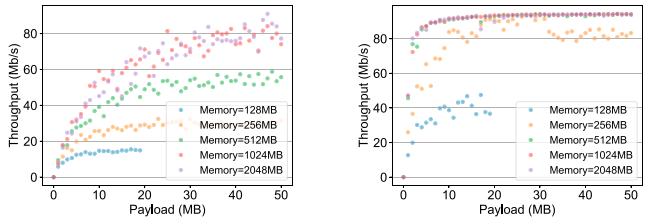


Fig. 4. Throughput between lambda functions and S3.

with it through gRPC or HTTP. However, such an implementation violates the core principles of serverless architectures—namely, the absence of server management and automatic scaling—by requiring additional PaaS-based cloud services and lacking support for automatic scalability to accommodate increasing communication traffic. The latter, represented by LambdaML [14], employs an external persistent storage service as the communication channel and uses lambda functions as aggregators to perform parameter aggregation. Such an implementation provides good scalability and high parallelism while adhering to deployment-free semantics. However, it also results in higher communication overhead. FuncPipe [13] points out that the communication time can be up to six times the computation time and proposes an optimization scheme based on model parallelism. This scheme reduces communication overhead through model partitioning and pipeline processing. In this work, we re-explore the parallelism in the communication under data parallelism by adjusting data partitioning and synchronization methods to reduce communication overhead and further accelerate training. Our work is orthogonal to FuncPipe, providing a different solution to the high communication overhead in storage-based communication.

With the storage-based communication channel, communication involves three steps: uploading, aggregation, and downloading. Based on their participation in the aggregation step, workers are categorized into two roles: aggregators and non-aggregators. Fig. 2 illustrates the process of a single communication cycle across three communication patterns, varying the number of aggregators involved. AllReduce and ScatterReduce are commonly adopted in distributed model training, and Fig. 2(a) and 2(b) show their implementation when applied in serverless computing [14]. Assuming there are W workers when training the model, AllReduce selects 1 worker as an aggregator for parameter aggregation. All the other workers (*i.e.,* non-aggregators) upload their intermediate parameters to the external storage, and then the aggregator collects them from the storage and handles the parameter aggregation. Finally, the aggregator stores the merged parameters in the storage and broadcasts them to the other workers. During the aggregation step, non-aggregators neither perform computations nor engage in communication; instead, they continuously poll to determine whether the aggregation is complete. Regarding ScatterReduce, all the W workers participate in the aggregation step and the model parameters are split into W shards accordingly. Each worker aggregates a shard of parameters, obtains other merged shards and recombines all into the updated model parameters. As the number of aggregators in AllReduce and ScatterReduce is 1 and W, respectively, we generalize by choosing an arbitrary number K ($1 \leq K \leq W$) of aggregators, as shown in Fig. 2(c). Specifically, we choose K aggregators for parameters aggregation and split the model parameters into K shards accordingly.

To observe the impact of aggregators on communication time, we conduct experiments using AWS Lambda as a serverless platform and AWS S3 as an external storage. Fig. 3 illustrates the overall trends of training and communication time within a single iteration as we changed the aggregator's number from 1 to W. To investigate the impact on models of different complexities, we choose ResNet50 [17] and SqueezeNet [18] as training models.

For ResNet50, the trend in communication overhead as K increases resembles a 'U' shape—high at both ends and lower in the middle. When K is small, a single aggregator in AllReduce becomes the communication bottleneck due to the low degree of parallelism, leading to high communication overhead. With the increase of aggregators, there is a sharp decrease in overhead, reaching a minimum at an intermediate K value. However, as K gets closer to W, the decrease slows and eventually reverses into a small increase due to the underutilization of the function's bandwidth. We test the throughput of Lambda functions with different memory quotas. Fig. 4 illustrates the throughput degradation as the payload transferred between Lambda functions and S3 gets smaller. When K is large, the model parameter shard during transmission is small, resulting in a decline in the function's throughput. For smaller models like SqueezeNet (4.71MB), opting for multiple aggregators to aggregate model

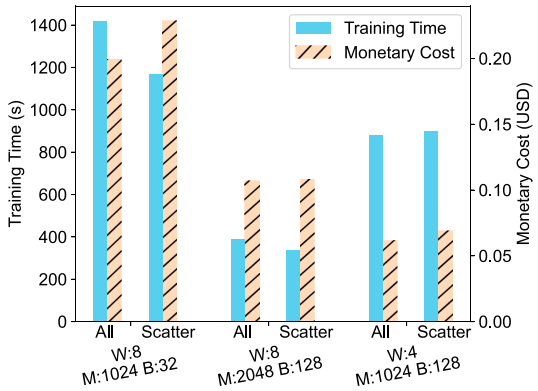


Fig. 5. Comparison on training time and monetary cost in serverless environment with different configurations (number of workers W, memory quota M and batch size B).

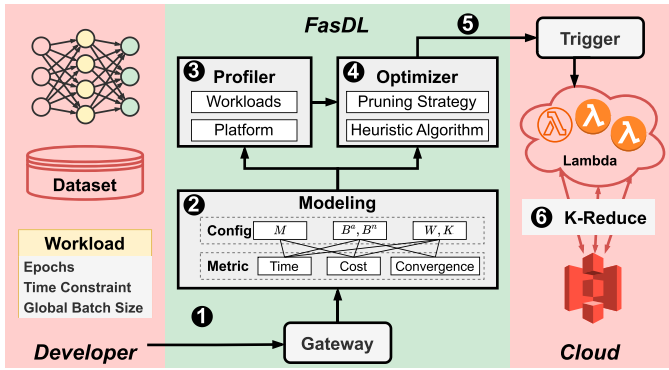


Fig. 6. System architecture and workflow.

parameters results in a more pronounced decline in function throughput. In this scenario, the advantages of parallel aggregation do not outweigh the drawbacks associated with low throughput. Consequently, choosing 1 or 2 aggregators minimizes the communication time for SqueezeNet. Overall, with an inappropriate choice of aggregator number K, the communication time would greatly exceed the training time.

C. Dual-Level Parameter Configuration

While serverless computing reduces O&M overhead, the configuration of system-level and application-level parameters is still essential for optimizing model performance and cost efficiency for deep learning practitioners. The system-level parameters of deep learning applications include the number of CPU cores [19], [20] and network bandwidth [13], [21] that influence the training performance and communication latency, respectively. Within the leading commercial serverless computing platforms, such as AWS Lambda, the allocation of these resources are automatically and linearly scaled based on the user-defined memory quota [22], which is one critical factor to the billing. Another system-level parameter is the number of workers, *i.e.,* the parallel functions in AWS Lambda. The application-level parameters include the batch size that influences both training speed and model performance.

We carry out training on ResNet50 with three sets of random parameter configurations for the same amount of data samples. As shown in Fig. 5, training with 8 workers, 1024MB memory and a 32 batch size takes more than twice the time and cost than that of training with 8 workers, 2048MB memory and a 128 batch size. The latter configuration shows inverse performance with the configuration of 4 workers, 1024MB memory and a 128 batch size on time and cost, *i.e.,* the training time doubles however the cost is only half. In conclusion, misconfiguration of dual-level parameters by developers can lead to unpredictable and excessively high training time and cost, posing a barrier to conducting training tasks on serverless computing platforms.

III. System Architecture

We propose **FasDL**, an efficient serverless-based deep learning training architecture to mitigate communication overhead and achieve automatic parameter configuration. **FasDL** consists of a K-REDUCE training framework, a Modeling module, a Profiler module, and an Optimizer module. The workflow of **FasDL** is shown in Fig. 6. Initially, deep learning practitioners ❶ submit their workloads through Gateway, including deep models, training datasets, training epochs and constraints. Compared with the existing platform in Fig. 1, **FasDL** aims to adaptively determine the parameter configuration within the time and convergence constraints, and the least monetary cost. **FasDL** firstly ❷ conducts system modeling, including end-to-end training time, monetary cost and convergence efficiency in terms of the workloads. The parameter configurations of the workload include memory quota M, number of workers W, batch size B, and number of aggregators K. Then **FasDL** ❸ employs a profiler to determine the workload-specific and platform-specific coefficients. The workload-specific coefficients characterize the relationship between the training speed of the given model and the dual-level parameter configuration. The platform-specific coefficients characterize the network capabilities provided by the platform. The modeling and profiling results will transfer to the optimizer to ❹ formulate the optimization problem, determine the parameter search range and determine parameter configurations based on the pruning-based heuristic algorithm. Finally, **FasDL** will ❺ trigger serverless functions as aggregators and non-aggregators respectively and allocate datasets of varying sizes. The workers ❻ carry out distributed training according to the K-REDUCE framework and use external storage as the communication channel.

IV. K-REDUCE for Serverless-Based Training

We design a new training framework, K-REDUCE, to enhance the performance of model training based on serverless computing. We adopt a novel communication pattern to reduce excessive communication overhead, and simultaneously, we design a suitable data partitioning scheme and synchronization protocol to further accelerate the training process. Fig. 7 illustrates the serverless-based deep learning training workflow with the framework of K-REDUCE.

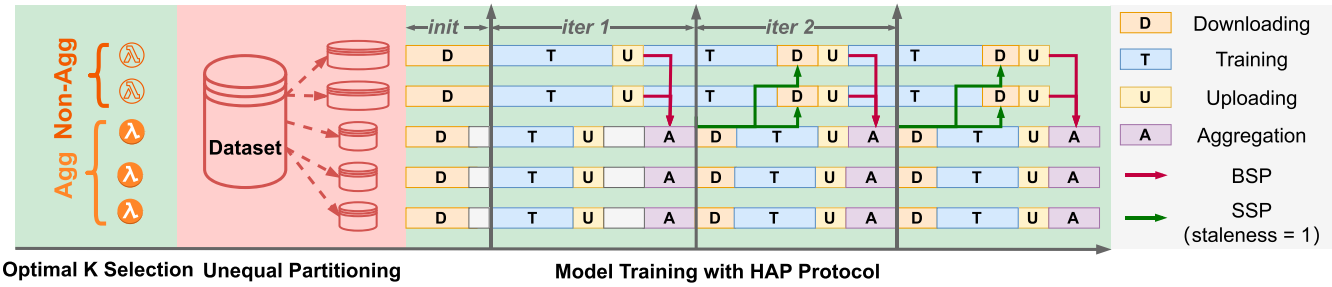


Fig. 7. Workflow of K-REDUCE for serverless-based distributed training.

A. Optimal K-Aggregator Selection for Communication Optimization

With the observation in Section II-B, we select optimal K ($1 \leq K \leq W$) workers to serve as aggregators and the other $W - K$ workers as non-aggregators to minimize the communication overhead. The communication pattern with K aggregators is shown in Fig. 2(c). To select the optimal number of aggregators K, we model the relationship between communication overhead and the number of aggregators in Section V, and design a heuristic algorithm for the efficient search of K in Section VI-B.

B. Enhanced Data Partitioning and Synchronization for Accelerated Training

As shown in Fig. 2, the non-aggregators are idle in the aggregation step. We leverage this idle time of non-aggregators to accelerate the training procedure. However, implementing such a design is far from trivial, as it could result in asynchronous training processes among workers as well as issues with asynchronous weight updates, which would invalidate conventional training data partitioning and synchronization protocols. Therefore, we design an unequal dataset partitioning scheme and a Hybrid Asynchronous Parallel protocol. The former mechanism precisely partitions additional data to non-aggregators, and the latter flexibly implements distinct synchronization protocols for different roles of workers. Under these design considerations, we effectively utilize the idle time of non-aggregators during the communication to accelerate the training. Only the aggregators experience brief idle periods during the initialization due to loading fewer data samples, and during the first iteration for synchronization.

a) Unequal Dataset Partitioning: To fully utilize the CPU time of non-aggregators in aggregation, we devise unequal dataset partitioning that precisely assigns additional training samples to the non-aggregators. The unequal partitioning should meet the alignment constraints on the following two aspects. First, the total time of non-aggregators should not exceed that of aggregators in each iteration; otherwise, it would cause idle waiting of aggregators. Second, the total number of iterations in an epoch should be the same for the two roles to ensure the aggregators and non-aggregators finish the training within a similar period. Therefore, we determine the exact batch size for non-aggregators in Section VI and partition the dataset ensuring that the ratio of the total training samples between aggregators and non-aggregators is consistent with their respective batch sizes, thereby maintaining an equal number of iterations across roles.

b) Hybrid Asynchronous Parallel: The iterative nature of deep learning training requires synchronization protocols to maintain progress consistency between multiple workers. Bulk Synchronous Parallel (BSP) [23], [24] adopted by ScatterReduce and AllReduce requires each worker to wait for the others in each iteration, ensuring high consistency but introducing lengthy overhead. In K-REDUCE, workers are divided into two distinct roles, aggregators and non-aggregators, each with different synchronization requirements. Aggregators are responsible for aggregating parameters after each training iteration, necessitating higher synchronization fidelity. In contrast, non-aggregators focus primarily on training and submit parameters at specific points for updates, thus requiring less stringent synchronization. Therefore, K-REDUCE introduces a new synchronization protocol, Hybrid Asynchronous Parallel (HAP). The aggregators employ the BSP protocol for inter-aggregator communication to keep the global model consistent. The protocol between aggregators and non-aggregators is Stale Synchronous Parallel (SSP) [25]. The SSP protocol allows workers to asynchronously retrieve stale parameters and sets a staleness threshold to ensure convergence performance. To alleviate the effects of asynchronous updates on weights, we fix the staleness of non-aggregators at 1, *i.e.,* non-aggregators retrieve the merged parameters of the previous iteration and immediately transition to the subsequent iteration without any extra waiting.

V. Modeling of K-REDUCE

The dual-level parameters of serverless distributed training include the memory quota M, the number of total workers W and the batch size B. With the K-REDUCE (Section IV), due to the categorization of the worker roles, an additional system-level parameter, the number of aggregators K, is introduced. Meanwhile, we denote the batch sizes for aggregators and non-aggregators as B^a and B^n, respectively. The configuration of the above parameters affects the training performance in the following three aspects: end-to-end training time, monetary cost and convergence efficiency. The key notations in our modeling of K-REDUCE are summarized in Table I.

TABLE I
KEY NOTATION IN THE MODELING OF K-REDUCE

Notation	Definition
S_m	Size of the total model parameters
S_s	Size of the model parameter shard in transmission
S_d	Size of the training dataset
D	Number of training data samples
I, E	Number of training iterations per epoch, epochs
W	Number of provisioned serverless functions (workers)
M	Memory allocation of serverless functions
K	Number of aggregators
B^a, B^n	Batch size of aggregators, non-aggregators
tp_{up}, tp_{down}	upload, download throughput of serverless functions

A. End-to-End Training Time

a) Communication Overhead: As shown in Fig. 2, the communication includes three steps, uploading, aggregation and downloading. Firstly, each worker splits the model parameters into K shards and uploads them to the communication channel in the uploading step. The shard size S_s equals $\frac{S_m}{K}$, where S_m denotes the size of the deep model. In the aggregation step, each aggregator downloads the corresponding shards from the other $W-1$ workers and uploads the merged shard back to the communication channel. Finally, each worker downloads all the merged shards from the communication channel and recombines them back to model parameters. Based on the above analysis, the time consumption of these three steps is shown in Equation 1a, Equation 1b and Equation 1c, respectively.

$$T_{up} = \frac{S_m}{tp_{up}} = K \cdot \frac{S_s}{tp_{up}} \tag{1a}$$

$$\begin{aligned} T_{agg} &= \frac{(W-1)}{K} \cdot \frac{S_m}{tp_{down}} + \frac{1}{K} \cdot \frac{S_m}{tp_{up}} \\ &= (W-1) \cdot \frac{S_s}{tp_{down}} + \frac{S_s}{tp_{up}} \end{aligned} \tag{1b}$$

$$T_{down} = \frac{S_m}{tp_{down}} = K \cdot \frac{S_s}{tp_{down}} \tag{1c}$$

In Equation 1, tp_{up} and tp_{down} represent the upload and download throughput of Lambda functions, respectively. As observed in Fig. 4, the throughput is related to the memory quota M and the size of the transferred shard S_s. The maximum bandwidth is positively proportional to the memory quota and when the shard transferred is small, the throughput declines due to the under-utilization. Therefore, we use an exponential saturation function to model the relationship between throughput tp and the size of the shard S_s under a fixed memory quota M, as shown in equation Equation 2.

$$tp = p(M) * \left(1 - e^{-t(M)*S_s}\right) \tag{2}$$

$p(M)$ represents the maximum bandwidth achievable under the specific memory quota M, and $t(M)$ characterizes the decay rate of throughput as the size of the shard diminishes. $p(M)$ and $t(M)$ are platform-specific coefficients and vary for upload and download.

Based on the above analysis, we obtain the mathematical model of the communication overhead in Equation 3 where $C_1 = S_m(\frac{W-1}{tp_{down}} + \frac{1}{tp_{up}})$ and $C_2 = S_m(\frac{1}{tp_{up}} + \frac{1}{tp_{down}})$. When K is small, the shard size S_s is large and the throughput tp_{up} and tp_{down} is approximately equal to the maximum bandwidth. Therefore, C_1 and C_2 are around a constant and the total time of communication T_{comm} is an inverse proportional function of K. However, as the K gets larger and approaches W, the model parameters are split into smaller shards, leading to the degradation of the throughput and the increase of communication overhead, which is consistent with the observation of Fig. 3.

$$\begin{aligned} T_{comm} &= T_{up} + T_{agg} + T_{down} \\ &= \frac{S_m}{tp_{up}} + \left(\frac{W-1}{tp_{down}} + \frac{1}{tp_{up}}\right)\frac{S_m}{K} + \frac{S_m}{tp_{down}} \\ &= \frac{C_1}{K} + C_2 \end{aligned} \tag{3}$$

b) One-Iteration Training Time: The execution time of computational workloads on CPUs is influenced by the volume of computation required and the available computational resources. In the context of a serverless platform, the duration of a single training iteration can be primarily attributed to the batch size B, which determines the computation volume, and the memory quota M, which is directly proportional to the CPU capacity provided by the Lambda service. We evaluate the one-iteration training time of ResNet50 and SqueezeNet with different batch sizes and memory quotas. Based on the observation from Fig. 8, the training time of an iteration T_{train} is linear to the batch size B and is the reciprocal of a linear function of memory quota M. Therefore, we formulate the training time of an iteration in Equation 4 where a, b and m are training-specific coefficients.

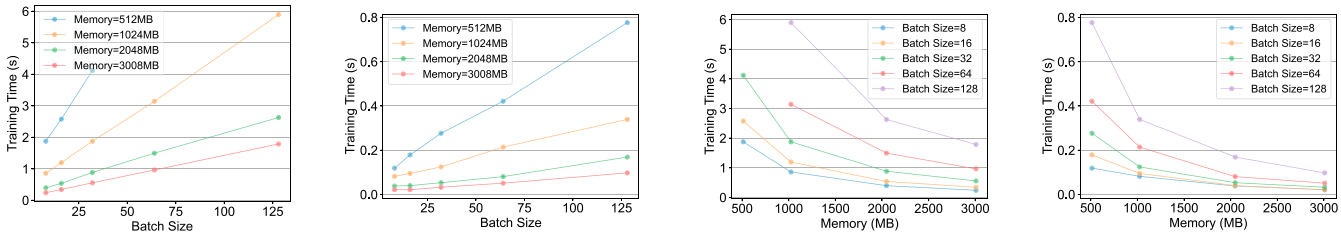
$$T_{train\_iter} = a \cdot \frac{B+b}{M+m} \tag{4}$$

c) Initialization Overhead: Prior to the initiation of training, workers loads the training model and partitioned training data from S3. Due to the 15-minute maximum lifetime constraint of Lambda functions, workers must halt at a consistent iteration before the function times out, and retrigger the function. Moreover, given the stateless nature of Lambda functions, each new worker necessitates reloading the model and dataset. We choose the completion of each epoch as the synchronization point to initiate a new round of functions. The duration of dataset loading is contingent upon the non-aggregators, which are apportioned a larger share of data owing to the unequal dataset partitioning scheme. As analyzed in Section IV-B, the size of the dataset allocated to different roles is proportional to their respective batch sizes. Consequently, the overhead associated with the loading of the model and training data is represented in Equation 5.

$$T_{load} = \frac{S_m + S_d \cdot \frac{B^n}{B^g}}{tp_{down}} \tag{5}$$

S_m and S_d represent the size of training model and dataset respectively. The global batch size B^g is defined as the sum of the local batch size of all the workers, *i.e.,* the number of training samples in an iteration. We formulate the global batch size as Equation 6 in K-REDUCE.

$$B^g = B^a \cdot K + B^n \cdot (W - K) \tag{6}$$



(a) ResNet50. (b) SqueezeNet. (c) ResNet50. (d) SqueezeNet.

Fig. 8. Training time with different batch size and memory.

d) End-to-End Training Time: In our design, the training of each epoch is accomplished by triggering one round of Lambda functions. The overhead of each epoch includes a one-time initialization and the training and communication incurred in each iteration. The number of iterations I in each epoch is determined by the total number of training samples D and the global batch size B^g, as $I = \frac{D}{B^g}$.

Therefore, the end-to-end training time can be calculated in Equation 7.

$$T = (T_{load} + (T_{train\_iter} + T_{comm}) \cdot I) \cdot E \tag{7}$$

B. Monetary Cost

The total cost C consists of the billing of Lambda Function C_λ and network transmission. The billing of Lambda functions is related to the memory quota M and running time T. Given the unit price of Lambda pr^λ, the cost of AWS Lambda can be calculated as

$$C_\lambda = pr^\lambda \cdot T \cdot M \cdot W \tag{8}$$

The selection of storage services includes persistent storage (e.g. S3) and caching (e.g. ElasticCache). The billing of S3 is based on the number of requests PUT and GET. The upload and download requests number of one-time communication can be calculated as $R_{up} = K \cdot W$ and $R_{down} = 2K \cdot (W - 1)$.

$$C_{S3} = I \cdot (pr^{S3}_{PUT} \cdot R_{up} + pr^{S3}_{GET} \cdot R_{down}) \tag{9}$$

The billing of ElasticCache is based on the size of the data transferred. According to Fig. 2, the total size of the model parameter in a each iteration is equal to $S_m \cdot (3W - 2)$. Given the unit price of ElasticCache pr^{EC}, the cost of ElasticCache can be calculated as

$$C_{EC} = pr^{EC} \cdot E \cdot (2S_d + I \cdot (S_m \cdot (3W - 2))) \tag{10}$$

Therefore, we formulate the total monetary cost in Equation 11.

$$C = C_\lambda + \begin{cases} C_{S3} & \text{with S3} \\ C_{EC} & \text{with ElasticCache} \end{cases} \tag{11}$$

C. Convergence Efficiency

The training loss is influenced by both the number of epochs E and global batch size B^g. As the global batch size increases, the model typically requires a greater number of epochs to achieve a specified training loss value, *i.e.,* converges slowly, as observed in studies such as [12]. Therefore, with a specified number of training epochs, the global batch size impacts the convergence efficiency of the model.

VI. Parameter Configuration

A. Problem Formulation

As indicated in Section V, the performance of `K-REDUCE` is related to the following five configuration-related parameters: the memory quota M of functions, the number of total workers W and aggregators K, the batch size of aggregators B^a and non-aggregators B^n. In addition, the performance can be evaluated in three aspects: end-to-end training time, monetary cost and convergence efficiency. Therefore, we formulate the optimization problem in `K-REDUCE` as follows: given the end-to-end time constraint T_{con} and convergence-enabling maximum global batch size B^g_{max}, find the optimal configuration-related parameters that minimize the monetary cost in total. The optimization problem can be formally defined in Equation 12.

$$\begin{aligned} \min_{M,W,K,B^a,B^n} \quad & C_{\texttt{K-REDUCE}} \\ \text{s.t.} \quad & T \le T_{\text{con}}, \\ & B^g \le B^g_{\max} \end{aligned} \tag{12}$$

By substituting Equations 6, 2, 1, 3, 4 into Equation 7 and Equation 11, the end-to-end training time T and monetary cost C are actually affected by M, W, K, B^a and B^n. Obviously, the total time T in constraint and the total monetary cost C are non-linear with the undetermined parameters above. Accordingly, the optimization problem in Equation 12 turns out to be in the form of non-linear integer programming, which is NP-hard to solve [26].

B. Pruning-Based Heuristic Algorithm

To solve the above NP-hard optimization problem, we design a pruning-based heuristic search algorithm to search for relatively optimal parameter configuration efficiently.

1) Parameter-Range Pruning: Firstly, we narrow down the search range of each parameter sequentially as follows:

a) Batch Size of Aggregators B^a: We define the training rate TR as the number of training samples processed per unit time in an iteration.

$$TR = \frac{B^a}{T_{train}} = \frac{M + m}{a} \cdot \frac{1}{1 + \frac{b}{B^a}} \tag{13}$$

The training rate is capped at $\frac{M+m}{a}$, with $\gamma = \frac{1}{1+\frac{b}{B^a}}$ representing the proportion of this maximum rate achieved. To ensure a high training rate, we set a minimum threshold γ_{min} for the proportion (empirically set between 0.6 and 0.8), thereby determining a lower bound for the batch size. As to the upper bound of batch size, two constraints need to be taken into consideration. First, as the increase of batch size, the minimal executable memory increases as well, which should not exceed the memory quota M. Second, the constraint of maximum global batch size should be satisfied. Therefore, we narrow down the search range of batch size B^a as follows:

$$\begin{aligned} &B_{lower} \le B^a \le B_{upper} \\ &B_{lower} = \frac{b}{\frac{1}{\gamma_{min}} - 1} \\ &B_{upper} = \min\left\{B_M, \frac{B^g_{max}}{W}\right\} \end{aligned} \tag{14}$$

where B_M is the maximum executable batch size under memory quota M. Given that the upper bound is dependent on W and M, we ascertain it subsequent to the determination of W and M. In addition, we search for the optimal batch size with a step of 16 to balance computational efficiency with precision.

b) Number of Workers W: Given the lower bound of batch size B_{lower}, the number of workers W is constrained by the maximum global batch size B^g_{max}. Therefore, we narrow the search range of W with the following upper bound:

$$W \le W_{upper} = \frac{B^g_{max}}{B_{lower}} \tag{15}$$

c) Memory Quota M: As to the memory quota of functions, we mainly consider the following two constraints.

First, for a given model and batch size, the memory quota must be greater than the minimum executable memory. We determine it through a profiler, and we provide a detailed description in Section VII. Second, the serverless platform imposes limits on the memory quota for individual functions, which is 10240MB in AWS Lambda. Although the memory allocation granularity is 1MB on AWS Lambda, we set the search step of memory as 128MB, which speeds up the searching significantly while only introducing a minor deviation in the results.

d) Number of Aggregators K: We search the number of aggregators K, starting from W that represents ScatterReduce, down to 1, indicative of AllReduce. The advantages of the `K-REDUCE` stem from two factors: the reduction of communication overhead and the effective utilization of non-aggregators' idle CPU. The optimal value for K is determined at the point where the combination of these two benefits is maximized.

e) Batch Size of Non-Aggregators B^n: According to the analysis in Section IV-B, the batch size of non-aggregators B^n should meet the alignment constraint as follows:

$$T_{train}(B = B^n) = T_{train}(B = B^a) + T_{agg} \tag{16}$$

In addition, same as the batch size of aggregators B^a, the choice of B^n should meet the constraints of the executability under the given memory quota M and global batch size with the given number of workers W.

2) Two-Stage Search: Lemma VI.1 proves that with the same configuration of parameters M, W, B^a, the minimum cost of `K-REDUCE` is no more than that of ScatterReduce.

Lemma 1: $\min_{K^*} C_{\texttt{K-REDUCE}} \le C_{Scatter-Reduce}$

Proof: As `K-REDUCE` further reduces the end-to-end training time T with the optimal K^*, the billing of Lambda service in `K-REDUCE` is no more than that of ScatterReduce with the same M and W. Besides, the number of requests of `K-REDUCE` in one-time communication is less than that of ScatterReduce as $K^* \le W$, so the billing of S3 service in `K-REDUCE` is also no more than that of ScatterReduce. In summary, with the same configuration of parameters M, W, B^a, the minimum cost of `K-REDUCE` is no more than that of ScatterReduce. □

Based on the insight above, we divide the five parameters into two sets: (1) M, W and B^a which are common dual-level parameters in ScatterReduce; (2) K and B^n which are introduced in `K-REDUCE`. Then, we divide the search of the original problem into two sequential stages, each targeting a separate set of parameters, thereby narrowing down the search space efficiently. This two-stage search approach acts as a pruning strategy, as it reduces the complexity by eliminating the need to consider all possible combinations of parameters simultaneously.

a) First Stage: In the first stage, we reformulate the optimization problem to identify the common dual-level parameters W, M, and B^a based on ScatterReduce as follows:

$$\begin{aligned} \min_{M,W,B^a} \quad & C_{Scatter-Reduce} \\ \text{s.t.} \quad & T \le \frac{T_{con}}{\delta} \\ & B^g \le B^g_{max} \cdot \delta \end{aligned} \tag{17}$$

Compared with ScatterReduce, the introduction of `K-REDUCE` in the second stage not only reduces the end-to-end training time but also increases the global batch size as we transform several aggregators into non-aggregators and set a larger batch size. Considering that, we relax the constraint on end-to-end training time and tighten the constraint on global batch size with a reduction factor δ ($0 < \delta \le 1$). We perform nested iterations over the parameters W, M, and B^a in a sequential manner. For a given W, we iterate over the M in decreasing order. Should it occur that no parameter configurations meet the time constraints under the current M, an early stopping mechanism is employed. This mechanism halts further iterations over smaller M values for the current W, thus avoiding unnecessary computations and enhancing the efficiency of the search process.

b) Second Stage: In the second stage, with the specified B^a, W and M, we determine the parameters K and B^n introduced by `K-REDUCE`. We iterate K from W to 1 and for the current K, we can determine the corresponding B^n by Equation 16. We repeat the two-stage search with different reduction factors δ (*i.e.,* 0.6, 0.7, 0.8, 0.9 and 1) and return the parameters configuration with the minimum cost. As depicted in Fig. 3, smaller models such as SqueezeNet are inclined to choose lower values of K to avoid the issue of throughput

degradation. Consequently, there exists a more substantial optimization opportunity for small models in the second stage, which permits the choice of smaller δ. In contrast, larger models generally exhibit a preference for higher values of δ.

c) Complexity Analysis: The complexity of the first stage is in the order of $O(p \cdot q \cdot l)$, where $p = W_{\text{upper}} - W_{\text{lower}} + 1$ denotes the cardinality of the search range of W, q denotes the number of possible function memory sizes, which is equal to $\frac{M_{\text{max}} - M_{\text{min}}}{M_{\text{step}}} + 1$ and l denotes the number of possible batch sizes, which is equal to $\frac{B_{\text{upper}} - B_{\text{lower}}}{B_{\text{step}}} + 1$. The complexity of the second stage is in the order of $O(W)$, where W is the number of workers identified in the first stage. Overall, by dividing the search for five parameters into two sequential stages, we reduce the complexity of the algorithm from $O(p \cdot q \cdot l \cdot W)$ to $O(p \cdot q \cdot l + W)$. Additionally, we adopt a parameter-range pruning strategy and early stopping mechanisms in the first stage, which further improves the efficiency of the algorithm. In particular, the pruning-based heuristic algorithm identifies a sub-optimal parameter configuration plan, as we narrow down the search space of parameters. We validate the efficiency and effectiveness of the pruning-based heuristic algorithm in Section VIII-C.

VII. Implementation

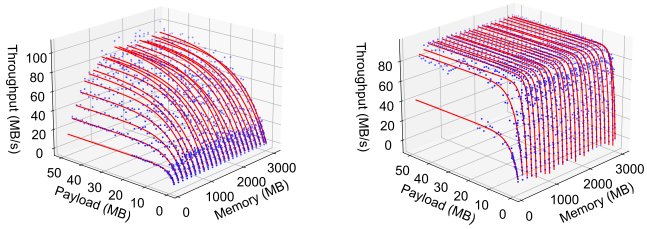
We build a prototype for **FasDL** in Python(∼1.5k loc) to realize the `K-REDUCE` atop AWS Lmabda, Amazon S3 and Amazon ElasticCache.

We design a profiler to determine the workload-specific and platform-specific coefficients. The workload-specific coefficients include model size and training-related coefficients a, b and m. Since the training-related coefficient is only related to the characteristics of the model itself, the profiler conducts the following steps by invoking a Lambda function to determine them. The profiler evaluates the one mini-batch training time of the given model under varying batch sizes and memory quotas, subsequently using the method of least squares to fit the coefficients based on the Equation 4. The platform-specific coefficients are used to characterize the network capabilities provided by the platform. According to Equation 2, with a fixed memory quota, the throughput demonstrates an exponential saturation relationship with the size of the payload. The platform-specific coefficients include a set of throughput-related coefficients $p(M)$ and $t(M)$, which are dependent on the memory quota M. The profiler measures the upload and download throughput between AWS Lambda and storage services by transferring payloads of varying sizes under varying memory quotas. The test interval of the memory quota is 128MB, which is consistent with the search step in the search algorithm. Subsequently, it employs the least squares method to fit the throughput-related coefficients under different memory quotas.

We follow the hierarchical invocation mechanism in LambdaML [14] to handle the limited lifetime of the Lambda function. Although we implement **FasDL** on top of AWS products, it is platform-independent and can serve as a plugin for other commercial or open-source serverless computing platforms.

TABLE II
Workloads

Model	Dataset	Model Size(MB)	Training-related Coefficients		
			a	b	m
BERT-Base	CoLa	417.17	2344.97	2.37	1145.66
ResNet50	CIFAR10	97.49	37.19	12.48	-111.46
MobileNet	CIFAR10	13.37	13.50	9.57	-92.88
SqueezeNet	CIFAR10	4.71	2.16	22.60	-93.22



(a) Upload Throughput. (b) Download Throughput.

Fig. 9. Profiling and fitting of throughput between AWS Lambda and Amazon S3.

VIII. Performance Evaluation

A. Experimental Setup

We implement a prototype of **FasDL** on AWS Lambda, adapting communication channels including both Amazon S3 [27] and Amazon ElasticCache Serverless [28]. We select SqueezeNet, MobileNet, ResNet50, and BERT-Base as our model workloads, using CIFAR-10 as the dataset for the first three models, and the CoLA dataset for the BERT-Base model. We employ the serverless adaptations of the AllReduce and ScatterReduce introduced by LambdaML [14] and λDNN [12] as the benchmark.

Before the evaluation experiments, we adopt the profiler to obtain the training-related coefficients of four workloads as shown in Table II and the communication throughput between the AWS Lambda function and storage services, such as Amazon S3, as illustrated in Fig. 9. Then, we evaluate the performance of **FasDL** as follows: Firstly, we assess the accuracy of the modeling (Section V) for both time and monetary cost. Secondly, we validate the performance of the pruning-based heuristic algorithm (Section VI), focusing on both efficiency and effectiveness of parameter configuration. Thridly, we compare the performance of `K-REDUCE` (Section IV) with AllReduce and ScatterReduce in LambdaML in terms of end-to-end training time, monetary cost and convergence efficiency. Finally, we compare the performance of different communication channel implementations and performance manifestations between FasDL and standalone training.

B. Accuracy of Modeling

In this section, we verify the accuracy of the modeling (Section V) adopted by the profiler and predictor. Firstly, we

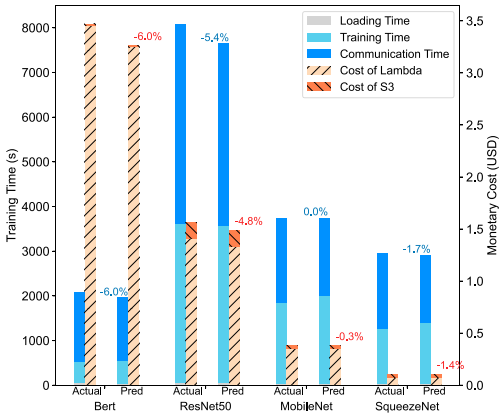


Fig. 10. Precision analysis of predictions for K-REDUCE.

TABLE III
COMPARISON OF DIFFERENT SEARCH STRATEGIES

Model	Search Strategy	Pred Cost (USD)	Performance Degradation	Search Time(s)	Search Speedup
BERT-Base	Two-Stage	3.42	11.6%	32.1	4.8x
	Brute-force	3.02		155.0	
ResNet50	Two-Stage	1.49	14.6%	290.5	9.1x
	Brute-force	1.30		2645.6	
MobileNet	Two-Stage	0.38	2.7%	83.1	11.4x
	Brute-force	0.37		954.6	
SqueezeNet	Two-Stage	0.104	6.1%	217.0	2.0x
	Brute-force	0.098		443.9	

obtain the training-related coefficients of four workloads with the profiler as shown in Table II and determine the parameter configuration of K-REDUCE with the optimizer as shown in Table IV. Then, with the parameter configuration, we carry out the training following the K-REDUCE framework and record the actual time and monetary cost. As shown in Fig. 10, we compare the actual values with the predicted values yielded by the predictor in terms of time and monetary cost. The prediction errors for time and monetary cost are within 6%. The prediction error primarily arises from inaccuracies in communication time, which are caused by the fluctuation of function throughput. Since the coefficients used by the predictor are fitted through the least squares method from a limited and noisy dataset, it consequently leads to the prediction error.

C. Efficiency and Effectiveness of Parameter Configuration

In this section, we measure the performance of the optimizer that employs the pruning-based heuristic algorithm (Section VI). Specifically, the optimizer narrows down the search range of parameters with the parameter-range pruning and then adopts the two-stage search to expedite the search process. The parameter-range pruning reduces the overhead of the NP-hard problem to an acceptable level and eliminates infeasible parameter configurations due to limitations such as insufficient memory for execution. For the two-stage search, we set up a baseline that conducts a brute-force search after parameter-range pruning. As Table III demonstrates, the two-stage search increases the search speed for at most $11.4\times$ compared with the brute-force search. This high efficiency is achieved by dividing the parameter configuration into two independent stages, which significantly reduces the complexity of the algorithm. To validate the effectiveness of the two-stage search, we define the performance degradation as $(\frac{Cost_{\text{Two-Stage}}}{Cost_{\text{Brute-force}}} - 1) \times 100\%$, which is only 11.6%, 14.6%, 2.7% and 6.1% for the four workloads respectively. We trade off a slight performance degradation for a significantly improved search efficiency.

D. Comprehensive Analysis of K-REDUCE

To validate the performance of K-REDUCE (Section IV), we conduct a comparative analysis of the K-REDUCE against the ScatterReduce and AllReduce in LambdaML, focusing on end-to-end training time and monetary cost and convergence efficiency. In this section, we adopt S3 as the communication channel.

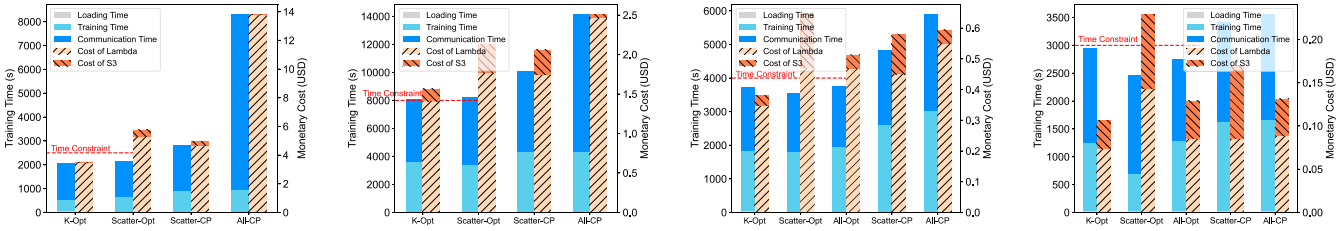
1) Performance on Time and Cost: We verify the effectiveness of the K-Reduce in optimizing training time and cost from two perspectives.

a) Performance Comparison for the Optimization Problem: With respect to the optimization problem presented in Section VI-A, which is to minimize cost under the given training time and global batch size constraints, we compare the performance of K-REDUCE with the ScatterReduce and AllReduce. The constraints for four workloads and their parameter configurations are shown in Table IV. For K-REDUCE, the parameters K-Opt are provided by the two-stage search in Section VI-B, while for ScatterReduce and AllReduce, the parameters Scatter-Opt and All-Opt are determined through brute-force search, respectively. For BERT-Base and ResNet50, no parameter configuration meets the time constraints when using AllReduce for communication. This is because for larger models, choosing only one aggregator for aggregation results in excessively high communication overhead. As illustrated in Fig. 11, given the end-to-end training time constraints, K-REDUCE can significantly reduce monetary cost compared to ScatterReduce and AllReduce. More specifically, compared with ScatterReduce, K-REDUCE shows a reduction of 39.0%, 26.7%, 25.8% and 53.2% on monetary cost when training with BERT-Base, ResNet50, MobileNet and SqueezeNet respectively. Compared with AllReduce, K-REDUCE shows a reduction of 25.8% and 17.6% in monetary cost when training with MobileNet and SqueezeNet. As K-REDUCE reduces communication time by selecting the optimal number of aggregators K and accelerates training through the unequal dataset partitioning scheme alongside the Hybrid Asynchronous Parallel protocol, it overall enhances the training performance. Consequently, as shown in Table IV, under the same time constraints, the parameter configuration of K-REDUCE requires fewer resources (i.e. number of workers W and memory quota M) than ScatterReduce and AllReduce, thus reducing the monetary costs of deployment and training.

b) Time and Cost Optimization Within Fixed Resources: We then compare K-REDUCE to ScatterReduce and AllReduce under the same resource configuration to verify the efficacy of K-REDUCE in terms of time and cost. We denote Scatter-CP and All-CP as ScatterReduce and AllReduce, respectively,

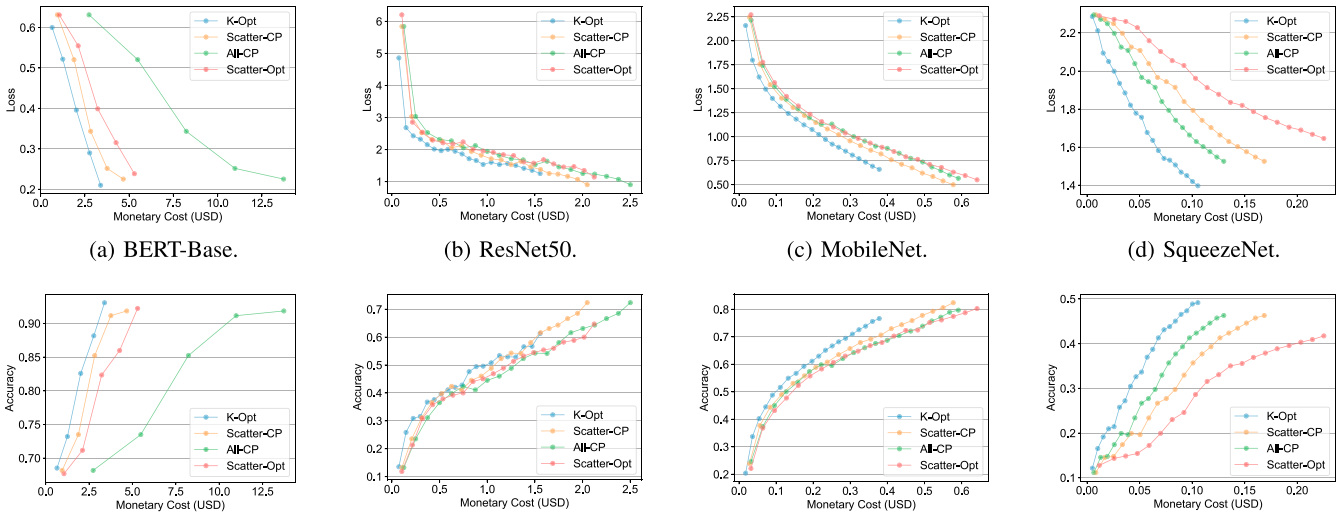
TABLE IV
PARAMETER CONFIGURATION OF DIFFERENT COMMUNICATION PATTERNS

Model	Time Constraint(s)	Global Batch Size	K-REDUCE					Scatter Reduce			All Reduce			λDNN		
			N	M	B^a	K	B^n	N	M	B	N	M	B	N	M	B
BERT-Base	2500	640	10	10240	32	4	84	15	10240	32	No Fit			10	10240	32
ResNet50	8000	1024	7	1536	128	4	170	8	1664	128				2	3328	512
MobileNet	4000	512	3	1920	128	1	192	4	2048	128	4	1920	128	2	2304	256
SqueezeNet	3000	384	2	768	128	1	201	2	1792	128	3	640	128	2	768	192



(a) BERT-BASE. (b) ResNet50. (c) MobileNet. (d) SqueezeNet.

Fig. 11. Comparison on end-to-end training time and monetary cost.



(a) BERT-Base. (b) ResNet50. (c) MobileNet. (d) SqueezeNet.

(e) BERT-Base. (f) ResNet50. (g) MobileNet. (h) SqueezeNet.

Fig. 12. Comparison on convergence efficiency.

each employing the common dual-level parameters (*i.e.*, M, W and B) shared with K-REDUCE (K-Opt). As shown in Fig. 11, K-Opt demonstrates superior performance improvements across all four models when compared to both Scatter-CP and All-CP. On average, K-REDUCE outperforms ScatterReduce by approximately 16.8% in terms of time and 28.3% in terms of cost and it outperforms AllReduce by roughly 33.6% in time and 31.4% in cost. With the fixed resource configuration, K-REDUCE reduces the end-to-end training time in two respects. On the one hand, K-REDUCE selects the optimal number of aggregators K from the given workers, which minimizes the communication time. On the other hand, K-REDUCE utilizes the idle CPU resources of non-aggregators during the aggregation step, thereby accelerating the model training. With the reduction of the end-to-end training time, the cost of Lambda decreases accordingly, as the billing of Lambda is proportional to running time. Moreover, compared to ScatterReduce, which involves all workers in aggregation, K-REDUCE decreases the number of communication requests by selecting fewer aggregators. Given that Amazon S3's billing for network usage is based on the number of requests, this approach by K-REDUCE reduces the cost of network.

2) Performance on Convergence Efficiency: Contrary to AllReduce and ScatterReduce employing the Bulk Synchronous Parallel protocol, K-REDUCE implements the Hybrid Asynchronous Parallel protocol. We compare the training loss and accuracy under the same cost conditions. As shown in Fig. 12, with the constraints on the global batch size and the fixed staleness of 1 in HAP (Section IV-B), the convergence efficiency of the K-REDUCE is effectively guaranteed. In most

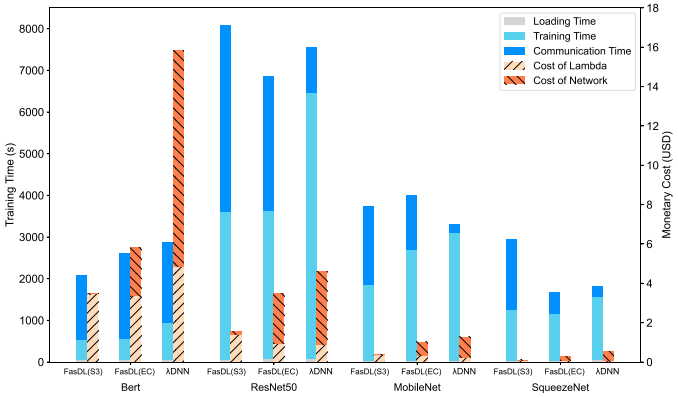


Fig. 13. Comparison of different communication channel implementations for minimized cost under time constraints.

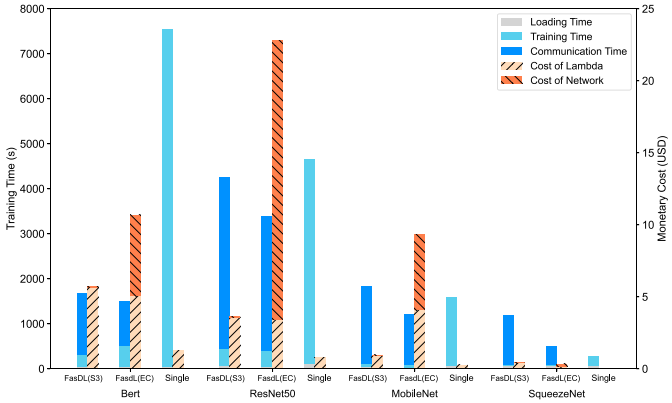


Fig. 14. Comparison between FasDL and standalone implementations for minimized end-to-end training time.

cases, K-REDUCE shows a convergence performance that is not only comparable to but often surpasses that of the other mechanisms.

E. Comparison of Different Communication Channel Implementations

To solve the lack of inter-function communication channels during distributed training on serverless platforms, there are two common implementation strategies: storage-based and server-based communication channels. FasDL adopts the former, utilizing cloud storage services, whereas λDNN adopts a parameter server for parameter aggregation. With our profiling and modeling, FasDL can accommodate different types of storage services, including persistent storage services (e.g. S3) and caching services (e.g. ElasticCache). In this section, we first compare the performance of FasDL for persistent storage services and caching services, using Amazon S3 and Amazon ElasticCache Serverless respectively. Then, we compare FasDL with λDNN to analyze the performance of different communication channel implementations.

1) Persistent Storage Service vs. Caching Service: Fig. 13 shows the minimized cost of FasDL with S3 and ElasticCache under given training time and global batch size constraints. Since the cost of ElasticCache is related to the size of the data transferred, its overhead is much higher than S3 as the model size increases. At the same time, due to the faster read/write speed of ElasticCache, the communication time is less than that of S3 in most cases. However, when training larger models (e.g. BERT), ElasticCache is inferior to S3 in terms of both time and cost. As shown in Equation 10, with substantial model sizes, the cost of ElasticCache increases significantly with the number of workers. Therefore, the optimal parameter configuration tends to involve fewer workers for training. Additionally, the maximum configurable memory for Lambda functions limits the batch size of workers, resulting in the global batch size not fully exploiting the constraint, thus requiring more rounds of communication.

2) Storage-Based vs. Server-Based Communication Channels: In this section, we compare FasDL with λDNN, which adopts a server as communication channel. We follow the implementation of λDNN, using an m5.large EC2 instance (equipped with 2 vCPUs and 8 GB memory) to serve as the Parameter Server (PS) and conduct experiments under the same optimization problem. The results are shown in Fig. 13. Similar to ElasticCache, the network billing of EC2 instance is related to the amount of data transferred, leading to similar characteristics in terms of communication time and cost. Overall, both S3 and ElasticCache implementations show superior cost efficiency compared to λDNN, with a maximum cost reduction of 78.13%. Additionally, compared to server-based implementations, cloud storage and caching services are pay-as-you-go and serverless, further reducing costs and deployment overhead for developers.

F. Comparison Between FasDL and Standalone Training

In this section, we compare the minimized end-to-end training time of FasDL with that of standalone training in the serverless context, *i.e.,*, conducting local training using a single Lambda function. Current serverless platforms impose maximum resource limits on individual functions, e.g., 10240MB in AWS Lambda. We conduct standalone training using the maximum resource allocation ,and for FasDL, we search for the parameter configurations that minimize training time under both S3 and ElasticCache implementations. As shown in Fig. 14, since local standalone training does not incur communication overhead, it outperforms FasDL in terms of both training time and cost when the model is relatively simple (e.g., SqueezeNet). However, as models are getting more complex, the maximum resource allocation for a single function becomes a bottleneck, leading to training times that can be up to 5 times longer than those of FasDL. In summary, standalone training in a serverless context can significantly reduce costs due to the pay-as-you-go billing strategy. Nevertheless, the resource limit of a single function remains a bottleneck for further accelerating training time, especially for complex models like BERT.

IX. Related Work

a) Distributed Training of DNNs: There are three common parallel mechanisms in distributed training to handle the

increase of the training data volume and the complexity of deep learning models: data parallelism, model parallelism and pipeline parallelism. Parameter server [29], [30] is the pioneering prototype of data parallelism with the training data distributed among workers, which has been further developed in subsequent works [31], [32]. It seeks to expedite the training process, whereas model parallelism [33], [34], [35] is beneficial where the model's size exceeds the memory capacity of a single machine. Recently introduced pipeline parallelism [36], [37], [38], [39] combines the benefits of data and model parallelism, achieving reduced training time for large-scale models. **FasDL** optimizes the training procedure based on the data parallelism since there is no peer-to-peer communication [1] between serverless functions while the communication overhead of the other mechanisms is more complex and demanding.

b) Serverless-Based Communication Optimization: Because of the stateless nature of serverless functions, the lack of peer-to-peer inter-function communication results in the overhead of serverless runtime. FaaSFlow [40] implements an adaptive storage library to enable communication between functions to bypass the external channel. Yu et al. propose Pheromone [41] equipped with a data-centric function orchestration approach, supporting direct and efficient data exchange between functions. These optimization techniques are orthogonal to ours, which can boost **FasDL**'s performance with elaborate design and consolidation.

c) Serverless-Based Training Optimization: Serverless computing has arisen attraction from deep learning practitioners [12], [13], [14], [42], [43], [44]. Among them, several prototypes have been proposed as the baseline of further research. Jiang et al. propose LambdaML [14] to conduct extensive characterization of serverless machine learning, including communication patterns and synchronization protocols. Many other improvement schemes [12], [13] have been proposed to optimize the performance of training deep learning models on serverless computing frameworks from various aspects. Compared with existing works, **FasDL** is designed to be platform-independent, which means that it can be deployed on other cloud platforms with similar serverless computing capabilities, and even on platforms with GPU support [45], [46].

d) Modeling and Configuration Optimization in Serverless Computing: The emergence of the serverless computing paradigm has brought about substantial shifts in application deployment methodologies. Consequently, numerous research has been directed towards achieving predictable performance and optimal resource configuration within this paradigm. Similar to our study, λDNN [12] and Funcpipe [13] target the enhancement of performance and predictability for distributed training tasks. Besides the training tasks, a substantial amount of work [47], [48], [49] is dedicated to optimizing the performance and resource efficiency of distributed inference tasks within the serverless framework. Moreover, some studies [50], [51], [52] have ventured into modeling the performance and cost associated with general serverless workflows.

X. Conclusion

We introduce **FasDL**, a novel architecture for optimizing distributed training workloads of deep learning models. **FasDL** employs a novel `K-REDUCE` framework to increase the computation-to-communication ratio and builds a comprehensive analytical module achieving predictable training performance and automatic resource configuration. Extensive prototype experiments on AWS Lambda demonstrate that **FasDL** achieves predictable training performance within 6% error, speeds up the training by up to 16.8% and reduces the cost by up to 28.3%, in comparison to the state-of-the-art architecture LambdaML with ScatterReduce.

References

[1] E. Jonas et al., "Cloud programming simplified: A Berkeley view on serverless computing," 2019, *arXiv:1902.03383*.

[2] Z. Li, L. Guo, J. Cheng, Q. Chen, B. He, and M. Guo, "The serverless computing survey: A technical primer for design architecture," *ACM Comput. Surv.*, vol. 54, no. 10s, pp. 1–34, 2022.

[3] H. Shafiei, A. Khonsari, and P. Mousavi, "Serverless computing: A survey of opportunities, challenges, and applications," *ACM Comput. Surv.*, vol. 54, no. 11s, pp. 1–32, 2022.

[4] "AWS Lambda," [Online]. Available: https://aws.amazon.com/lambda/

[5] R. Moreno-Vozmediano, R. S. Montero, and I. M. Llorente, "IaaS cloud architecture: From virtualized datacenters to federated cloud infrastructures," *Computer*, vol. 45, no. 12, pp. 65–72, Dec. 2012.

[6] E. Van Eyk, L. Toader, S. Talluri, L. Versluis, A. Uță, and A. Iosup, "Serverless is more: From PaaS to present cloud computing," *IEEE Internet Comput.*, vol. 22, no. 5, pp. 8–17, Sep. 2018.

[7] W. Tsai, X. Bai, and Y. Huang, "Software-as-a-service (SaaS): Perspectives and challenges," *Sci. China Inf. Sci.*, vol. 57, pp. 1–15, May 2014.

[8] M. Zhang, F. Wang, Y. Zhu, J. Liu, and B. Li, "Serverless empowered video analytics for ubiquitous networked cameras," *IEEE Netw.*, vol. 35, no. 6, pp. 186–193, 2021.

[9] R. B. Roy, T. Patel, and D. Tiwari, "DayDream: Executing dynamic scientific workflows on serverless platforms with hot starts," in *Proc. Int. Conf. High Perform. Comput., Netw., Storage Anal.*, 2022, pp. 1–18.

[10] H. Wang, D. Niu, and B. Li, "Distributed machine learning with a serverless architecture," in *Proc. IEEE Conf. Comput. Commun. (INFOCOM)*, Piscataway, NJ, USA: IEEE Press, 2019, pp. 1288–1296.

[11] M. Sánchez-Artigas and P. G. Sarroca, "Experience paper: Towards enhancing cost efficiency in serverless machine learning training," in *Proc. Int. Middleware Conf.*, 2021, pp. 210–222.

[12] F. Xu, Y. Qin, L. Chen, Z. Zhou, and F. Liu, "λDNN: Achieving predictable distributed DNN training with serverless architectures," *IEEE Trans. Comput.*, vol. 71, no. 2, pp. 450–463, Feb. 2022.

[13] Y. Liu et al., "FuncPipe: A pipelined serverless framework for fast and cost-efficient training of deep learning models," *Proc. ACM Meas. Anal. Comput. Syst.*, vol. 6, no. 3, pp. 1–30, 2022.

[14] J. Jiang et al., "Towards demystifying serverless machine learning training," in *Proc. Int. Conf. Manage. Data (SIGMOD)*, 2021, pp. 857–871.

[15] X. Lian, C. Zhang, H. Zhang, C.-J. Hsieh, W. Zhang, and J. Liu, "Can decentralized algorithms outperform centralized algorithms? A case study for decentralized parallel stochastic gradient descent," in *Proc. Adv. Neural Inf. Process. Syst.*, vol. 30, 2017, pp. 5331–5341.

[16] A. Mampage, S. Karunasekera, and R. Buyya, "Deep reinforcement learning for application scheduling in resource-constrained, multi-tenant serverless computing environments," *Future Gener. Comput. Syst.*, vol. 143, pp. 277–292, Jun. 2023.

[17] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit. (CVPR)*, 2016, pp. 770–778.

[18] F. N. Iandola, S. Han, M. W. Moskewicz, K. Ashraf, W. J. Dally, and K. Keutzer, "SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and 0.5 MB model size," 2016, *arXiv:1602.07360*.

[19] M. Bilal, M. Canini, R. Fonseca, and R. Rodrigues, "With great freedom comes great opportunity: Rethinking resource allocation for serverless functions," in *Proc. Eur. Conf. Comput. Syst. (EuroSys)*, 2023, pp. 381–397.
[20] H. Yu, H. Wang, J. Li, X. Yuan, and S.-J. Park, "Accelerating serverless computing by harvesting idle resources," in *Proc. ACM Web Conf.*, 2022, pp. 1741–1751.
[21] A. Mahgoub, E. B. Yi, K. Shankar, S. Elnikety, S. Chaterji, and S. Bagchi, "Orion and the three rights: Sizing, bundling, and prewarming for serverless dags," in *Proc. USENIX Symp. Operating Syst. Des. Implementation (OSDI)*, 2022, pp. 303–320.
[22] Amazon Web Services, "AWS Lambda—FAQ," Accessed: Mar. 13, 2024. [Online]. Available: https://aws.amazon.com/lambda/faqs/
[23] A. V. Gerbessiotis and L. G. Valiant, "Direct bulk-synchronous parallel algorithms," *J. Parallel Distrib. Comput.*, vol. 22, no. 2, pp. 251–267, 1994.
[24] J. K. Bradley, A. Kyrola, D. Bickson, and C. Guestrin, "Parallel coordinate descent for L1-regularized loss minimization," 2011, *arXiv:1105.5379*.
[25] Q. Ho et al., "More effective distributed ML via a stale synchronous parallel parameter server," in *Proc. Adv. Neural Inf. Process. Syst.*, vol. 26, 2013.
[26] S. Boyd, S. P. Boyd, and L. Vandenberghe, *Convex Optimization*. Cambridge, U.K.: Cambridge Univ. Press, 2004.
[27] "Amazon Simple Storage Service (S3)," Amazon. [Online]. Available: https://aws.amazon.com/s3/
[28] "Amazon ElastiCache Serverless," Amazon. [Online]. Available: https://aws.amazon.com/elasticache
[29] M. Li et al., "Scaling distributed machine learning with the parameter server," in *Proc. USENIX Symp. Operating Syst. Des. Implementation (OSDI)*, 2014, pp. 583–598.
[30] Y. Huang et al., "FlexPS: Flexible parallelism control in parameter server architecture," *Proc. VLDB Endowment*, vol. 11, no. 5, pp. 566–579, 2018.
[31] H. Zheng, F. Xu, L. Chen, Z. Zhou, and F. Liu, "Cynthia: Cost-efficient cloud resource provisioning for predictable distributed deep neural network training," in *Proc. 48th Int. Conf. Parallel Process.*, 2019, pp. 1–11.
[32] R. Shang, F. Xu, Z. Bai, L. Chen, Z. Zhou, and F. Liu, "SpotDNN: Provisioning spot instances for predictable distributed DNN training in the cloud," in *Proc. IEEE/ACM 31st Int. Symp. Qual. Service (IWQoS)*, Piscataway, NJ, USA: IEEE Press, 2023, pp. 1–10.
[33] J. K. Kim et al., "STRADS: A distributed framework for scheduled model parallel machine learning," in *Proc. Eur. Conf. Comput. Syst. (EuroSys)*, 2016, pp. 1–16.
[34] M. Shoeybi, M. Patwary, R. Puri, P. LeGresley, J. Casper, and B. Catanzaro, "Megatron-LM: Training multi-billion parameter language models using model parallelism," 2019, *arXiv:1909.08053*.
[35] A. Xu, Z. Huo, and H. Huang, "On the acceleration of deep learning model parallelism with staleness," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit. (CVPR)*, 2020, pp. 2088–2097.
[36] Y. Huang et al., "GPipe: Efficient training of giant neural networks using pipeline parallelism," in *Proc. Adv. Neural Inf. Process. Syst.*, vol. 32, 2019.
[37] J. H. Park et al., "HetPipe: Enabling large DNN training on (whimpy) heterogeneous GPU clusters through integration of pipelined model parallelism and data parallelism," in *Proc. USENIX Annu. Tech. Conf. (USENIX ATC)*, 2020, pp. 307–321.
[38] D. Narayanan et al., "PipeDream: Generalized pipeline parallelism for DNN training," in *Proc. USENIX Symp. Operating Syst. Des. Implementation (OSDI)*, 2019, pp. 1–15.
[39] H. Shi, W. Zheng, Z. Liu, R. Ma, and H. Guan, "Automatic pipeline parallelism: A parallel inference framework for deep learning applications in 6g mobile communication systems," *IEEE J. Sel. Areas in Commun.*, vol. 41, no. 7, pp. 2041–2056, Jul. 2023.
[40] Z. Li et al., "FaaSFlow: Enable efficient workflow execution for function-as-a-service," in *Proc. ACM Int. Conf. Archit. Support Program. Lang. Operating Syst. (ASPLOS)*, 2022, pp. 782–796.
[41] M. Yu, T. Cao, W. Wang, and R. Chen, "Following the data, not the function: Rethinking function orchestration in serverless computing," in *Proc. USENIX Symp. Netw. Syst. Des. Implementation (NSDI)*, 2023, pp. 1489–1504.
[42] T. Khan, W. Tian, G. Zhou, S. Ilager, M. Gong, and R. Buyya, "Machine learning (ML)-centric resource management in cloud computing: A review and future directions," *J. Netw. Comput. Appl.*, vol. 204, 2022, Art. no. 103405.
[43] S. Pan, H. Zhao, Z. Cai, D. Li, R. Ma, and H. Guan, "Sustainable serverless computing with cold-start optimization and automatic workflow resource scheduling," *IEEE Trans. Sustain. Comput.*, vol. 9, no. 3, pp. 329–340, May/Jun. 2024.
[44] B. Jamil, H. Ijaz, M. Shojafar, K. Munir, and R. Buyya, "Resource allocation and task scheduling in fog computing and internet of everything environments: A taxonomy, review, and future directions," *ACM Comput. Surv.*, vol. 54, no. 11s, pp. 1–38, 2022.
[45] H. Zhao et al., "faaShark: An end-to-end network traffic analysis system atop serverless computing platforms," *IEEE Trans. Netw. Sci. Eng.*, vol. 11, no. 3, pp. 2473–2484, May/Jun. 2024.
[46] Z. Cai, Z. Chen, R. Ma, and H. Guan, "SMSS: Stateful model serving in metaverse with serverless computing and GPU sharing," *IEEE J. Sel. Areas Commun.*, vol. 42, no. 3, pp. 799–811, Mar. 2024.
[47] J. Chen, F. Xu, Y. Gu, L. Chen, F. Liu, and Z. Zhou, "HarmonyBatch: Batching multi-SLO DNN inference with heterogeneous serverless functions," 2024, *arXiv:2405.05633*.
[48] H. Hu, F. Liu, Q. Pei, Y. Yuan, Z. Xu, and L. Wang, "Grapher: A resource-efficient serverless system for GNN serving through graph sharing," in *Proc. Web Conf.*, 2024, pp. 2826–2835.
[49] Q. Pei, Y. Yuan, H. Hu, Q. Chen, and F. Liu, "AsyFunc: A high-performance and resource-efficient serverless inference system via asymmetric functions," in *Proc. ACM Symp. Cloud Comput.*, 2023, pp. 324–340.
[50] F. Liu and Y. Niu, "Demystifying the cost of serverless computing: Towards a win-win deal," *IEEE Trans. Parallel Distrib. Syst.*, vol. 35, no. 1, pp. 59–72, Jan. 2024.
[51] Z. Wen, Q. Chen, Y. Niu, Z. Song, Q. Deng, and F. Liu, "Joint optimization of parallelism and resource configuration for serverless function steps," *IEEE Trans. Parallel Distrib. Syst.*, vol. 35, no. 4, pp. 560–576, Apr. 2024.
[52] C. Lin and H. Khazaei, "Modeling and optimization of performance and cost of serverless applications," *IEEE Trans. Parallel Distrib. Syst.*, vol. 32, no. 3, pp. 615–632, Mar. 2020.

Xinglei Chen is currently working toward the master's degree in computer science with Shanghai Jiao Tong University, Shanghai, China. His research interest includes serverless computing.

Zinuo Cai received the bachelor's degree in software engineering from Shanghai Jiao Tong University, Shanghai, China, in 2021, where he is currently working toward the Ph.D. degree in computer science. His research interests include resource schedule and system security in cloud computing.

Hanwen Zhang is currently working toward the bachelor's degree with the School of Cyberspace Security, Shanghai Jiao Tong University, China. His research interests include network attack and defense security.

Ruhui Ma (Member, IEEE) received the Ph.D. degree in computer science from Shanghai Jiao Tong University, Shanghai, China, in 2011. Currently, he is an Associate Professor with the Department of Computer Science and Engineering, Shanghai Jiao Tong University. His research interests include cloud computing systems, artificial intelligence (AI) systems, and machine learning.

Rajkumar Buyya (Fellow, IEEE) is a Redmond Barry Distinguished Professor and the Director of the Cloud Computing and Distributed Systems (CLOUDS) Laboratory, University of Melbourne, Australia. He has authored more than 625 publications and seven text books including *Mastering Cloud Computing* published by McGraw Hill, China Machine Press, and Morgan Kaufmann for Indian, Chinese and international markets respectively. He is one of the highly cited authors in computer science and software engineering worldwide.