MinFlow: High-performance and Cost-efficient Data Passing for I/O-intensive Stateful Serverless Analytics

Tao Li, Yongkun Li, and Wenzhe Zhu, *University of Science and Technology of China;* Yinlong Xu, *Anhui Province Key Laboratory of High Performance Computing, University of Science and Technology of China,* John C. S. Lui, *The Chinese University of Hong Kong*

https://www.usenix.org/conference/fast24/presentation/li

This paper is included in the Proceedings of the 22nd USENIX Conference on File and Storage Technologies.

February 27–29, 2024 • Santa Clara, CA, USA

978-1-939133-38-0

MinFlow: High-performance and Cost-efficient Data Passing for I/O-intensive Stateful Serverless Analytics

Tao Li[1], Yongkun Li[1*], Wenzhe Zhu[1], Yinlong Xu[1,3], John C. S. Lui[2]

[1]*University of Science and Technology of China* [2]*The Chinese University of Hong Kong*
[3]*Anhui Province Key Laboratory of High Performance Computing, USTC*

Abstract

Serverless computing has revolutionized application deployment, obviating traditional infrastructure management and dynamically allocating resources on demand. A significant use case is I/O-intensive applications like data analytics, which widely employ the pivotal "shuffle" operation. Unfortunately, the shuffle operation poses severe challenges due to the massive PUT/GET requests to remote storage, especially in high-parallelism scenarios, leading to high performance degradation and storage cost. Existing designs optimize the data passing performance from multiple aspects, while they operate in an isolated way, thus still introducing unforeseen performance bottlenecks and bypassing untapped optimization opportunities. In this paper, we develop MinFlow, a holistic data passing framework for I/O-intensive serverless analytics jobs. MinFlow first rapidly generates numerous feasible multi-level data passing topologies with much fewer PUT/GET operations, then it leverages an interleaved partitioning strategy to divide the topology DAG into small-size bipartite sub-graphs to optimize function scheduling, further reducing over half of the transmitted data to remote storage. Moreover, MinFlow also develops a precise model to determine the optimal configuration, thus minimizing data passing time under practical function deployments. We implement a prototype of MinFlow, and extensive experiments show that MinFlow significantly outperforms state-of-the-art systems, FaaSFlow and Lambada, in both the job completion time and storage cost.

1 Introduction

Serverless computing, or simply "serverless", represents a transformative cloud-computing model that dramatically streamlines application deployment. Within this paradigm, the burdensome tasks of traditional infrastructure management recede into the background as cloud providers dynamically allocate resources, billing solely for the consumed computing power. Platforms such as AWS Lambda [5] and Azure Functions [31] exemplify this shift, facilitating the seamless execution of code in response to specific triggers. As we navigate the evolving expanse of cloud technologies, the prominence of serverless is undeniable, marking a significant change in the development, deployment, and scaling of modern applications. This shift becomes particularly noteworthy when considering I/O-intensive applications like data analytics [21, 22, 35]. Embedded in this analytical landscape are frameworks like Google's MapReduce [15] and Apache Spark [44].

[*]Yongkun Li is the corresponding author.

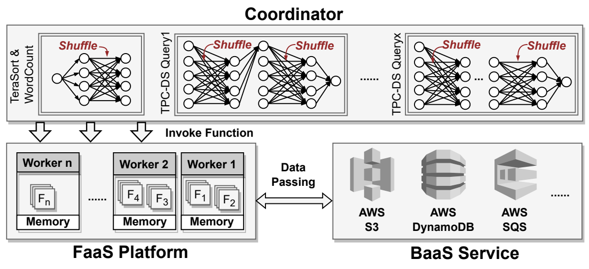


Figure 1: Serverless Computing Framework. *Circles represent sub-tasks, and each link between circles represents a data transfer, consisting of one PUT plus one GET.*

In data analytics, the "*shuffle*" operation is pivotal for data passing between stages. Notably, over half of Facebook's daily analytics entails at least one shuffle operation [45]. Given the stateless nature of serverless, data are largely passed through remote Object Stores like S3 [8], during which each pair of sender and receiver functions involve a PUT and a GET operation. However, shuffle's all-to-all connectivity, i.e., each function should pass its output to all functions in the next stage, usually leads to a huge number of PUT/GET requests, especially under high parallelism of functions. For instance, with 500 functions, one can anticipate 500×500=250,000 PUTs and an equal number of GETs, a total of 500,000 requests. Due to the request rate caps of S3, excessive PUT/GET operations risk exceeding these limits, causing prolonged delays. For example, in the Pocket framework, shuffle can dominate, taking up 62% of the time for certain jobs [22]. Worse yet, while S3's storage capacity is affordable, the cost tied to massive PUT/GET operations can escalate very high.

In data analytics, optimizing the shuffle operation has led to a myriad of solutions, each has its unique trade-off. Though these solutions propose diverse optimization strategies, they lack a comprehensive and integrative consideration, resulting in suboptimal performance. First, the approach of using private storage has been utilized [22, 35], wherein shuffle is conducted through self-maintained storage such as ElastiCache clusters [4]. While it offers enhanced shuffle speed by granting users exclusive ownership of the storage medium, the ensuing costs are considerably elevated. Additionally, the onus of intricate cluster management falls back on the developers, somewhat undermining the convenience of serverless computing. The method of leveraging intra-worker memory offers another alternative [13, 25], harnessing over-provisioned local memory in workers for faster shuffle operations. However, its applicability remains tethered to functions situated within the same worker, and due to the all-to-all data passing requirement between functions, only a small portion of data passing can be performed via the local memory of workers. Lastly, the technique of utilizing multi-level shuffle [32, 34], inspired by the mesh networks from HPC (High Performance Computing) [23], endeavors to streamline shuffle operations. Yet, it incurs multiplied data to be transmitted, making the bandwidth limit on the function side a new bottleneck, especially when the size of the data input is large. In conclusion, rather than offering a holistic solution, existing techniques operate in isolated realms, sometimes incurring unintended costs or introducing unforeseen bottlenecks. Moreover, the absence of a systematic exploration implies that potential optimizations still remain untapped.

In this paper, we propose MinFlow, a unified data passing framework for I/O-intensive analytics jobs atop serverless, which pinpoints globally optimal configuration to simultaneously achieve high performance and low cost. MinFlow contains the following key innovations:

- It optimizes the data passing topology by first segmenting functions into adaptive groups and then progressively converging the groups to get integrated multi-level topologies. This methodology not only greatly reduces the number of PUT/GET operations, but also provides the flexibility of selecting from a broader range of feasible topologies under real-world settings.
- It develops an interleaved partitioning strategy to optimize the function scheduling. Specifically, it partitions a multi-level topology into bipartite structures, and schedules functions in units of the bipartite sub-graphs so as to allow the localization of data passing within workers.
- It leverages a precise model to pinpoint the optimal configuration according to real function deployments, i.e., the best combination of topology and function scheduling, so as to simultaneously minimize the number of PUTs/GETs and the storage cost.

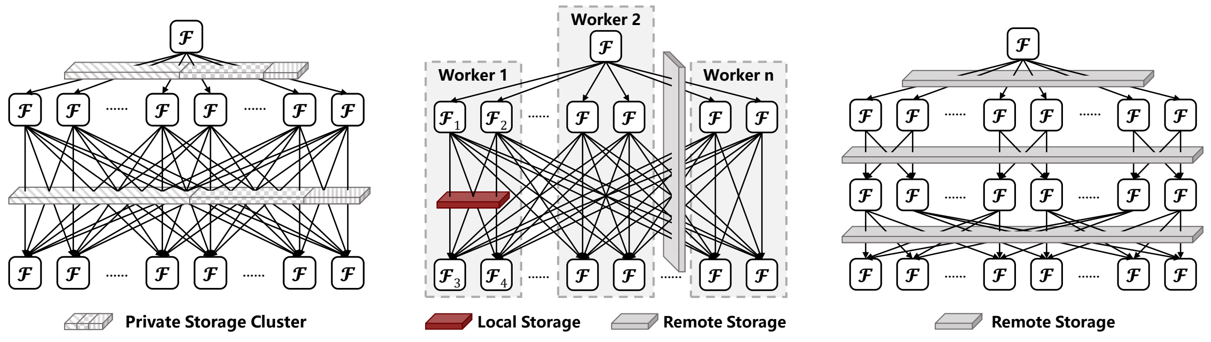
We implement a prototype of MinFlow open-sourced at https://github.com/lt2000/MinFlow and conduct extensive experiments based on Amazon cloud service. Our experiments using the benchmarks of TeraSort, TPC-DS, and WordCount show that MinFlow significantly outperforms state-of-the-art works in both the shuffling performance and storage cost. For example, in high-parallelism case of 600 mapper and 600 reducer functions for 200GB TeraSort, MinFlow reduces the shuffle time by 66.62% and 89.22%, compared to Lambada [32] and FaaSFlow [25], respectively, and it also reduces the storage cost by 86% and 98.71%, respectively.

2 Background and Motivation

2.1 Background

Serverless Computing Framework. As the building blocks of serverless, FaaS and BaaS (e.g., Amazon Lambda [5] and S3 object store [8]) respectively empower users to directly invoke predefined functions in containers and access remote back-end services via RESTful APIs. When employing serverless services, a common practice is first to decouple applications' states and compute logic, then delegate them to BaaS-side storage and FaaS-side functions separately (see Figure 1). Merits of the architecture are twofold. First, the separation of storage and computation and the containerized functions greatly facilitate scaling up/down compute resources as needed (e.g., to tackle bursty workloads). Second, it provides a fine-grained "pay-as-you-go" billing model that charges for actually used resources rather than the reserved amount; e.g., Amazon Lambda provides billing increments of one millisecond during function execution [6]. Due to all its virtues, an increasing number of applications have embraced the architecture, including Web, IoT, data analytics, etc. [7, 18, 31].

Data Analytics atop Serverless. Data analytics aims at efficiently processing huge amounts of data as specified to obtain desired results, and it has been employed in a wide range of domains, including scientific computing, machine learning, large-scale graph computations, etc. [39]. To offer essential scalability and fault-tolerance, mainstream data analytics frameworks [15, 30, 43] commonly adopt the bulk-synchronous-parallel model (BSP) [40], which divides a job into consecutive stages, each stage composed of parallel sub-tasks. When each stage is completed, the intermediate results are transferred to the next stage, via communication primitives such as *shuffle* and *broadcast* [14, 19] for further computation. Thereby, the workflow of jobs employing BSP can be represented as DAGs, as illustrated in Figure 1. To deploy data analytics jobs atop serverless platforms, users typically first declare the job's workflow to a coordinator using configuration files. Then, the coordinator assumes control, activating functions to perform consecutive stages sequentially, with sub-tasks within each stage executed by parallel functions. Users receive notifications when the whole job is completed.



(a) Shuffle via Private Storage **(b)** Shuffle via Intra-worker Memory **(c)** Multi-level Shuffle

Figure 2: Existing Approaches.

Notably, since serverless functions are unable to directly communicate with each other, data transmission between stages is realized via remote back-end storage, typically S3, rather than direct peer-to-peer (P2P) data passing. Since analytics workloads typically have a wide variance of resource needs over time [34, 35, 46], traditional physical/VMs deployments can frequently suffer from resource wastage or performance degradation. On the contrary, benefiting from the elasticity and fine-grained billing model of FaaS, the job's computation can be easily accelerated by splitting each stage into more sub-tasks assigned to parallel functions, at a significantly lower cost and higher performance. Thereby, a lot of research works have focused on running data analytics based on serverless [13, 20, 22, 25, 27, 32, 34, 35, 45].

2.2 Dilemma Caused by *Shuffle*

In data analytics, *Shuffle* is the most common primitive for passing data between adjacent stages. Prior research shows that more than 50% of daily data analytics jobs at Facebook involve at least one shuffle operation [45]. As shown in Figure 1, during the shuffle process, each sub-task in the previous stage distributes its output to all sub-tasks in the next stage. Such an all-to-all data passing method would greatly "break down" the intermediate results, causing proliferating requests: for instance, if the parallelism of stages is N, then at least $2N^2$ object PUTs/GETs are required to pass the intermediate results since there would be N^2 links between stages and each link represents a PUT plus a GET. This causes problems in two aspects. First, it significantly degrades the performance. Due to S3's request rate limit (3.5k and 5.5k req/s for writes and reads [10]), the quadratic $2N^2$ PUTs/GETs could easily be throttled, especially when N is large. Consequently, while the computation time could be slashed by improving the parallelism N, the entire data analytics process is significantly slowed down by shuffle. For example, in Pocket, over 62% of time is spent shuffling data, while computation only takes 17% of time for 100GB TeraSort [22]. Second, it drastically inflates the cost. Albeit S3 offers cheap storage (0.023 USD$ per GB/month), it incurs large access cost as it charges in increments of single request (0.005/0.0004 USD$ per 1k PUTs/GETs) [9]. As a result, the $2N^2$ PUTs/GETs would rapidly increase the cost as N goes up. In conclusion, both the elasticity and economy of serverless get severely impeded by the shuffle.

2.3 Existing Approaches

Existing approaches bypass or mitigate S3's throttling by (1) *performing Shuffle via a private storage cluster*, (2) *performing Shuffle via intra-worker memory*, or (3) *using multi-level Shuffle to decrease the number of PUTs/GETs*.

Shuffle via Private Storage. As a public cloud storage service shared by numerous users and applications, S3 inherently allocates a limited request rate to each single user, to guarantee fairness and avoid interference among tenants. A straightforward way to eliminate this restriction is to replace S3 with self-maintained private storage, for example, ElastiCache clusters [4]. This provides the user an exclusive ownership of the storage service, thereby greatly improving shuffle speed. However, losing S3's sharing economy and fine-grained billing model often leads to a significant increase in cost. As Pocket [22] suggests, the cost is 100 times higher than S3 for sorting jobs. Therefore, some remedies have been proposed to mitigate the surging cost, e.g., as shown in Figure 2(a), Pocket [22] and Locus [35] dynamically rightsizing resources, and combine high-end and cheap storage media to achieve better trade-offs between performance and cost.

Shuffle via Intra-worker Memory. Another way to bypass S3's throttling is to reclaim and leverage over-provisioned memory in workers to accelerate shuffle [13, 25]. More specifically, data passing between functions located in the same worker is performed via its local memory. Take functions F_2 and F_3 co-located in worker W_1 in Figure 2(b) as an example. Suppose data to be passed from F_i to F_j is denoted as $<F_i, F_j>$. To deliver data to F_3, F_2 first puts $<F_2, F_3>$ into W_1's local memory, then F_3 fetches $<F_2, F_3>$ immediately afterwards and finishes the transmission. This approach performs well in both performance and cost, since the reclaimed over-provisioned local memory not only offers much higher bandwidth and lower latency, but also does not incur extra overhead. The downside is the limitation in its applicability, as only co-located functions can adopt this approach [25].

Multi-level Shuffle. Borrowing ideas from HPC, which achieves all-to-all connections among processors through the k-dimensional Mesh Network [16], Starling [34] and Lambada [32] project shuffle-involved functions onto a k-dimensional mesh with a side length of $\sqrt[k]{N}$ where N refers to the number of functions, and applies the all-to-all collective primitive to subsets of functions, once for each dimension, to realize all-to-all shuffle among functions. Compared to direct data passing, such a multi-level indirect manner (*ML-Shuffle*) greatly decreases the number of requests, since each request loads a larger volume of data, and only one PUT plus one GET is required for each link. To be more precise, k-level shuffle (*kL-Shuffle*) reduces the number of requests from $2N^2$ to $2kN\sqrt[k]{N}$. For example, in Figure 2(c) we show a *2L-Shuffle* by setting k as 2. As can be seen, only 60 requests are needed, compared to 72 when directly connecting functions (see Figure 2(a)). Due to fewer requests that need to be transmitted through remote S3 during the shuffle, performance degradation caused by S3's throttling gets mitigated, and lower fees are charged as well.

2.4 Limitations

The aforementioned approaches face respective limitations in cost, performance, or applicability. First, to maintain the private storage for faster shuffle, users have to bear additional management works like resource scaling, fault tolerance, etc., which should have been undertaken by serverless, thus violating the easy-of-use principle. Besides, private storage still entails high costs. Although using dedicated NVMe storage seems cost-efficient, the need to mount NVMe devices to VMs [22] and the limited network bandwidth of VMs significantly hinder the speed advantage, requiring the allocation of numerous NVMe instances for performance. To illustrate this, we run the state-of-the-art KV database Apache kvrocks [11] on varying numbers of NVMe instances (EC2 i3.2xlarge, the same as Pocket uses) for shuffle. As depicted in Figure 3, increasing NVMe instances reduces shuffle time but at a significant cost. Pocket also reports that the cost of Pocket-NVMe is 40 times that of S3 for TeraSort [22].

Second, for performing shuffle via intra-worker memory (e.g., FaaSFlow [25]), it's only applicable to functions co-located at the same worker. For analytics jobs, each group of co-located functions represents a sub-graph in the whole workflow DAG, and all groups together make up the whole DAG. As a result, though functions in the same sub-graph can communicate through local memory, due to the all-to-all feature of shuffle, links between sub-graphs still dominate, which necessitates the use of remote storage for data passing. Worse yet, the benefits of memory-assisted data passing could be easily offset by the stragglers caused by slower remote storage. As shown in Figure 3, under different configurations,

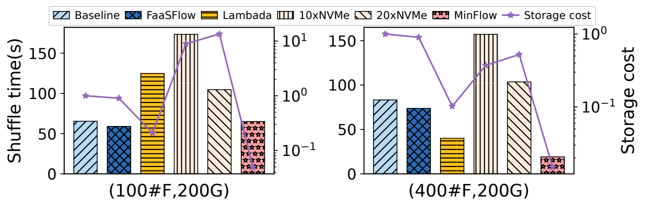


Figure 3: TeraSort Shuffle Time under Different Configurations. *Baseline transfers all intermediate data via S3 and the storage costs are normalized to the baseline.*

the shuffle time of FaaSFlow is only reduced by 9.94% to 11.39% than that of the Baseline.

Regarding *ML-Shuffle* (e.g., Lambada [32]), existing methods based on k-dimensional mesh suffer from an applicability problem, i.e., it mandates a symmetrical Mapper-Reducer setting, which means the number of Mappers and Reducers must be the same (e.g., both are N). Besides, while allowing to adjust the topology with different parameters (e.g., k), they merely set parameters arbitrarily and delegate the tricky task of choosing the optimal parameters to users, which easily leads to sub-optimal performance. For example, while larger k decreases the number of requests more significantly, it brings about multiplied extra data volume to be transferred. Because cloud vendors often assign limited network bandwidth to each function [12, 22, 32], such heavy traffic could exacerbate the problem. On the contrary, smaller k often comes with an unsatisfactory effect on reducing the number of requests. For example, under 100 functions and 200GB input data, the shuffle time of Lambada is 1.91× than that of the Baseline (see Figure 3). Moreover, the 2-level shuffle algorithm can not be easily applied to more levels, and the extension from the 2-level shuffle algorithm to a general k-level one is non-trivial (see §A.3).

Last, as shown in Figure 3, compared to the above three approaches, MinFlow achieves a high-performance and cost-efficient shuffle. We will further carry out extensive experiments to show the benefits of MinFlow in §4.

Inefficiency Analysis. Though a series of optimizing "actions" can be employed, for lack of a systematic understanding, there isn't a judicious "decision maker" that can use them collaboratively. Consequently, multiple factors together decide the efficiency of shuffle, e.g., DAG topology, function scheduling, transmission manner assignment, existing optimizations work in their respective single dimension, paying disproportionate expenses or leaving the rest as a bottleneck. Besides, even for each single dimension, the possible action space is still not fully explored. For instance, current *ML-Shuffle* directly migrates the k-dimensional mesh from the HPC field, whose applicability is strictly limited in the serverless scenario. Last, rather than carefully considering the characteristics of specific analytics jobs and environment variables to select the most appropriate choice, they often merely offer empirical value, e.g., k is set as 2. All these make existing approaches reach the sub-optimal configuration, leading to degraded performance/cost/ease-of-use.

2.5 Main Idea and Challenges

Main Idea. The key factors deciding analytics jobs' efficiency include function topology represented by the DAG, function scheduling, and the data transmission media. Compared to considering them separately, optimizing them in a unified way greatly helps find the optimal configuration, so as to eradicate bottlenecks from the whole workflow. For instance, *ML-Shuffle* facilitates traffic localization through local memory since the links at each level are more sparse and functions can be co-located more easily to avoid cross-worker data transmission. Also, traffic localization largely absorbs the additional traffic volume induced by *ML-Shuffle*. Therefore, for any given analytics job, our main idea is to first construct the whole configuration space by considering all three dimensions, then derive the optimal configuration from the space, based on user requirements, the task's characteristics, and the serverless platform's rate limit and billing rules.

Challenges. To realize the above idea, we mainly face the following challenges.

- **Constructing *ML-Shuffle* topology space.** To ensure applicability, we must be able to construct the complete topological space for any analytics job, including those with an asymmetric setting of Mappers and Reducers, despite the conventional mesh-based method supposes $\#mapper = \#reducer = N$ and only provides a concrete algorithm for 2-level shuffle[†] . Plus, for a specified analytics job, the complete *ML-Shuffle* topology space contains a number of possible combinations. Thus we need to efficiently construct the *ML-Shuffle* topology space with low overhead.
- **Function co-location and data transmission.** For each possible topology in the space, we need to carefully assign functions to workers to maximize the proportion of leveraging local memory for data passing, while simultaneously ensuring load-balance among workers and avoiding stragglers. This process is equivalent to that of searching for a partitioning scheme that divides the whole DAG into sub-graphs consisting of co-located functions in accordance with requirements, which is an NP-hard problem and especially time-consuming when the number of functions is large.
- **Finding the optimal configuration.** To select the optimal configuration from the space, we need to precisely model the mapping from each configuration to its performance and cost. To achieve this, we must take multiple key factors into consideration, e.g., the analytics job's intermediate data volume and the number of I/O requests, functions' network bandwidth, and remote storage's request rates, some of which can only be obtained at runtime, or be dependent on specific platforms.

[†] $\#mapper$ and $\#reducer$ are the number of mappers and reducers, and N is a positive integer.

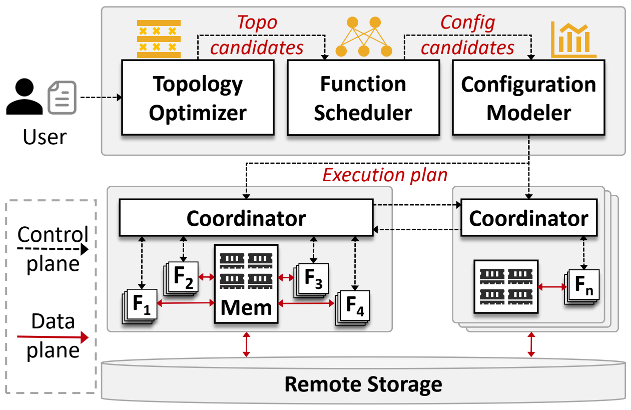


Figure 4: Overview of MinFlow Architecture.

3 MinFlow Design

To optimize the data passing between functions, we propose MinFlow, a unified data passing framework for analytics jobs atop serverless platforms, which seeks the global optimal configuration to achieve high performance and low cost simultaneously. We first introduce the overall architecture (§3.1) and elaborate on each technique in detail (§3.2-§3.4).

3.1 Overview

As Figure 4 illustrates, MinFlow resides in the cloud-side control plane, generating appropriate configuration for specific analytics jobs upon receiving user-submitted workflow specifications which delineate data passing paths between functions and the communication operators executed by these functions. Then, the coordinators deploy and run the task accordingly, upon FaaS and BaaS platforms. Specifically, MinFlow consists of three key components that work collaboratively to meet this goal while tackling the aforementioned challenges at the same time. A brief introduction of the components and their interaction is as follows.

- **Topology Optimizer.** For a given analytics job, it generates equivalent multi-level topologies based on the original single-level topology, via a novel progressively converging method to sidestep the inherent applicability downside of the mesh-based approach. More specifically, all candidates for the ultimate optimal topology, i.e., those with the fewest edges for each possible level, are rapidly constructed by a dynamic programming algorithm, while others are ignored.
- **Function Scheduler.** For each generated candidate topology, the Function Scheduler decides which functions should be co-located at the same worker and passes data through local memory, by dividing the complete topology into subgraphs. The partitioning must simultaneously achieve load balance, cross-worker traffic minimization, and straggler

avoidance. To solve the NP-hard problem, MinFlow employs a heuristic algorithm to find the near optimal solution quickly.

- **Configuration Modeler.** Configuration Modeler selects the optimal configuration, i.e., that with the shortest estimated completion time and lowest cost for data passing, among candidates. At its core is a mathematical model that factors in key variables, including serverless platform features and analytics job characteristics, to achieve high estimation accuracy. In particular, for those variables needed to be obtained at runtime, it determines them by an efficient and lightweight sampling method.

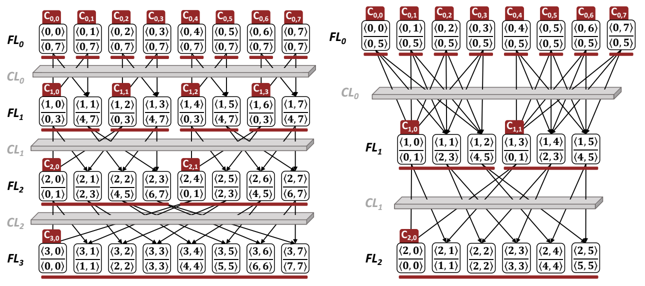
Note that though our current design follows FaaSFlow's distributed function coordination, which employs multiple coordinators to prevent function scheduling from becoming the bottleneck (see Figure 4), MinFlow also applies to the more conventional architecture with a centralized coordinator.

3.2 Topology Optimizer

Progressively Converging ML-Shuffle. Following the mesh-based *ML-Shuffle*, the progressively converging *ML-Shuffle* attempts to generate optimized topology, which is equivalent to the original single-level shuffle directly linking all pairs of mappers and reducers, by adding intermediate functions to reduce the number of required links. In contrast, it aims to offer essential flexibility to search in the complete feasible space for the optimal topology, instead of only providing a single sub-optimal topology as the mesh-based method does [32,34]. Moreover, it's a general k-level shuffle algorithm that allows an asymmetric number of mappers and reducers.

The key idea behind the progressively converging *ML-Shuffle* is a "divide and conquer" strategy. To avoid ambiguity, we clarify that a *kL-Shuffle* network consists of $k+1$ function levels (denoted as *flevel*) and k communication levels (denoted as *clevel*), and Figure 5(a) shows a *3L-Shuffle* network involving four *flevels* and three *clevels*. Rather than projecting functions to a rigid k-dimensional grid, progressively converging first divides functions in the first *flevel* into groups of the same size (initially one) and gradually lets them converge into larger groups in the next *flevel*, while preserving the full connection between each group and its upstream mappers, until all functions in the last *flevel* exist in the same group, thus ultimately achieving global all-to-all connection. For example, as Figure 5(a) illustrates, to build a three-level shuffle when $\#mapper = \#reducer = 8$, functions are respectively divided into 8, 4, 2, and 1 group for *flevel* 0, 1, 2, 3. Suppose we let $C_{i,j}/F_{i,j}$ denote the j-th group/function at *flevel* i, the data in $C_{0,0}$ in turn passes into $C_{1,0}$, $C_{2,0}$, and $C_{3,0}$. Analogously, the data in $C_{0,7}$ passes into $C_{1,3}$, $C_{2,1}$, and $C_{3,0}$. The rest are similar.

More generally, to derive an L-level topology comprising $L+1$ function levels, with each *flevel* having N functions, we divide the functions of the i-th *flevel* into g_i groups, where



(a) Regular Topo **(b)** Irregular Topo

Figure 5: Progressively Converging ML-Shuffle Topology. *Squares represent functions, and the upper and lower tuples inside functions respectively represent the function's id and the data's range.*

$g_i(\ 0 \le i \le L)$ meets the following conditions:

$$g_0 = N,\ g_L = 1,\ g_i = d_i \times g_{i+1} \text{ where } d_i \in \mathbb{N}^+ \backslash \{1\}. \quad (1)$$

When converging groups, to preserve the full connection between the new group in the *flevel* $i+1$ and its upstream mappers, for each function in the new group, a unique path between it and any upstream group in the *flevel* i must be guaranteed. To achieve this, we set the receiver functions of $F_{i,j}$ as R:

$$R = \{F_{i+1,k} | \lfloor k/s_{i+1} \rfloor = \lfloor j/s_{i+1} \rfloor \wedge \lfloor k\%s_{i+1}/d_i \rfloor = j\%s_i\}, \\ \text{where } s_i = \lfloor N/g_i \rfloor,\ s_{i+1} = \lfloor N/g_{i+1} \rfloor,\ d_i = \lfloor g_i/g_{i+1} \rfloor. \quad (2)$$

For example, as shown in Figure 5(a), when converging $C_{1,0}$ and $C_{1,1}$ into $C_{2,0}$, we link $F_{1,0}$ to $\{F_{2,0}, F_{2,1}\}$, $F_{1,1}$ to $\{F_{2,2}, F_{2,3}\}$, $F_{1,2}$ to $\{F_{2,0}, F_{2,1}\}$, and $F_{1,3}$ to $\{F_{2,2}, F_{2,3}\}$, thus preserving the full connection between $\{F_{2,0}, F_{2,1}, F_{2,2}, F_{2,3}\}$ with their upstream mappers $\{F_{0,0}, F_{0,1}, F_{0,2}, F_{0,3}\}$. As we see, the link distribution is also kept even to balance the transmission load among functions. Moreover, to ensure the correctness of data passing, each function must carefully partition and distribute received data to the next *flevel* functions. For function $F_{i,j}$, it first shards its data into $|R|$ continuous and equal-sized parts, then orderly assigns them to the receiver functions, i.e., functions in its R. For instance, as illustrated in Figure 5(a), $F_{0,0}$ shards its data $\langle 0,7 \rangle$ into two parts $\langle 0,3 \rangle$ and $\langle 4,7 \rangle$, and passes $\langle 0,3 \rangle$ and $\langle 4,7 \rangle$ to $F_{1,0}$ and $F_{1,1}$, respectively.

Notably, the flexibility of the progressively converging method lies in the setting of $D = \{d_i | 0 \le i \le L-1\}$, since it determines $G = \{g_i | 0 \le i \le L\}$ and any G that satisfies condition Equation (1) corresponds to a unique valid multi-level topology. In other words, by adjusting D we can easily derive a space containing multiple optional topologies, which may vary in edge distribution and the number of *clevels* and thus have different preferences for the number of requests, data transmission volume, etc. For example, the data passing volume is obviously proportional to L since each additional *clevel* incurs one more intermediate data transmission, and combining Equation (2) we have that the number of edges is $N \times \sum_{i=0}^{L-1} d_i$, by doubling which we can get the number of PUTs/GETs. Actually, the space covers those topologies generated by conventional mesh-based methods. And it can be proven that supposing N can be decomposed as the product of p prime factors, the space size $SS = \sum_{j=1}^{p} \sum_{i=1}^{j} \frac{(-1)^{j-i} i^p}{i!(j-i)!}$ (e.g., $SS = 115975$ when $p = 10$ and the detailed proof in §A.1). Such selectivity greatly facilitates seeking the most appropriate topology for an analytics job. Later, we will detail how to select the appropriate topology from the space, by carefully setting D.

Note that our approach may not work well in some corner cases, especially under prime function parallelism N. However, this problem can be easily addressed by allowing slight adjustments to the number N within a given bound, i.e., $[N-\alpha, N+\alpha]$. In our paper, we select $\alpha = 3$, and we will provide a detailed discussion on the impact of α in §4.5.

Last, compared to the mesh-based approach, the applicability gets significantly improved as well. As depicted in Figure 5(b), our approach even works for an asymmetric number of senders and receivers ($\#mapper = 8 \neq \#reducer = 6$), provided we keep the intermediate *flevel* the same size as the Reduce *flevel*, and link functions as Equation (2) suggests.

Candidates for Optimal Topology. For the Topology Optimizer, not provided with essential information (like function scheduling plan, data transmission manner, and other runtime states) to predict resulting completion time precisely, simply deciding the best topology by completion time is a rub. On the other hand, indiscriminately outputting all possible topologies forces all of them to go through all modules, incurring high overhead. Thus we adopt a middle-ground solution, i.e., to first select a small set of candidates, based solely on a comparison between their topological structure, then relegate the ultimate decision-making for the best to subsequent modules. In particular, though it's hard to directly find a total order for topologies' structure, comparison between them can be summarized as the following cases:

- Case 1. Under the same L, the topology with the fewest edges has the shortest completion time and data passing cost, as it transfers the data with the fewest PUTs/GETs that are more promptly processed by remote storage service.
- Case 2. Under different L, the comparison could be ambiguous, since on the one hand, larger L reduces edges, thus the number of PUTs/GETs. On the other hand, it transmits the intermediate data L times, potentially throttled by the function's network bandwidth.

Therefore, based on the partially ordered comparison, we add the locally optimal topology under each possible L, i.e., the one with the fewest edges, to our candidate set. Recall that the number of edges is $N \times \sum_{i=0}^{L-1} d_i$. Then suppose N can be decomposed into p prime factors, which means feasible L lies in $[1, p]$, candidate selection can be transformed into a series of optimization problems as follows:

$$\text{For } L \in [1,p], \begin{cases} \text{minimize} & N \times \sum_{i=0}^{L-1} d_i \\ \text{subject to} & \prod_{i=0}^{L-1} d_i = N,\ d_i \in \mathbb{N}^+ \backslash \{1\} \end{cases} \quad (3)$$

We propose a dynamic programming algorithm to solve these problems at once. Let $MinSum(i, j)$ denote the minimized sum of factors when factorizing i into j factors. Then we need to find $MinSum(N, L)$ for $L \in [1, p]$. The state transition equation is as follows:

$$MinSum(i,j) = \begin{cases} \min_{n|i}(n + MinSum(i/n, j-1)) & j > 1 \\ i & j = 1 \end{cases} \quad (4)$$

As we see, the equation formulates the value of $MinSum(N, L)$ recursively in terms of its sub-problems. Thus we employ a bottom-up dynamic programming approach, i.e., iteratively solving $MinSum(i, j)$ with smaller i and j first and use their solutions to arrive at solutions to bigger i and j. More specifically, we can calculate all $MinSum(N, L)$ for $L \in [1, p]$ in a nested loop. In the inner loop, i starts from 1 to N, while in the outer loop, j progresses from 1 to p. Along the way, all desired $MinSum(N, L)$, $1 \leq L \leq p$ gets solved. Moreover, we use $Sol(i, j)$ to track the decomposition path of $MinSum(i, j)$, i.e., $Sol(i, j > 1) =$ the selected n of $MinSum(i, j)$ in Equation (4) and $Sol(i, 1) = i$. Then by iteratively putting $Sol(i, j)$ along the decomposition path of $MinSum(N, L)$ into a sequence, we can get the desired $D = \{d_i | 0 \leq i \leq L-1\}$, by which we can easily derive the corresponding L-level topology with the fewest edges.

3.3 Function Scheduler

The Function Scheduler assigns a scheduling plan to each of the candidate topologies, indicating when and on which worker each function should be invoked. While the "when" question is straightforward to deal with by monitoring the completion time of functions and following the data dependency between functions, the latter "where" question must be treated carefully to satisfy several important and interacting requirements. Next, we first formulate the problem and then demonstrate how to solve it.

Problem Formulation. The function placement problem is equivalent to partitioning the whole DAG into sub-graphs, where functions within each sub-graph must be co-located to pass data via local memory, while different sub-graphs are placed independently and communicate via remote storage. Then our goal is to search for a partitioning scheme that satisfies the following requirements:

1) *Traffic localization.* Since functions within sub-graphs are co-located and communicate via faster local memory, the resulting sub-graphs should include edges in the DAG as much as possible, to localize more traffic and thus accelerate the data passing.
2) *Transmission straggler avoidance.* Due to the synchronization barrier of the BSP model, the duration of each *clevel*'s transmission is decided by the slowest edge. Thus edges in the same *clevel* should be either all included in sub-graphs or not included at all, to avoid the benefit of faster local memory being offset by stragglers caused by remote storage.

3) *Load balancing.* The DAG must be partitioned until all subgraphs width, i.e., the number of functions in the *flevel* with the most functions, must be capped to ensure functions' computing and communication load can be easily spread among workers at all *flevels* and *clevels*.

Interleaved Graph Partitioning. As §2.4 suggests, the above requirements are contradictory in the original single-level and all-to-all topology. Surprisingly, the progressively converging *ML-Shuffle* brings an opportunity to achieve them simultaneously. Since it transforms the rigid all-to-all connection into multiple levels of more sparse connections, each *clevel* has the favorable feature as follows.

Theorem 1. *For a multi-level topology generated by the progressively converging method (the parallelism is N), the i-th level of links corresponding to factor d_i, along with two adjacent flevels of functions, can be divided into $\frac{N}{d_i}$ disjoint complete bipartite graphs with width d_i.*

Proof. According to Equation (2), $F_{i,m}$ and $F_{i,n}$, where $\lfloor \frac{m}{s_{i+1}} \rfloor = \lfloor \frac{n}{s_{i+1}} \rfloor$ *and* $m \equiv n(mod\ s_i)$, have the same receiver functions R. Hence, the functions at *flevel i* can be categorized into $\frac{N}{d_i}$ conjugacy classes, with the elements within each class sharing the same R. Each conjugacy class at *flevel i* and its R at *flevel* $i+1$ together constitute a complete bipartite graph with width d_i (detailed proof in § A.2). □

From Theorem 1, any *clevel* can be decomposed into isolated Complete Bipartite Graphs (CBGs), which are ideal units for function co-location, since all edges in the *clevel* are evenly included by same-sized CBGs as shown in Figure 6. In other words, by putting functions within each CBG to the same worker, data transmission of the *clevel* can be done via workers' memory instead of remote storage, greatly accelerating data passing. Meanwhile, different CBGs can be placed arbitrarily, without the need to be co-located.

So far we've found an excellent way to place functions for each individual *clevel*, yet the method can't be directly generalized to function placement for the whole multi-level graph. Since the communication of adjacent *clevels* involves a shared *flevel*, e.g., *clevels* 0 and 1 both involve *flevel* 1 (see Figure 6), co-location constraints of two *clevels* must be met at once, which leads to multiplied width of co-location units. Worse yet, when jointly considering all *clevels*, due to the cascade effect, functions in the whole graph must be co-located to the same worker, violating the load balance requirement. To address the problem, we employ an interleaved partitioning strategy, to decouple the tightly bound *clevels* so as to solve them independently. Specifically, it removes edges in all odd-numbered *clevels*, delegating that portion of data passing to remote storage. The rationality lies in the following corollary, which could be easily derived from Theorem 1.

Corollary 1. *For a k-level topology generated by progressively converging method (the parallelism is N), where each*

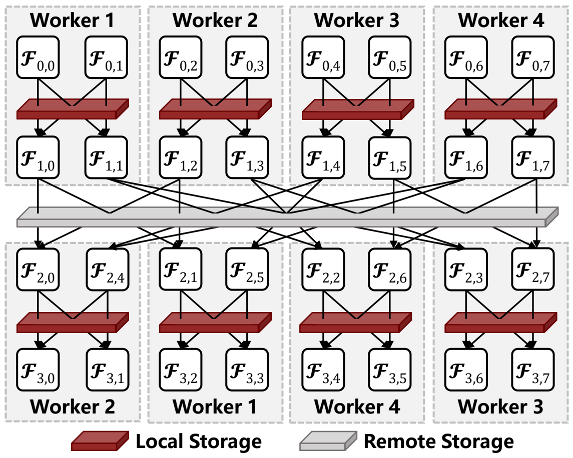

Worker 1
Worker 2
Worker 3
Worker 4
$\mathcal{F}_{0,0}$ $\mathcal{F}_{0,1}$ $\mathcal{F}_{0,2}$ $\mathcal{F}_{0,3}$ $\mathcal{F}_{0,4}$ $\mathcal{F}_{0,5}$ $\mathcal{F}_{0,6}$ $\mathcal{F}_{0,7}$
$\mathcal{F}_{1,0}$ $\mathcal{F}_{1,1}$ $\mathcal{F}_{1,2}$ $\mathcal{F}_{1,3}$ $\mathcal{F}_{1,4}$ $\mathcal{F}_{1,5}$ $\mathcal{F}_{1,6}$ $\mathcal{F}_{1,7}$
$\mathcal{F}_{2,0}$ $\mathcal{F}_{2,4}$ $\mathcal{F}_{2,1}$ $\mathcal{F}_{2,5}$ $\mathcal{F}_{2,2}$ $\mathcal{F}_{2,6}$ $\mathcal{F}_{2,3}$ $\mathcal{F}_{2,7}$
$\mathcal{F}_{3,0}$ $\mathcal{F}_{3,1}$ $\mathcal{F}_{3,2}$ $\mathcal{F}_{3,3}$ $\mathcal{F}_{3,4}$ $\mathcal{F}_{3,5}$ $\mathcal{F}_{3,6}$ $\mathcal{F}_{3,7}$
Worker 2
Worker 1
Worker 4
Worker 3
Local Storage
Remote Storage

Figure 6: Function Scheduling.

level has the corresponding factors d_0, d_1, ..., d_{k-1}, if we remove edges of all odd-numbered levels (1,3, ..., $\lfloor \frac{k}{2} \rfloor \times 2 - 1$), the whole graph can be divided into disjoint CBGs with width lying within D_{even} = {d_0,d_2, ..., $d_{\lfloor \frac{k-1}{2} \rfloor \times 2}$}

The interleaved approach (see Figure 6) acts as a heuristic algorithm to solve the NP-hard graph partitioning problem [17,24,25], by giving quick solutions that fit the aforementioned requirements. Specifically, due to at least half of the *clevels* being left for local memory to perform data passing, performing function placement in units of resulting CBGs localizes over 50% of overall traffic. Meanwhile, since transmission media (i.e., local memory or remote storage) is assigned in an interleaved manner, no communication straggler exists during the job's execution. Last, it allows us to selectively decide which factors in D would be put in D_{even}, to minimize the resulting CBGs' width, thus achieving a fine-grained function placement that facilitates load balancing.

3.4 Configuration Modeler

Topology Optimizer (§3.2) has offered a group of candidate multi-level topologies, and Function Scheduler (§3.3) provided each with its appropriate function scheduling and placement scheme. Configuration Modeler's responsibility is to select the optimal one out of them. Since the additional function level of a multi-level topology only works to assist communication and doesn't change the job's computing time, Configuration Modeler opts to choose the one with the shortest overall data passing time.

To achieve this, the Configuration Modeler must precisely model each configuration's resulting passing time. As discussed in §2.4, during each *clevel*'s communication, either the function side or the storage side acts as the real bottleneck, depending on the actual number of requests and data volume. More specifically, the duration would be $T_i = 2 \times \max(T_{i,f}, T_{i,s}), 0 \le i \le L-1$, where $T_{i,f}$ and $T_{i,s}$ respectively represents the time spent on function putting/fetching data in fixed rate and storage side processing received requests, and since the two parts are overlapping, we take the maximum of them. The reason behind the multiplier 2 is that each *clevel*'s communication includes sender functions writing to the storage side plus receivers reading back. Due to S3-based and memory-based communication alternating in different *clevels*, caused by the interleaved transmission media assignment (see §3.3), $T_{i,f}$ and $T_{i,s}$ are modeled differently in the two types of *clevels*. Suppose the function number is N in each *flevel*, in the S3-based *clevel* we have $T_{i,f} = \frac{D_i}{N*b_f}$ where D_i/N and b_f are each function's transmitted data volume at *flevel* i and bandwidth ceiling respectively, and $T_{i,s} = \frac{R_i}{q_s}$ where R_i is the involved number of requests at *clevel* i and q_s is S3's request rate. In contrast, for the memory-based *clevel* $T_{i,f} = \frac{D_i}{M*b_t}$ and $T_{i,s} = \frac{R_i}{M*q_t}$, where M is the number of cluster nodes, b_t and q_t respectively are the bandwidth ceiling and I/O rate limit of Tmpfs [1] which we leverage to establish the elastic reclaimed-memory file system. To summarize, the overall data transmission time of a multi-level network is:

$$T = 2 * \sum_{i=0}^{L-1} \begin{cases} \max(\frac{D_i}{N*b_f}, \frac{R_i}{q_s}), & i \text{ is odd}. \\ \max(\frac{D_i}{M*b_t}, \frac{R_i}{M*q_t}), & i \text{ is even}. \end{cases} \tag{5}$$

Except for the data volume D_i, other parameters in Equation (5) can be obtained before running the job. Yet D_i is only available by the runtime, preventing choosing the optimal configuration before the job runs. We use a sampling and profiling method to address the problem. As there is commonly a linear, or a non-linear but deterministic relationship between the size of input data and intermediate data [33], and D_i keeps consistent for all *clevels*, Configuration Modeler repeatedly samples the original input with different sizes and executes the job, recording the amount of intermediate data. Then each time it gets a new <input data size, intermediate data size> pair. By fitting these pairs using a curve, Configuration Modeler can estimate the intermediate data size under the whole input. Thus, by bringing all parameters into Equation (5), the Configuration Modeler predicts the transmission time of all candidate configurations, and selects the fastest one.

4 Evaluation

4.1 Experiment Setup

TestBed. We deploy our FaaS framework on 10 Amazon EC2 m6i.24xlarge instances, each with 96 vCPUs, 384GB memory, and 37.5 gigabits/s bandwidth, and we adopt Amazon S3 as the remote storage. All compute instances run Ubuntu 22.04 LTS with Linux kernel 5.15.0. For our FaaS framework, similar to FaasFlow, we run self-maintained functions within docker containers (24.0.6 version), rather than directly adopting function services that are not transparent to us, so as to better manage functions' execution and lifetime.

Workload. We adopt three widely employed benchmarks involving shuffle operations, ranging from typical MapReduce-style tasks to SQL-style queries.

- *TeraSort.* Sorting a dataset based on the specified key.
- *TPC-DS-Q16.* TPC-DS consists of multiple SQL queries. Among them, we select the most data-intensive one, i.e., the 16th query that performs a large joining via shuffle.
- *WordCount.* Counting word frequency in documents.

The datasets of the above workloads are respectively generated by Sort Benchmarks Generator [2], TPC-DS Tools [3], and Purdue MapReduce Benchmarks Suite [36].

Comparison. We compare MinFlow (denoted as MF) with the basic practice and two state-of-the-art works, in terms of both performance (execution time) and cost (fees charged).

- *Baseline.* The most common and straightforward approach, i.e., all intermediate results during shuffle are transferred through remote S3 object store, denoted as BL.
- *FaaSFlow.* FaaS framework with state-of-the-art function scheduling mechanism, which transmits intermediate data via local storage within workers, referred to as FF.
- *Lambada.* State-of-the-art topology optimizing method, performing multi-level shuffle to reduce PUTs/GETs to S3. We select its optimal configuration and denote it as LBD.

Configuration. During all our experiments, we set the resource limit of each function as 2 CPU, 3GB memory, and 75MB/s bandwidth, similar to prior research works [21, 26, 41, 47], to simulate a common setting of Amazon's commercial function service Lambda. By default, we respectively set the input size as 100GB and 200GB, and set the parallelism as 400 functions and 600 functions, since MinFlow mainly focuses on processing massive datasets with a large number of functions, which is in accordance with the serverless paradigm's goal to support hyper-scale computation with its superior scalability. Besides, we extensively adjust the input size and parallelism to show MinFlow's performance under broader settings (see §4.5).

4.2 Microbenchmark Results

Shuffle Time & Storage Cost. Now we evaluate MinFlow's effectiveness in improving the shuffle speed and saving the storage cost. Figure 7, 8 and 9 shows the shuffle time and normalized storage cost (divided by the BL's storage cost) of all approaches under three different workloads.

Taking TeraSort as the example (Figure 7, we first focus on the 600-function parallelism with 200GB input data size, the BL takes nearly 180s to finish the shuffle operation. During shuffling, not only do all functions await, but the bill for using functions continues increasing as function services usually charge based on time (e.g., in increments of 1ms). Besides, compared to BL, FF only slightly reduces the shuffle time and storage cost by 14.45% and 9.98%, respectively, since most of

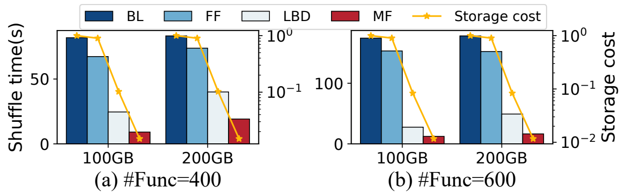


Figure 7: Shuffle Time of TeraSort.

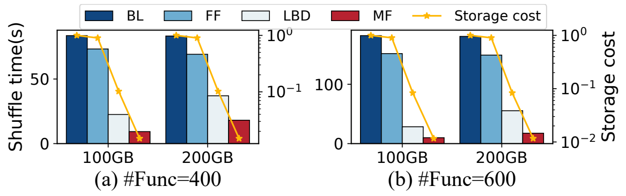


Figure 8: Shuffle Time of TPC-DS.

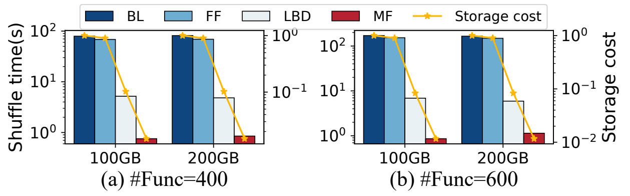


Figure 9: Shuffle Time of WordCount.

PUTs/GETs (324000 out of 360000 = 90%) carrying intermediate data still go through remote S3 and only a tiny portion (the rest 10%) can be performed via local storage, which we presented the reason in §2.4. In contrast, LBD could greatly accelerate the shuffle process with much less storage cost. Specifically, it shortens the shuffle time by over 72.36% and 67.69% compared to BL and FF, meanwhile saving the storage cost by 91.68% and 90.76%, respectively. Nevertheless, LBD still experiences ~50s idle time to wait for the shuffle's completion. As for MinFlow, it outperforms all the competitors in terms of performance by slashing the shuffle time to 16s. Compared to BL and FF, MinFlow achieves 10.8× and 9.3× shuffle acceleration respectively, and even compared to LBD, MinFlow still achieves 3× faster shuffle speed. In addition, MinFlow greatly saves the storage cost (98.84%, 98.71%, and 86% compared to BL, FF, and LBD), for it not only greatly reduces the number of PUTs/GETs but also largely eliminates additional intermediate data volume via local storage. Under 400-function TeraSort with 200GB input data size, similar to the 600-function parallelism, MinFlow preserves considerable performance and cost improvement – as Figure 7(a) shows it achieves 2.1× shuffle acceleration and 85.37% cost saving compared to LBD. Yet one noticeable change is that the performance benefit of MinFlow over BL and FF shrinks, although still reaches as high as 76.99% and 74.03%, respectively. It's because the performance degradation caused by excessive PUTs/GETs alleviates under lower parallelism. We will conduct more in-depth experiments about this phenomenon later (see §4.5).

Aside from the above TeraSort, Figure 8 and 9 show the

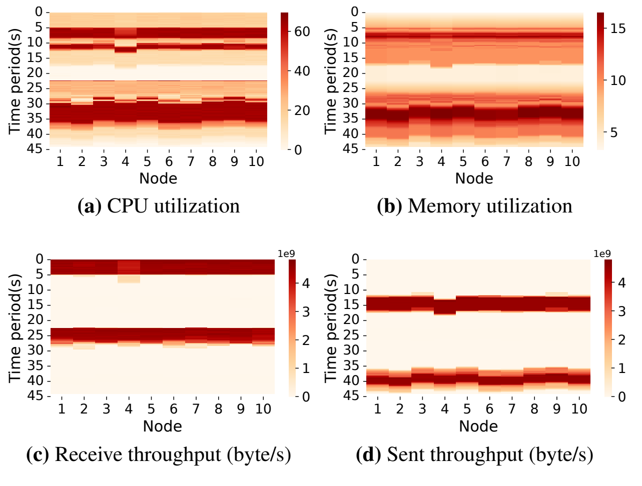


Figure 10: Load Balance. *TeraSort,600#F,200GB.*

results of TPC-DS-Q16 and WordCount. It can be found that while the results of TPC-DS-Q16 are quite similar to those of TeraSort, MinFlow shows much higher performance benefit over other approaches when running WordCount, as shown in Figure 9 where the vertical axis is logarithmic to more clearly present the shuffle time. For example, under 600 functions with 200GB input data, compared to BL and FF, MinFlow reduces the shuffle time by as high as 99.31% and 99.23%. Such phenomenon can be explained from the aspect of intermediate data size – while TPC-DS-Q16 and TeraSort share a common characteristic that the intermediate data size of shuffle is consistent with the input data size, WordCount has much less intermediate data since duplicate words in the input would be eliminated with a counter.

Load Balance among Workers. Load balance has always been a necessity for large-scale distributed systems since it directly determines systems' resource efficiency and quality of service. To demonstrate MinFlow's capability in load balancing, we count the load of each worker every 50ms to show a fine-grained resource usage of workers. Figure 10(a), 10(b), 10(c) and 10(d) respectively show the CPU usage, memory occupation, and traffic load of all 10 workers when running TeraSort under 600 functions and 200GB input data with MinFlow, where the brighter red represents higher load. First, as we can see, all types of loads are kept even among workers throughout the process. Second, the load intensity of each worker varies noticeably along the timeline, which is in line with the BSP model's characteristic that compute/memory and traffic peaks appear alternatively. For example, the compute and memory peaks indicated by the bright-red "stripes" in Figure 10(a) and 10(b) represent the positive correlation between compute and memory peaks. On the other hand, the peaks of incoming traffic occur at around 0-5s (input) and 22.5-27.5s (GETs of remote storage shuffle), and peaks of outgoing traffic occur at around 12.5-17.5s (PUTs of remote storage shuffle) and 37.5-42.5s (output). They are both interleaved with the above compute/memory peak. In addition, by

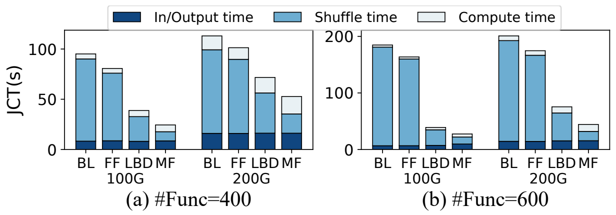


Figure 11: Overall Time of TeraSort.

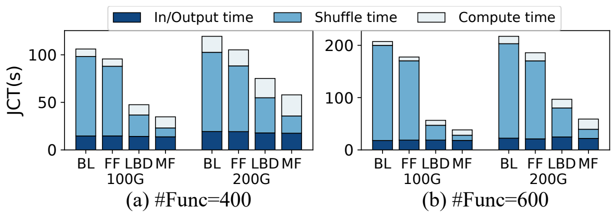


Figure 12: Overall Time of TPC-DS.

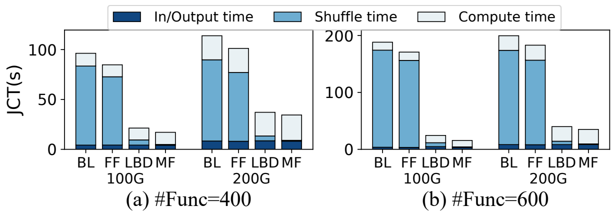


Figure 13: Overall Time of WordCount.

combining this set of results, we could find that for MinFlow the remote storage shuffle only accounts for ~10s out of the overall ~45s job running time, partly verifying MinFlow's high shuffle speed.

4.3 Overall Performance Analysis

Figure 11, 12 and 13 show the overall job completion time under the same setting as in §4.2. We still first take the 600-function with 200GB input data group as the example. In terms of the overall job completion time, as we can see, compared to other approaches MinFlow could contribute 41.35%-77.98% improvement for TeraSort workload, 39.12%-72.86% for TPC-DS, and 12.26%-82.46% for WordCount. The improvement mainly comes from shuffle time reduction, since among the compute, shuffle, and input/output time only the shuffle time changes significantly, while the other two parts basically remain constant across all approaches.

Among three workloads, for TeraSort and TPC-DS that have same-sized intermediate data with their input, shuffle time plays a non-neglectable part throughout all compared approaches. For example, in the TeraSort group BL, FF, and LBD respectively spend 88.65%, 87.23%, and 65.24% time on shuffling. By employing MinFlow the proportion could be reduced to 37.13%, contributing to not only more efficient computation but also better function use, since functions fees are charged in increments of time units, say 1ms. However, when it comes to WordCount, though the overall time improvement is still significant compared to BL and FF, the benefit over LBD shrinks, as Figure 13 shows. The reason is WordCount's reduction in intermediate size for word deduplication – While this does not noticeably impact BL and FF's overall time for their bottleneck lies in the excessive number of PUTs/GETs instead of the transmitted data volume, it greatly alleviates LBD's bottleneck that is mainly caused by function bandwidth limit. As a result, LBD's shuffle time only accounts for 14.83% of the overall job execution time, largely neutralizing the great shuffle time improvement of MinFlow over LBD.

Last, since theoretically the compute and input/output speed are proportional to the function number, increasing the parallelism can easily slash both compute and input/output time, which does not hold for shuffle time. Therefore shuffle time accounts for a higher portion under high parallelism, providing more optimization space. For example, as Figure 11 shows, under 200GB input data size, compared with 400-function parallelism in which MinFlow achieves 26.49%-53.38% overall time reduction, in 600-function parallelism the values increase to 41.35%-77.98% respectively. To conclude, these results demonstrate that MinFlow could improve the overall time considerably compared to existing works throughout all three workloads.

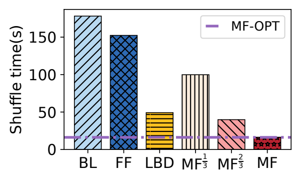


Figure 14: Breakdown.

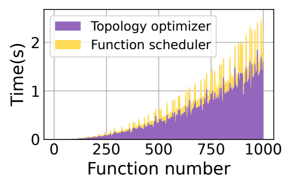
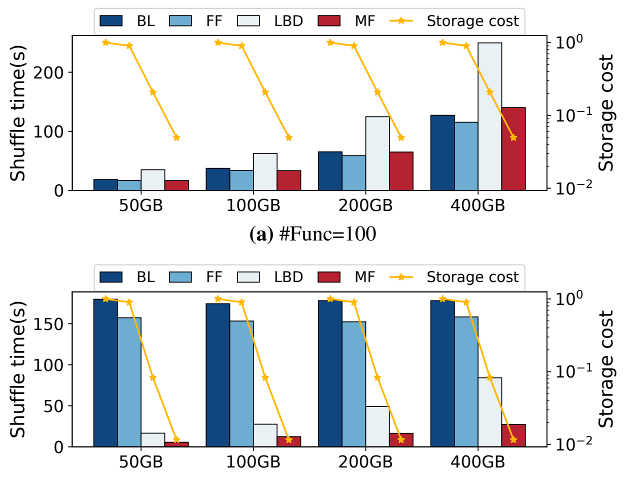


Figure 15: Scalability.

4.4 Breakdown and Overhead

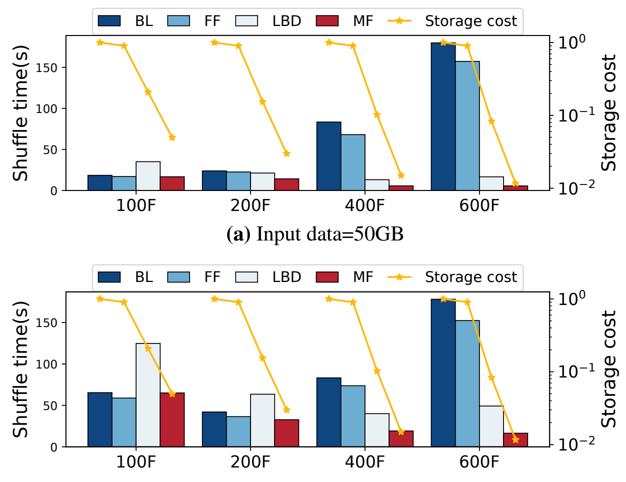
Performance Breakdown. We progressively integrate the three components to show their respective contribution to MinFlow's shuffle time reduction. Figure 14 shows the results of TeraSort under 600 function and 200GB input data, where the MinFlow with only Topology Optimizer is referred to as $MF^{\frac{1}{3}}$, the version with both Topology Optimizer and Function Scheduler as $MF^{\frac{2}{3}}$, and the full version MinFlow denoted as MF. As it suggests, $MF^{\frac{1}{3}}$ decreases the shuffle time by 43.93% and 34.46% compared with BL and FF but is slower than LBD. This is because compared to LBD which offers a two-level shuffle, by default Topology Optimizer of MinFlow chooses the highest *clevel* number it can generate, to decrease entailed PUTs/GETs maximally. Yet this often incurs too much additional intermediate data. Fortunately, after the Function Scheduler is combined such issue gets greatly alleviated, thus $MF^{\frac{2}{3}}$ performs better than LBD, by 19% in the figure. Last, the full version of MinFlow further integrates Configuration Modeler to judiciously select the optimal *clevel* number and corresponding suitable function scheduling plan, instead of just gluing the Topology Optimizer and Function Scheduler. As a result, the full version of MinFlow could out-



(a) #Func=100

(b) #Func=600

Figure 16: Various Input Data Size.



(a) Input data=50GB

(b) Input data=200GB

Figure 17: Tunable Function Parallelism.

perform other approaches by 66.62%-90.77%.

Besides, we further compare with MF-OPT, whose configuration is obtained by iteratively running all configurations and picking the one with the shortest shuffle time. As the purple line in Figure 14 shows, MinFlow achieves basically the same shuffle speed with MF-OPT once we ignore the tiny difference (below 1%) caused by our testbed's performance fluctuation.

System Overhead. Now we evaluate MinFlow's system overhead. First, the Topology Optimizer consumes additional CPU cycles to generate the candidate topologies before running jobs (see §3.2). Figure 15 presents the time cost, when MinFlow uses a single thread to perform topology calculation. As we can see, it basically increases linearly as the parallelism, i.e., the function number goes up. Under 600-function parallelism, the time is not above 1s, which can be ignored compared to MinFlow's improvement on shuffle time. Second, the Function Scheduler spends time searching in each of the candidate topologies for biparties, which are the basic function scheduling units (see §3.3). This part of the time cost is close to the topology calculation time as Figure 15 shows. As to some spikes in the figure, they appear when the parallelism value corresponds to more candidate topologies, i.e., the value that can be decomposed into more prime factors. For example, under $2 \times 2 \times 3 \times 5 \times 7 = 420$-function setting, it has 5 candidate topologies. In short, both of the above time costs are dwarfed by MinFlow's benefits. Moreover, if needed the time cost can be easily slashed by using multi-threads. Besides, though multi-level shuffle entails more functions, MinFlow eliminates the cost by keeping warm and reusing functions across levels. The memory consumed by local storage is also the reclaimed memory as in [25].

4.5 Impact of Different Configurations

As mentioned earlier, two factors impact MinFlow's performance, including the input size and function parallelism. Now we investigate the impact more extensively, by comparing MinFlow to other approaches under a broader range of input size and parallelism settings. For space limit, we only put results of TeraSort, while TPC-DS and WordCount show similar trends.

Input Size. First, we fix the parallelism as 100-function and tune the input size from 50GB to 100GB, 200GB, and 400GB. As we can see in Figure 16(a), across all approaches the shuffle time increases proportionally with the input size. The trend can be easily explained – Due to the number of PUTs/GETs to S3 not greater than $100\times100\times2$ = 20000, S3's speed of thousands of requests per second is enough to rapidly process them. Therefore for all approaches, the shuffle time is mainly determined by the volume of data to be transmitted, which is proportional to the input size. Note that even under such a low-parallelism setting, which is not MinFlow's target scenario, MinFlow could achieve near-optimal performance compared to others. By contrast, in the 600-function group (see Figure 16(b)), though LBD and MinFlow still exhibit a similar trend, the shuffle time of BL and FF remains consistent across all input sizes. Such difference stems from their distinct bottleneck. Specifically, for BL and FF the 600-function parallelism setting would incur $600\times600\times2$=720000 PUTs/GETs, making S3's speed the main bottleneck. As a result, their shuffle time is insensitive to the changing input size. However, due to LBD and MinFlow's great effectiveness in reducing the number of PUTs/GETs, their bottleneck still lies in functions' aggregated bandwidth sending/receiving intermediate data, leading to the shuffle time proportional to the input size. Note though it seems that under high-parallelism, say 600-function, the performance advantage of MinFlow over BL and FF shrinks as the input size increases, such trend would stop at a certain point where the input size is large enough to replace the massive PUTs/GETs as the new bottleneck.

Tunable Parallelism. Figure 17(a) shows the shuffle time results under parallelism of 100, 200, 400, and 600 functions, with 50GB input size. As we can see, for the low efficiency of performing shuffle, i.e., the huge number of PUTs/GETs

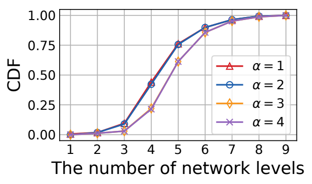


Figure 18: CDF of Network Levels for Prime within 1000.

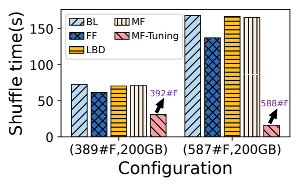


Figure 19: Shuffle Time under Prime Parallelism Cases.

to S3, both BL and FF's shuffle time keeps getting worse with the parallelism increasing, greatly impeding the critical scalability advantage of the serverless paradigm. By contrast, LBD exhibits a distinct trend that the shuffle time rapidly decreases as the parallelism gets higher, though its multiplied intermediate data, which must be transferred via remote S3, severely degrades its shuffle speed. For example, under low parallelism, say 100-function, it performs even worse than BL. In comparison, MinFlow not only preserves the continuously decreasing shuffle time but also avoids such degradation.

When the parallelism N is a prime number, we select the substitute in $[N-\alpha, N+\alpha]$ which can generate a network with as many *clevels* as possible. Figure 18 shows that after adjusting the prime numbers within 1000, the cumulative distribution of the number of network *clevels* can be generated. We can see that when $\alpha = 1$, all prime numbers except 2 can generate networks with more than two *clevels* and when $\alpha = 3$, more than 97% of prime numbers can generate networks with more than four *clevels*. Therefore, in cases involving prime parallelism, as illustrated in Figure 19, MinFlow after fine-tuning continues to outperform other approaches significantly.

Summary. MinFlow significantly outperforms other approaches in terms of both shuffle time and storage cost under high-parallelism, its target scenario. And even in a low-parallelism setting, it preserves good performance close to its best competitor, while considerably saving the storage cost.

5 Related Work

Optimization of Serverless DAGs. Several recent proposals have aimed to decrease job completion time by optimizing the performance of serverless DAGs. Orion [28] first proposes the idea of bundling multiple parallel invocations to mitigate execution skew and finds the best bundle size through trial and error. WiseFuse [29] goes a step further on Orion, it builds the performance model to determine bundle size and proposes the fusion of successive functions to reduce communication latency between consecutive stages in the DAG. However, given the huge data volume and dense topology inherent in serverless data analytics, fusion and bundling both struggle to mitigate the data exchange overhead. A complementary line of work provides efficient scheduling for serverless DAGs. Wukong [13] and FaaSFlow [25] provide decentralized and parallel scheduling distributed across function workers. Additionally, they harness over-provisioned local memory in the workers to expedite data exchange among functions within the same worker. This results in serverless DAGs that utilize both network I/O and local memory highly efficiently. Nevertheless, this approach proves inadequate when applied to serverless data analytics, as elaborated in §2.4. Overall, no prior work in this category can effectively reduce the data movement overhead of serverless data analytics.

Optimization of Serverless Intermediate Data Store. Besides DAGs optimization, recent work also reduces job completion time by optimizing the intermediate data store. Pocket [22] and Locus [35] show that current options for remote storage are either slow disk-based (e.g., S3) or expensive memory-based (e.g., ElastiCache). Thus, to balance performance and cost, Pocket combines different storage media (e.g., DRAM, NVMe, HDD) that users can choose to conform to their application needs. But this approach only makes economic sense when running different applications, e.g., when exclusively executing tasks like TeraSort, Pocket consistently selects the costly NVMe storage as the intermediate data repository. Faasm [37] and Cloudburst [38] accelerate data movement between functions, through a distributed shared memory across worker nodes. They rely on specific assumptions regarding the sandbox runtime and the programming interface exposed to tenants for developing their applications and in terms of consistency semantics and protocols between the FaaS workers and the backend storage. In contrast to existing efforts, MinFlow uses only cheap S3 and reclaimed memory, achieving performance and economic gains.

6 Conclusion

In this paper, we develop MinFlow, a holistic data passing framework for I/O-intensive serverless analytics jobs. MinFlow efficiently creates multi-level data passing topologies with fewer PUT/GET operations and uses an interleaved strategy to partition the topology DAG into complete bipartite sub-graphs. This optimizes function scheduling and cuts data transmission to remote storage by over one half. Additionally, MinFlow employs a precise model to pinpoint the best configuration. Experiments on our prototype demonstrate that MinFlow significantly outperforms state-of-the-art systems in both the job completion time and storage cost.

Acknowledgments

We thank our shepherd, Eno Thereska, and the anonymous reviewers for their comments. This work was supported in part by NSFC (62172382). The work of Yongkun Li was supported in part by the Youth Innovation Promotion Association CAS and Zhai Guanglong Scholarship. The work of John C.S. Lui was supported in part by the RGC GRF 14202923. Yongkun Li is USTC Tang Scholar.

A APPENDIX

A.1 Topology Space Size

Theorem A.1. *For a symmetric single-level shuffle network with function parallelism N, the topology space size of its multi-level shuffle network is* $SS = \sum_{j=1}^{p} \sum_{i=1}^{j} \frac{(-1)^{j-i} i^p}{i!(j-i)!}$, *where p refers to the number of prime factors of N.*

According to §3.2, the topology space size of the multi-level shuffle for the aforementioned network is equivalent to the search space size encompassing all factorizations of N. An intuitive way to explore all factorizations of N is to combine its prime factors, e.g., the 2-factorization of N is equivalent to dividing the prime factors of N into two nonempty sets. Next, we prove that this method results in the search space equal to SS in Theorem A.1.

Proof. Assume that $N = n_1 \times n_2 \times ... \times n_p$, where n_i is a prime, $1 \le i \le p$ and let $S(n,k)$ denote the number of $k-factorization$s of an integer with n prime factors. Then, $SS = \sum_{j=1}^{p} S(p,j)$. Note that we can derive all factorizations in $S(n,k)$ from factorizations in $S(n-1,k)$ and $S(n-1,k-1)$ through the following two methods:

- **case1:** Assume the extra factor of $S(n,k)$ compared to $S(n-1,k)$ as m. Combine m with any factor of a factorization in $S(n-1,k)$.
- **case2:** Assume the extra factor of $S(n,k)$ compared to $S(n-1,k-1)$ as m. Let m become a new factor of a factorization in $S(n-1,k-1)$.

Therefore, we can conclude that $S(n,k) = k \times S(n-1,k) + S(n-1,k-1)$ and $S(n,k)$ is the *Stirling Number of the Second Kind*, whose general formula is $\sum_{i=1}^{k} \frac{(-1)^{k-i} i^n}{i!(k-i)!}$ [42]. Furthermore, we can deduce that $SS = \sum_{j=1}^{p} S(p,j) = \sum_{j=1}^{p} \sum_{i=1}^{j} \frac{(-1)^{j-i} i^p}{i!(j-i)!}$. ☐

A.2 CBGs in Multi-level Networks

Theorem A.2. *For a multi-level topology generated by the progressively converging method (the parallelism is N), the i-th clevel of links corresponding to factor d_i, along with two adjacent flevels, can be divided into $\frac{N}{d_i}$ disjoint complete bipartite graphs with width d_i.*

Proof. Define the mapping function $\mathcal{F} := \langle f_1, f_2 \rangle := \langle x, x \rangle \mapsto \langle \lfloor x/s_{i+1} \rfloor, x\%s_i \rangle, x \in \{0,1,...,N-1\}$. According to Equation (2), $F_{i,m}$ and $F_{i,n}$, where $\mathcal{F}(m) = \mathcal{F}(n)$, have the same receiver functions R.

We categorize the functions at *flevel* i into conjugacy classes, with the elements within each class sharing the same R. Assume that the ranges of $\mathcal{F}, f_1$ *and* f_2 are $\mathcal{R}, r_1$ *and* r_2 respectively, then $|\mathcal{R}| = |r_1| \times |r_2| = \frac{N}{s_{i+1}} \times s_i = \frac{N}{d_i}$, in other words, the functions at *flevel* i can be partitioned into $\frac{N}{d_i}$ conjugacy classes. For any element $\langle m,n \rangle$ in $\mathcal{R}$, we can find it's $\frac{s_{i+1}}{s_i} = d_i$ preimages, namely, $m \times s_{i+1} + k \times s_i + n, k \in \{0,1,...,d_i-1\}$. In summary, the functions at *flevel* i can be divided into $\frac{N}{d_i}$ conjugate classes of size d_i.

Because each conjugacy class at *flevel* i and its R at *flevel* $i+1$ together constitute a complete bipartite graph, the i-th *clevel* of links corresponding to factor d_i, along with two adjacent levels of functions, can be divided into $\frac{N}{d_i}$ disjoint complete bipartite graphs with width d_i. ☐

A.3 Applicability of Mesh-based Networks

Constructing k-level shuffle networks for function parallelism in N is equivalent to grouping N functions k times and performing intra-group shuffle after each grouping, which is revealed by Theorem A.2. Note that this process necessitates distinct groupings at each *flevel*.

Mesh-based method [32,34] groups functions by projecting N functions into a k-dimensional mesh to construct a k-level shuffle network. Specifically, they adopt the unary mapping $h_1(x,c) := x \mapsto \lfloor x/c \rfloor$ and $h_2(x,c) := x \mapsto x\%c$ for grouping, where $c|N, x \in \{0,1,...N-1\}$. However, this approach does not give guidance on selecting the side length (i.e., group size) of the k-dimensional mesh, while we conduct a detailed theoretical analysis in §3.2. Even worse, as shown in Theorem A.3, despite having insights into selecting an appropriate group size, the grouping methods outlined in [32, 34] still are proved to be ineffective in many cases.

Theorem A.3. *For multi-level networks where the number of flevels with the same group size is greater than three, unable to generate distinct groupings for flevels with the same group size using the unary mapping in* $h_1(x,c) := x \mapsto \lfloor x/c \rfloor$ *and* $h_2(x,c) := x \mapsto x\%c$, *where* $c|N, x \in \{0,1,...N-1\}$.

Proof. Assume that $N = s^n, n \ge 3$. According to §3.2, performing progressively converging for the N functions, we can construct a multi-level network where the number of *flevels* with group size s is greater than three. However, we can not use $h_1(x,c)$ and $h_2(x,c)$ to achieve this.

For $h_1(x,s)$, we can only perform $h_1(x,s) := x \mapsto \lfloor x/s \rfloor$ to divide N functions into groups with group size s. This is because that $h_1(x,s^m), 2 \le m \le n$ divides N functions into groups with group size greater s.

Likewise, as to $h_2(x,c)$, we can only perform a grouping $h_2(x,s^{n-1}) := x \mapsto x\%s^{n-1}$ distinct with $h_1(x,s)$ to divide N functions into groups with group size s. ☐

Corollary A.3. *For multi-level networks where the number of flevels with the same group size is greater than three, unable to generate distinct groupings for flevels with the same group size using arbitrary nesting of the unary mapping in* $\{h_1(x,c)|\ c|N\} \bigcup \{h_2(x,c)|\ c|N\}$, *where* $x \in \{0,1,...N-1\}$.

Proof. It is omitted as it is similar to the proof of Theorem A.3. ☐

Note that our method uses binary mapping $\mathcal{F}$ in Theorem A.2 instead of unary mapping to solve the limitations of mesh-based methods.

B Artifact Appendix

Abstract

Our artifact includes the prototype implementation of MinFlow and three other state-of-the-art comparison methods, along with the three data analytics benchmarks evaluated in our experiments. Additionally, we provide experiment scripts for reproducing our results on Amazon EC2 instances.

It's important to note that reproducing all of our results will take tens of hours and thousands of dollars with the Amazon cloud service.

Scope

The artifact has two main goals: The first is to enable the validation of the main claims presented in the paper. The second is to facilitate others in building upon MinFlow for their own projects. We include code to reproduce Figures 7-19.

Contents

The artifact is hosted in a git repository. This repository includes MinFlow 's source code as well as documentation and example applications. It is structured as follows:

benchmark/: This folder contains the code for the three evaluation applications (Terasort, TPC-DS, and WordCount) and input data generators for each application. By running `create_image.bash` in each application directory, serverless job-specific images can be deployed on the worker node.

config/: This folder contains the configuration file `config.py`, allowing users to configure the database and node information. It also provides options to select the application and comparison method for evaluation.

scripts/: This folder contains scripts (`conda_install.bash`, `python_install.bash`, and `docker_install.bash`) to automatically install the software dependencies of MinFlow, including Anaconda, Python, and Docker.

src/: This folder contains the source code of MinFlow and three other comparison methods (Baseline, FaaSFlow, and Lambada). We have integrated them and users can switch between different systems using the configuration file. The source code is structured as follows:

- **base/ & container/:** These two folders contain the code that builds the base images which expose hybrid-store APIs used in Function Scheduler (§3.3) and Configuration Modeler (§3.4) for applications.
- **parser/:** This folder contains the code that parses the application's YAML configuration file, written in the Workflow Definition Language, into a DAG object used in Topology Optimizer (§3.2).
- **grouping/:** This folder contains the code for Topology Optimizer (§3.2), Function Scheduler (§3.3), and Configuration Modeler (§3.4) to find the optimal execution plan.
- **workflow_manager/:** This folder contains the code for workflow management, including monitoring function status and triggering functions.
- **function_manager/:** This folder contains the code for managing containers (including creating, keeping warm, and removing) and executing functions.

test/: This folder contains the code for reproducing most of our evaluation results in Figures 7-9,11-13,16-17 (see folders **fast/** and **cost/**), 10 (see folder **load_balance/**), 14 (see folder **breakdown/**), 15 (see folder **scalability/**), 18 (see folder **alpha/**) and 19 (see folder **prime/**).

README.md: This documentation details how to install the software, set up the system, and reproduce the results in our paper.

Hosting

MinFlow artifact repository is hosted on GitHub and archived using Zenodo with a permanent DOI.

- **Repository:** https://github.com/lt2000/MinFlow.
- **Zenodo Archive:** https://zenodo.org/records/10494631.
- **DOI:** https://zenodo.org/doi/10.5281/zenodo.10494631.

Requirements

The artifacts have been developed and tested on an Amazon EC2 cluster comprising 10 m6i.24xlarge instances. And, the artifact uses Docker containers to host serverless functions, orchestrate the functions, and organize them in a DAG. §4.1 details the exact environment we used in our experiments.

Environment Setup

1. First, install the software dependencies by running the scripts directory in **scripts/** and mount the Tmpfs as our local storage.
2. Then, generate input data and build base and job-specific images for evaluation applications.
3. Configure the system configuration files and use `src/grouping/metadata.py` to generate the optimal execution plan.
4. Run the system and reproduce the results following the detailed instructions in `README.md`.

References

[1] Linux tmpfs, 2023. https://www.kernel.org/doc/html/latest/filesystems/tmpfs.html, Last accessed on 2023-6-29.

[2] Sort Benchmarks, 2023. https://sortbenchmark.org/, Last accessed on 2023-6-29.

[3] TPC-DS, 2023. https://www.tpc.org/tpcds/#, Last accessed on 2023-6-29.

[4] AMAZON. Amazon elasticache, 2023. https://aws.amazon.com/elasticache/, Last accessed on 2023-6-29.

[5] AMAZON. Amazon Lambda, 2023. https://aws.amazon.com/lambda, Last accessed on 2023-6-11.

[6] AMAZON. Amazon Lambda price, 2023. https://aws.amazon.com/cn/lambda/pricing/.

[7] AMAZON. Amazon Lambda usecases, 2023. https://docs.aws.amazon.com/lambda/latest/dg/applications-usecases.html, Last accessed on 2023-7-2.

[8] AMAZON. Amazon S3, 2023. https://aws.amazon.com/s3/, Last accessed on 2023-6-11.

[9] AMAZON. Amazon S3 price, 2023. https://aws.amazon.com/cn/s3/pricing/.

[10] AMAZON. Amazon S3 QPS limit, 2023. https://aws.amazon.com/cn/about-aws/whats-new/2018/07/amazon-s3-announces-increased-request-rate-performance/, Last accessed on 2023-6-29.

[11] Apache. Apache Kvrocks, 2023. https://github.com/apache/kvrocks, Last accessed on 2023-7-2.

[12] J. Carreira, P. Fonseca, A. Tumanov, A. Zhang, and R. Katz. Cirrus: A serverless framework for end-to-end ml workflows. In *Proceedings of the ACM Symposium on Cloud Computing*, pages 13–24, 2019.

[13] B. Carver, J. Zhang, A. Wang, A. Anwar, P. Wu, and Y. Cheng. Wukong: A scalable and locality-enhanced framework for serverless parallel computing. In *Proceedings of the 11th ACM Symposium on Cloud Computing*, pages 1–15, 2020.

[14] M. Chowdhury and I. Stoica. Coflow: A networking abstraction for cluster applications. In *Proceedings of the 11th ACM Workshop on Hot Topics in Networks*, pages 31–36, 2012.

[15] J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[16] J. Duato, S. Yalamanchili, and L. Ni. *Interconnection networks*. Morgan Kaufmann, 2003.

[17] J. Herrmann, M. Y. Özkaya, B. Uçar, K. Kaya, and U. V. Çatalyürek. Multilevel algorithms for acyclic partitioning of directed acyclic graphs. *SIAM J. Sci. Comput.*, 41(4):A2117–A2145, jan 2019.

[18] IBM. IBM cloud functions usecases, 2023. https://cloud.ibm.com/docs/openwhisk?topic=openwhisk-use_cases, Last accessed on 2023-7-2.

[19] E. Jonas, J. Schleier-Smith, V. Sreekanti, C.-C. Tsai, A. Khandelwal, Q. Pu, V. Shankar, J. Carreira, K. Krauth, N. Yadwadkar, et al. Cloud programming simplified: A berkeley view on serverless computing. *arXiv preprint arXiv:1902.03383*, 2019.

[20] A. Khandelwal, Y. Tang, R. Agarwal, A. Akella, and I. Stoica. Jiffy: Elastic far-memory for stateful serverless analytics. In *Proceedings of the Seventeenth European Conference on Computer Systems*, pages 697–713, 2022.

[21] A. Klimovic, Y. Wang, C. Kozyrakis, P. Stuedi, J. Pfefferle, and A. Trivedi. Understanding ephemeral storage for serverless analytics. In *2018 USENIX annual technical conference (USENIX ATC 18)*, pages 789–794, 2018.

[22] A. Klimovic, Y. Wang, P. Stuedi, A. Trivedi, J. Pfefferle, and C. Kozyrakis. Pocket: Elastic ephemeral storage for serverless analytics. In *OSDI*, pages 427–444, 2018.

[23] V. Kumar, A. Grama, A. Gupta, and G. Karypis. *Introduction to parallel computing*, volume 110. Benjamin/Cummings Redwood City, CA, 1994.

[24] H. R. Lewis. Michael r. πgarey and david s. johnson. computers and intractability. a guide to the theory of np-completeness. wh freeman and company, san francisco1979, x+ 338 pp. *The Journal of Symbolic Logic*, 48(2):498–500, 1983.

[25] Z. Li, Y. Liu, L. Guo, Q. Chen, J. Cheng, W. Zheng, and M. Guo. Faasflow: Enable efficient workflow execution for function-as-a-service. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 782–796, 2022.

[26] Y. Liu, B. Jiang, T. Guo, Z. Huang, W. Ma, X. Wang, and C. Zhou. Funcpipe: A pipelined serverless framework for fast and cost-efficient training of deep learning models. *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, 6(3):1–30, 2022.

[27] A. Mahgoub, K. Shankar, S. Mitra, A. Klimovic, S. Chaterji, and S. Bagchi. Sonic: Application-aware data passing for chained serverless applications. In *USENIX Annual Technical Conference (USENIX ATC)*, 2021.

[28] 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 *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 303–320, 2022.

[29] A. Mahgoub, E. B. Yi, K. Shankar, E. Minocha, S. Elnikety, S. Bagchi, and S. Chaterji. Wisefuse: Workload characterization and dag transformation for serverless workflows. *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, 6(2):1–28, 2022.

[30] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: a system for large-scale graph processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, pages 135–146, 2010.

[31] Microsoft. Azure Functions usecases, 2023. https://learn.microsoft.com/en-us/dotnet/architecture/serverless/serverless-business-scenarios, Last accessed on 2023-7-2.

[32] I. Müller, R. Marroquín, and G. 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.

[33] S. M. Nabavinejad, M. Goudarzi, and S. Mozaffari. The memory challenge in reduce phase of mapreduce applications. *IEEE Transactions on Big Data*, 2(4):380–386, 2016.

[34] M. Perron, R. Castro Fernandez, D. DeWitt, and S. Madden. Starling: A scalable query engine on cloud functions. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, pages 131–141, 2020.

[35] Q. Pu, S. Venkataraman, and I. Stoica. Shuffling, fast and slow: Scalable analytics on serverless infrastructure. In *NSDI*, volume 19, pages 193–206, 2019.

[36] Purdue. Purdue mapreduce benchmarks suite, 2023. https://engineering.purdue.edu/~puma/datasets.htm, Last accessed on 2023-6-29.

[37] S. Shillaker and P. Pietzuch. Faasm: Lightweight isolation for efficient stateful serverless computing. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pages 419–433, 2020.

[38] V. Sreekanti, C. Wu, X. C. Lin, J. Schleier-Smith, J. M. Faleiro, J. E. Gonzalez, J. M. Hellerstein, and A. Tumanov. Cloudburst: Stateful functions-as-a-service. *arXiv preprint arXiv:2001.04592*, 2020.

[39] S. Thomas, L. Ao, G. M. Voelker, and G. Porter. Particle: ephemeral endpoints for serverless networking. In *Proceedings of the 11th ACM Symposium on Cloud Computing*, pages 16–29, 2020.

[40] L. G. Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103–111, 1990.

[41] L. Wang, M. Li, Y. Zhang, T. Ristenpart, and M. Swift. Peeking behind the curtains of serverless platforms. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 133–146, 2018.

[42] Wolfram. Stirling Number of the Second Kind, 2023. https://mathworld.wolfram.com/StirlingNumberoftheSecondKind.html.

[43] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauly, M. J. Franklin, S. Shenker, and I. Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Presented as part of the 9th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 12)*, pages 15–28, 2012.

[44] M. Zaharia, R. S. Xin, P. Wendell, T. Das, M. Armbrust, A. Dave, X. Meng, J. Rosen, S. Venkataraman, M. J. Franklin, et al. Apache spark: a unified engine for big data processing. *Communications of the ACM*, 59(11):56–65, 2016.

[45] H. Zhang, B. Cho, E. Seyfe, A. Ching, and M. J. Freedman. Riffle: Optimized shuffle service for large-scale data analytics. In *Proceedings of the Thirteenth EuroSys Conference*, pages 1–15, 2018.

[46] H. Zhang, Y. Tang, A. Khandelwal, J. Chen, and I. Stoica. Caerus: Nimble task scheduling for serverless analytics. In *NSDI*, pages 653–669, 2021.

[47] J. Zhang, A. Wang, X. Ma, B. Carver, N. J. Newman, A. Anwar, L. Rupprecht, V. Tarasov, D. Skourtis, F. Yan, and Y. Cheng. Infinistore: Elastic serverless cloud storage. *Proc. VLDB Endow.*, 16(7):1629–1642, may 2023.