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

Submission #152

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 less 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 the 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 of 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 compute power. Platforms such as AWS Lambda [5] and Azure Functions [28] 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 [18, 19, 32]. Embedded in this analytical landscape are frameworks like Google's MapReduce [12] and Apache Spark [40].

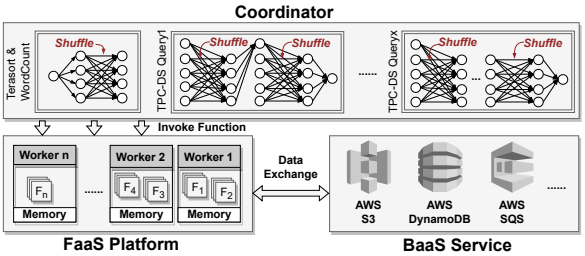


Figure 1: Serverless computing architecture. *Circles represent sub-tasks, and each link between circles represent one data transfer, consisting one PUT plus one GET*

In data analytics, the "shuffle" operation is pivotal for data passing between stages. Notably, over half of Facebook's daily analytics entail at least one shuffle operation. Given the stateless nature of serverless, data are largely passed through remote Object Stores like S3 [6], 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 $50 \times 50 = 250{,}000$ PUTs and an equal number of GETs, totally 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 tasks [19]. 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 approach, resulting in suboptimal performance. Foremost, the approach of using private storage has been utilized [19, 32], wherein shuffle is conducted through self-maintained storage such as ElastiCache clusters [3]. 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 [10, 22], harnessing over-provisioned local memory in workers for faster shuffle operations. However, its applicability remains tethered to functions situated within the same worker. Yet, 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 [29, 31], inspired by the mesh networks from HPC [20], endeavors to streamline shuffle operations. Yet, it incurs multiplied data to be transmitted, making the bandwidth limit in the function-side a new bottleneck, especially when the size of 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 merging 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 segments 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 works.
- 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 PUT/GETs and also the storage cost.

We implement a prototype of 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 of the shuffling performance and storage cost. For example, in high-parallelism case of 600 mapper and 600 reducer functions for Terasort, MinFlow reduces the shuffle time by 66.62% - 89.22%, compared to Lambada [29] and FaaSFlow [22], respectively, and it also reduces the storage cost by 84% - 98.57%, respectively.

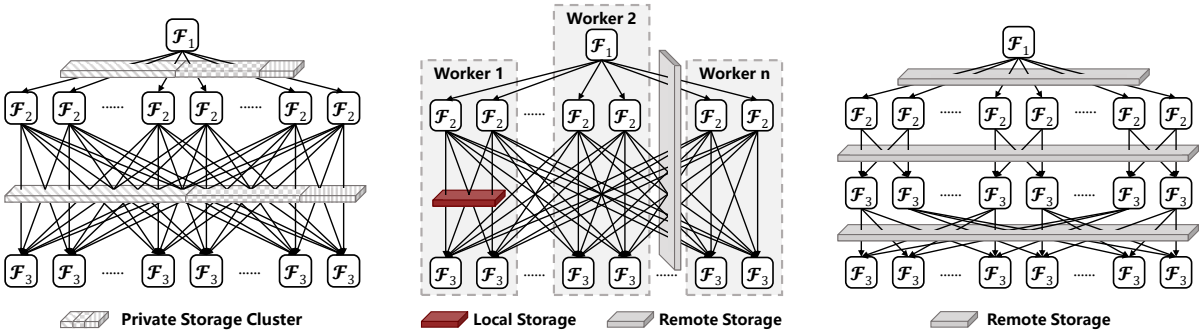
We will release the source codes in the final paper.

2 Background and Motivation

2.1 Background

Serverless Computing Architecture. As the building blocks of Serverless, FaaS and BaaS (e.g., Amazon lambda [5] and S3 object store [6]) 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 to first 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, along with the containerized functions, greatly facilitates scaling up/down compute resources as needed (e.g., to tackle bursty workloads). Second, it provides 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 100 milliseconds during function execution. Due to all its virtues, the increasing number of applications starts to embrace the architecture, including Web, IoT, data analytics, and etc. [4, 15, 28].

Data Analytics atop Serverless. Data analytics aims at efficiently processing huge amounts of data as specified so as to obtain desired results, and it has been employed in a wide range of domains, including scientific computing, machine learning, large-scale graph computations, and etc. [36]. To offer essential scalability and fault-tolerance, mainstream data analytics frameworks [12,27,39] commonly adopt bulk-synchronous-parallel model (BSP) [37], which divides a whole job into consecutive stages, each stage composed of parallel sub-tasks. When each stage completes, the intermediate results are transferred to the next stage, via communication primitives such as *shuffle* and *broadcast* [11, 16], for further computation. Thereby, the workflow of tasks employing BSP can be represented as DAGs, as Figure 1 illustrates. To deploy data analytics jobs atop Serverless platforms, users typically first declare the task's workflow to a coordinator via configuration files. Then the coordinator takes over the control, activating functions to perform consecutive stages in turn, sub-tasks within each stage executed by parallel functions. And users get notified when the whole job completes. Notably, since serverless functions are network unaddressable, data transmission between stages is realized via remote back-end storage, typically S3, rather than direct P2P data passing. Benefiting



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

Figure 2: Existing Approaches

from the elasticity and fine-grained billing model of FaaS, the job's computation could be easily accelerated by splitting each stage into more sub-tasks allocated to parallel functions, with far less cost than traditional physical/virtual machines. Thereby, a lot of research works have focused on running data analytics based on serverless [10, 16, 17, 19, 22, 24, 29, 41].

2.2 Dilemma Caused by *Shuffle*

In data analytics, *shuffle* is the most common primitive that transmits data between adjacent stages. Prior research shows that over 50% of daily data analytics jobs at Facebook involve at least one *shuffle* operation [41]. As Figure 1 shows, when performing *shuffle*, each sub-task in the previous stage distributes its output to all sub-tasks in the next stage. And such all-to-all transmission manner would greatly "break down" the intermediate results, causing proliferating requests: suppose stages' parallelism is N, then at least $2N^2$ object PUT/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 problem 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 [8]), the quadratic $2N^2$ PUT/GETs could easily get throttled, especially when N is large. Consequently, while computation time could be slashed by improving the parallelism N, the whole data analytics progress is significantly slowed down by *shuffle*. For example, in Pocket, over 62% of time is spent shuffling data, while computation only takes 16% of time for 100GB Terasort [19]. Second, it drastically inflates the cost. Albeit S3 offers cheap capacity (0.023 USD$ per GiB/-month), it charges expensive in increments of single request (0.005/0.0004 USD$ per 1k PUT/GET) [7]. As a result, the $2N^2$ PUT/GETs would rapidly increase the cost as N goes up. In conclusion, both elasticity and economy of Serverless get severely impeded by *shuffle*.

2.3 Existing Approaches

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

Shuffle via Private Storage. As a public cloud storage service shared by massive users and applications, S3 inherently assigns limited request rate to each single user, to guarantee fairness and avoid interference among tenants. A straightforward way to eliminate the restriction is to replace S3 with self-maintained private storage, for example, ElastiCache clusters [3]. This provides the user an exclusive ownership of the storage service, thus greatly improves *Shuffle* speed. However, for losing S3's sharing economy and fine-grained billing model, it often brings about significantly increasing cost. As Pocket [19] suggests, the cost is 100 times higher than S3 for sort jobs. Though some remedies have been proposed to mitigate the surging cost, e.g., Pocket [19] and Locus [32] dynamically rightsizing resources and combine high-end and cheap storage media to achieve better trade-offs between performance and cost, maintaining private storage is still intolerably expensive, 42 times higher than S3 for sort jobs.

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* [10, 22]. More specifically, data passing between functions located in the same worker is performed via its local memory. For example, function F_2 and F_3 in Figure 2(b) are co-located in worker W_1. Suppose data to be transferred 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 has much higher bandwidth and lower latency, but also does not incur extra overhead. The downside is the applicability constraint that only co-located functions can adopt this manner [22].

Multi-level Shuffle. Borrowing idea from HPC (High Performance Computing) field, which achieves all-to-all connection among processors through the k-dimensional Mesh Network [13], Starling [31] and Lambada [29] project *Shuffle*-involved functions onto a k-dimensional mesh with side length $\sqrt[k]{P}$ where P 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 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 $2P^2$ to $2kP\sqrt[k]{P}$. 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 less requests are needed to be transmitted through remote S3 during *Shuffle*, performance degradation caused by S3' throttling gets mitigated, and less fee is charged as well.

2.4 Limitations

The aforementioned approaches face respective limitations in cost, performance, or applicability. First, private storage for faster *Shuffle* entails high surging cost [19], even considering those remedies compromising performance for better cost-efficiency [19, 32]. Besides, to maintain the private storage, users have to bear additional management works like resource scaling, fault tolerance, etc., which should have been undertaken by Serverless, thus violating the easy-to-use principle.

Second, for performing *Shuffle* via intra-worker memory, 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 has to resort to remote storage for data transmission. Worse yet, the benefits of memory-assisted transmission could be easily offset by the straggler caused by slower remote storage.

Last, regarding *ML-Shuffle*, existing methods based on k-dimensional mesh suffer applicability problem. i.e., it mandates a symmetrical Mapper-Reducer setting, which means the number Mappers and Reducers must be the same (e.g., both are P). Besides, while allowing adjusting 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 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 [9, 19, 29], such heavy traffic could exacerbate the problem. On the contrary, smaller k often comes with unsatisfactory effect on reducing the number of requests. Moreover, the 2-level shuffle algorithm can not be easily applied to more levels, and the extension from 2-level shuffle algorithm to a general k-level one is non-trivial.

Inefficiency Analysis. Though a series of optimizing "actions" that can be employed, for lack of a systematic understanding, there isn't a judicious "decision maker" that can use them collaboratively. Consequently, despite 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 part 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 HPC field, whose applicability is strictly limited in Serverless scenario. Last, rather than carefully considering specific analytics jobs' characteristics 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.

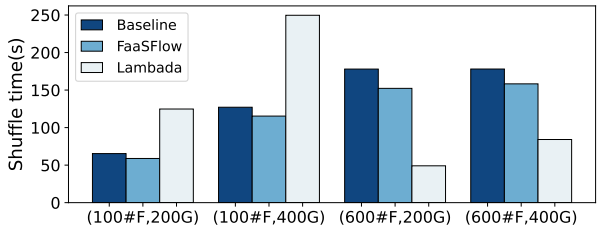


Figure 3: Terasort Shuffle Time under Different Configuration. *Baseline transfers all intermediate data via S3.*

2.5 Main Idea and Challenges

Main Idea. The key factors deciding analytics jobs' efficiency includes function topology represented by DAG, function scheduling, and transmission media selection. Compared to considering them separately, optimizing them in synergy greatly helps find the optimal configuration, so as to eradicate bottleneck from the whole workflow. For example, *ML-Shuffle* facilitates traffic localization via local memory, since the links in each level are more sparse and functions can be more easily co-located to avoid cross-worker data transmission. Also, traffic localization largely absorbs the extra traffic volume caused by *ML-Shuffle*. Therefore, for any given analytics job, our main idea is to first construct the whole configuration space by taking all those three dimensions into consideration, then derive the optimal configuration from the space, based on user requirements, the task's characteristics, and 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 jobs, including those with asymmetric Mappers and Reducers, despite conventional mesh-based method supposes $\#mapper = \#reducer = P$ and only provides concrete algorithm for 2-level shuffle.[1] Plus, for a specified analytics job, the complete *ML-Shuffle*

[1] $\#mapper$ and $\#reducer$ are the number of mapper and reducer, n is a positive integer greater than 1, i.e., $n \in \mathbb{N}^+ \backslash \{1\}$.

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 transmission, while simultaneously ensuring load-balance among workers and avoiding stragglers. This process is equivalent to searching for a partitioning scheme that divides the whole DAG into sub-graphs consisting of co-located functions in accordance with requirements, which is obviously an NP-hard problem and especially time-consuming when the number of function is large.
- **Modeling to find 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 dependent on specific platforms.

3 MinFlow Design

To optimize the data passing between functions, we propose MinFlow, an unified data passing framework for analytics jobs atop Serverless platforms, which seeks the globally optimal configuration to simultaneously achieve high-performance and low-cost. We first introduce the overall architecture (§3.1), and elaborate on each technique in details (§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 job upon receiving user-submitted task-running request, followed by the coordinators deploying and running 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. 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 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 provided candidate topology, function scheduler decides which functions should be co-located at the same worker and transfer data through

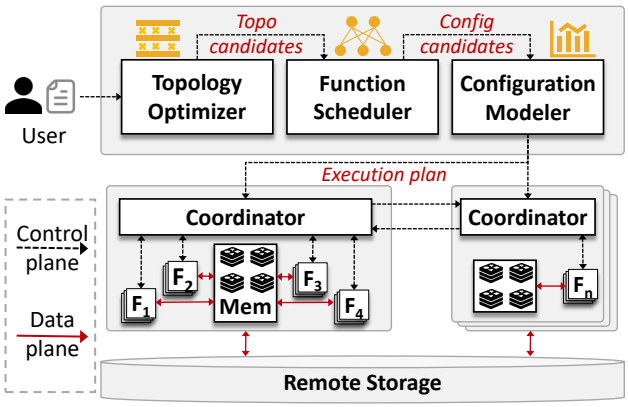


Figure 4: Overview of MinFlow Architecture.

local memory, by dividing the complete topology into sub-graphs. And the partitioning must simultaneously achieving load balance, cross-worker traffic minimization, and straggler avoidance. To address the NP-hard problem, MinFlow employs a heuristic algorithm to quickly find the nearly optimal solution.

- **Configuration Modeler.** Configuration modeler selects the optimal configuration, i.e., that with the shortest estimated completion time and lowest cost for data transmission, among candidates. At its core is a mathematical formula 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 obtain in runtime, it determines them by an efficient and lightweight sampling method.

Note though our current design largely follow FaaSFlow's distributed function coordination, which employs multiple coordinators to prevent function scheduling 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 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 mesh-based method does. Moreover, it's a general k-level shuffle algorithm that allows asymmetric number of mappers and reducers.

The key idea behind the Progressively converging ML-Shuffle is a "divide and conquer" strategy. Rather than projecting functions to a rigid k-dimensional grid, it first divides functions in the first level into groups of the same size (initially 1), and gradually lets them converge into larger groups in the next level, while preserving full connection between each group and its upstream mappers, until all functions in the last

level exist in the same group, thus ultimately achieving global all-to-all connection. For example, as Figure 5a illustrates, to build a three-level shuffle when $\#mapper = \#reducer = 8$, functions are respectively divided into 8, 4, 2, and 1 group for level 0, 1, 2, 3. And suppose we let $C_{i,j}/F_{i,j}$ denote the j-th group/function at level i, the data in $C_{0,0}$ in turn passes into $C_{2,0}$, $C_{3,0}$, and $C_{4,0}$. Analogously, the data in $C_{1,0}$ passes into $C_{2,1}$, $C_{3,0}$, and $C_{4,0}$. The rest are similar.

More generally, to derive an L-level topology comprising $L+1$ function levels, with each level having N functions, we divide the functions of i-th level into g_i groups, where $g_i(\ 0 \le i \le L)$ meets the following conditions:

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

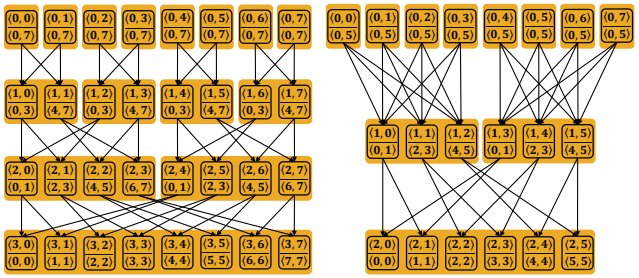
When converging groups, to preserve the full connection between the new group and its upstream mappers, for each function in the new group, a unique path between it and any upstream group in the previous level must be guaranteed. To achieve this, we set the receiver functions of $F_{i,j}$ as R:

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

For example, 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 $\{F_{2,2}, F_{2,3}\}$. As we see, the link distribution is also kept even to balance transmission load among functions. Moreover, to ensure the correctness of data passing, each function must carefully partition and distribute received data to the next levelfunctions. For function $F_{i,j}$, supposing $D_{i,j}$ denote its received data, it first partitions $D_{i,j}$ into $|R|$ continuous and equal-sized parts, then orderly assigns them to the receiver functions, i.e., functions in its R.

Notably, the flexibility of the Progressively converging method lies in the setting of $D = \{d_i | 0 \le i \le L\}$, since it determines G 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 level number thus have different preference for number of request, data transmission volume, etc. For example, the data transmission volume is obviously proportional to L, since each additional level incurs one more intermediate data transmission, and combining Equation 2 we have that the number of edge is $N \times \sum_{i=1}^{L} d_i$, by doubling which we can get the number of GET/PUTs. Actually the space covers those topologies generated by conventional mesh-based method. And it can be proven that supposing N can be decomposed as the product of p prime factors, the space size $SS = 1 + \sum_{k=2}^{p-1} [(1+\frac{k}{2})k^{p-k} - \frac{k}{2}] = \mathbf{O}(p * \frac{p}{2}^{\frac{p}{2}})$ (e.g. $SS = 808686$ when N is 4096). Such selectivity greatly facilitates seeking the most appropriate topology for an analytics job. Later we will detail how to select appropriate topology from the space, by carefully setting D.

Last, compared to mesh-based approach, the applicability gets significantly improved as well. As depicted in Figure 5b,



(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.*

Progressively converging Shuffle even works for asymmetric number of senders and receivers ($\#mapper = 8 \neq \#reducer = 6$), provided we keep the intermediate level the same size as the Reduce level, and link functions as Equation (2) suggests.

Candidates for Optimal Topology. For 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 going 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 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 fewest GET/PUTs that are more promptly processed by remote storage service.
- Case 2. Under different L, the comparision could be ambiguous, since on one hand larger L reduces edges, thus GET/PUTs, more significantly. On the other hand, it transmits the intermediate data L times, potentially throttled by 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 edge number is $N \times \sum_{i=1}^{L} d_i$. Then suppose N can be decomposed into p prime factors, which means feasible L lies in $[2, p]$, the candidate selecting can be transformed into a series of optimization problems as follows:

$$\text{For } L \in [2,p] \begin{cases} \text{minimize} & N \times \sum_{i=1}^{L} d_i \\ \text{subject to} & \prod_{i=1}^{L} d_i = N,\ d_i \in \mathbb{N}^+ \backslash \{1\} \end{cases} \quad (3)$$

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 $N \in [2,p]$. And 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 as 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 $L \in [2,p]$ 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 $MinSum(N,L)$, $2 \le L \le p$ gets solved. Moreover, we use $Sol(i,j)$ to track the decomposition path of $MinSum(i,j)$, i.e., the selected n of $MinSum(i,j)$ in Equation 4. Then by iteratively putting $Sol(i,j)$ along the decomposition path of $MinSum(N,L)$ into a sequence, we can get the desired $D = \{d_i | 0 \le i \le L\}$, by which we can easily derive the corresponding L-level topology with the fewest edges.

3.3 Function Scheduler

Function scheduler assigns a scheduling plan to each of the candidate topologies, indicating when and on which worker each functions should be invoked. While the "when" question is straightforward to deal with, by monitoring the completion time of 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 transmit 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 thus accelerate the data transmission.
2) *Transmission straggler avoidance.* Due to the barrier synchronizations of BSP model, the duration of each level's transmission is decided by the slowest edge. Thus edges in the same level 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 sub-graphs' width, i.e., the number of functions in the tier with the most functions, must be capped to ensure functions' computing and communication load can be easily spread among workers during all levels.

Interleaved graph partitioning. As sec xx suggests, the above requirements are contradictory in the original single-level and all-to-all topology. Surprisingly, the Progressively converging ML-Shuffle brings opportunity to simultaneously achieve them. Since it transforms the rigid all-to-all connection into multiple levels of more sparse connection, each level has the favorable feature as follows.

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

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

From Theorem 1, any level can be decomposed into isolated Complete Bipartite Graphs (CBGs), which are ideal units for function co-location, since all edges in the level are evenly included by same-sized CBGs. In other words, by putting functions within each CBG to a same worker, data transmission of the level can be done via workers' memory instead of remote storage, greatly accelerating the communication. 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 level, yet the method can't be directly generalized to placement for the whole multi-level graph. Since the communication of adjacent levels' involve a shared function layer, e.g., level xx and xx both involve function layer xxx, co-location constraints of two levels must be simultaneously met, which leads to multiplied width of co-location units. Worse yet, when jointly considering all levels, 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 levels so as to solve them independently. Specifically, it removes edges in all even-numbered levels, delegating that portion of data transmission 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 Progressive convergence method (the parallelism is N), where each level respectively corresponding factors d_1, d_2, ..., d_k, if we remove edges of all even-numbered levels (2,4, ..., $\lfloor \frac{k}{2} \rfloor \times 2$), the whole graph can be divided into disjoint CBGs with width lying within $D_{odd} = \{d_1, d_3, ..., d_{\lfloor \frac{k}{2} \rfloor \times 2+1}\}$*

The interleaved approach acts as a heuristic algorithm to solve the NP-hard graph partitioning problem [14, 21, 22],

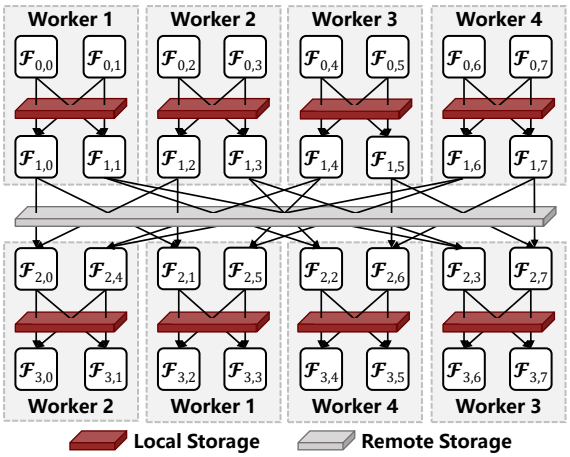


Figure 6: Function Scheduling

by giving quick solutions that fit the aforementioned requirements. Specifically, due to at least half of levels would be left for local memory to perform data transmission, performing function placement in units of resulting CBGs localizes over 50% of overall traffic. Meanwhile, since transmission media is assigned in an interleaved manner, no communication straggler exist during the job's execution. Last, it allows to selectively decide which factors in D would be put in D_{odd}, to minimize the resulting CBGs' width, thus achieving a fine-grained function placement that facilitates load balancing.

3.4 Configuration Modeler

As far, 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 layer 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 transmission time.

To achieve this, Configuration Modeler must precisely model each configuration's resulting transmission time. As discussed in §2.4, during each level's communication either the function side or storage side act as the real bottleneck, depending on the actual request number and data volume. More specifically, the duration would be $T_i = 2 \times \max(T_{i,f}, T_{i,s})$, 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 level's communication includes sender functions writing to storage side plus receivers reading back. Due to S3-based and memory-based communication alternate in different levels, 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 levels. Suppose the function number is N in each layer, in the S3-based level we have $T_{i,f} = \frac{D_i}{N * b_f}$ where D_i and b_f are each function's assigned data volume and bandwidth ceiling respectively, and $T_{i,s} = \frac{R_i}{q_s}$ where R_i is the involved request number and q_s is S3's request rate. In contrast, for the memory-based level $T_{i,f} = \frac{D_i}{N * b_d}$ and $T_{i,s} = \frac{R_i}{M * q_r}$, where M is the number of cluster nodes, b_d is the docker bridge bandwidth ceiling and q_r is redis I/O rate limit. To summarize, the overall data transmission time of a multi-level network is:

$$T = 2 * \sum_{L=0}^{maxLevel} \begin{cases} \max(\frac{D_i}{N * b_f}, \frac{R_i}{q_s}) & L \text{ is odd.} \\ \max(\frac{D_i}{N * b_d}, \frac{R_i}{M * q_r}) & L \text{ is even.} \end{cases} \quad (5)$$

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 runtime, preventing choosing the optimal configuration before job running. We use a sampling and profiling method to address the problem. Due to there is commonly a linear or non-linear but deterministic relationship between the size of input data and intermediate data [30], and D_i keeps consistent for all levels, Configuration Modeler repeatedly sample the original input with different sizes and execute the job, recording the amount of intermediate data. Then each time it gets a new <input 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 Configuration Modeler predicts the transmission time of all candidate configurations, and select 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 (both the compute instances and S3 buckets in cn-northwest-1 region). 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 which are not transparent to us, so as to better manage functions' execution and lifetime.
Workload. We adopt three widely-employed benchmarks involving the shuffle operation, 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 multiple SQL queries. Among them we select the most data-intensive one, i.e., the 16-th 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 [1], TPC-DS Tools [2], and Purdue MapReduce Benchmarks Suite [33].

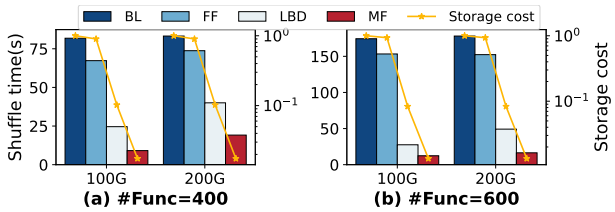


Figure 7: Terasort Shuffle Time.

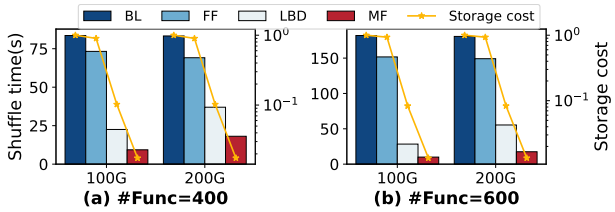


Figure 8: TPC-DS Shuffle Time.

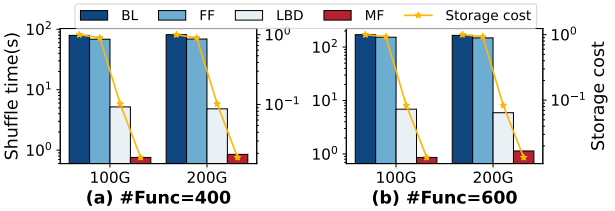


Figure 9: Wordcount Shuffle Time.

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 PUT/GETs to S3. We select its optimal configuration and denote it as LBD.

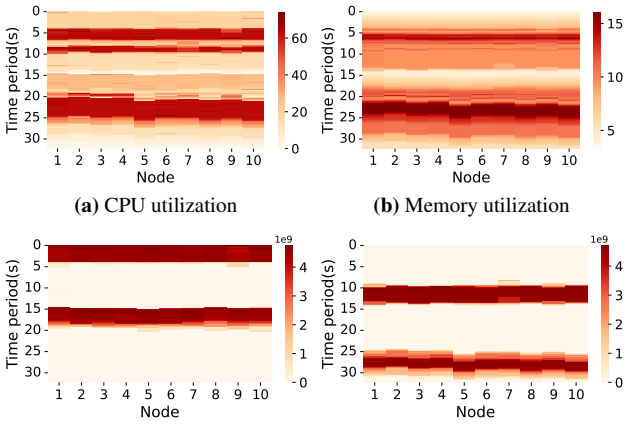
Configuration. During all our experiments we set the resource limit of each function as 2 CPU, 3GiB memory and 75MiB/s bandwidth, similar to prior research works [18, 23, 38, 42], to simulate a common setting of Amazon's commercial function service Lambda. By default we respectively set the input size as 100G and 200G, and set the parallelism as 400-function and 600-function, since MinFlow's mainly focuses on processing massive datasets with a large number functions, which is in accordance with serverless paradigm's goal to support hyper-scale computation with its superior scalability. Besides, we more extensively adjust the input size and parallelism to show MinFlow's performance under broader settings (see Section 4.5).

4.2 Microbenchmark results

Shuffle time & storage cost. Now we evaluate MinFlow's effectiveness on improving shuffle speed and saving storage cost. Figure 7,8,9 show the shuffle time and normalized storage cost (divided by the Baseline's storage cost) of all approaches and three different workloads.

Taking TeraSort as the example (Figure 7), we first focus on the 600-function parallelism with 200G input data size, the Baseline takes nearly 180s to finish the shuffle operation, during which not only all functions await, by the bill for using functions continues increasing as function services usually charge based on time (e.g., in increments of 100ms). Besides, compared to Baseline, FaaSFlow only slightly reduces the shuffle time and storage cost by 14.45% and 6.66% respectively, since most of PUT/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 Section2.4. In contrast, Lambada 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 Baseline and FaaSFlow, meanwhile saves the storage cost by 91.68% and 91.09% respectively. Nevertheless, Lambada still experience ~50s idling and awaiting shuffle's completion. As for MinFlow, it outperforms all the competitors in terms of both performance by slashing the shuffle time to 16s. Compared to Baseline and FaasFlow, MinFlow decreases the shuffle time by 10.8× and 9.3× respectively, and even compared to Lambada, MinFlow still achieves 3× faster shuffle speed. In addition, MinFlow greatly saves the storage cost (98.67%, 98.57%, and 84% compared to BL, FF, and LBD), for it not only greatly reduces PUT/GET number but also largely eliminates additional intermediate data volume via local storage. Under 400-function TeraSort with 200G input data size, similar to the 600-function parallelism MinFlow preserves considerable performance and cost improvement - as Figure 7 shows it achieves 2.1× shuffle acceleration and 5.6× cost saving compared to the runner-up Lambada. Yer one noticeable change is that the performance advantage of MinFlow over Baseline and FaasFlow shrinks, although still reaching as high as 47.97% and 53.38% respectively. To put it simply, it's because the performance degradation caused by excessive PUT/GETs alleviates under lower parallelism. We will conduct more in-depth experiments about this phenomenon later (see Section4.5).

Aside from the above TeraSort, Figure 8 and 9 show the results of TPC-DS-Q16 and WordCount. It can be found that while the results of TPC-DS-Q16 are quite similar with TeraSort, MinFlow exhibits much higher performance advantage 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 200G input data, compared to Baseline and FaasFlow, MinFlow optimizes the shuffle time by as high as 99.31% and 99.23%. Such phenomenon can be explained from the aspect of intermediate size - while TPC-DS-Q16 and TeraSort share a common characteristic that the intermediate data size of shuffle is consistent with the input size, WordCount has much less intermediate data since duplicate words in the input would be



(a) CPU utilization **(b)** Memory utilization

(c) Receive throughput (byte/s) **(d)** Sent throughput (byte/s)

Figure 10: Load balance. *Terasort,600#F,200G.*

counted for just once.

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 10a to 10d respectively show the CPU usage, Memory occupation, and traffic load of all 10 workers when running TeraSort under 600 functions and 200G 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, load intensity of each worker varies noticeably along the timeline, which is in line with 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 10a and 10b, at around 5-10s and 20-25s respectively, represents the positive correlation between compute and memory peaks. On the other hand, the peaks of incoming traffic occurs at around 0-5s (input) and 15-20s (GETs of shuffle), and peaks of outgoing traffic occurs at around 10-15s (output) and 25-30 (PUTs of shuffle). They are both interleaved with above compute/memory peak. In addition, by combining this set of results we could find that for MinFlow the shuffle only accounts for ~10s out of overall 30s job running time, partly verifying MinFlow's high shuffle speed.

4.3 Overall performance analysis

Figure 11 to 13 show the overall job completion time under the same setting as in Section 4.2. We still first take the 600-function with 200G 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% to 77.98% improvement for Terasort workload, 39.12% to 72.86% for TPC-DS, and 12.26% to 82.46% for WordCount. The improvement mainly comes from shuffle time reduction, since among the compute, shuffle, and input/output time only the

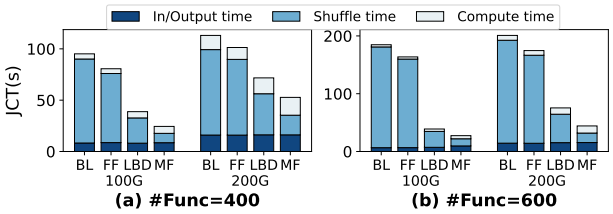


Figure 11: Terasort Overall Time

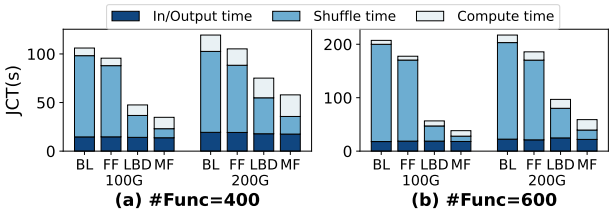


Figure 12: TPC-DS Overall Time

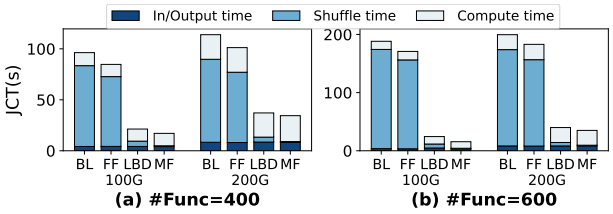


Figure 13: Wordcount Overall Time

shuffle time changes significantly, while the other two parts basically keep constant across all approaches.

Among three workloads, for TeraSort and TPC-DS that have same-sized intermediate data with their input, shuffle time plays an non-neglectable part throughput all compared approaches. For example, in the TeraSort group BL, FF, and LBD respectively spend 88.65%, 87.23% and 65.24% time in 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 100ms. However, when it comes to WordCount, though the overall time improvement is still significant compared to BL and FF, the benefit over Lambada shrinks, as Figure 13 shows. The reason is WordCount's reduction in intermediate size for word deduplication - While this does not noticably impact BL and FF's overall time for their bottleneck lies in excessive number of GET/PUTs instead of the transmitted data volume, it greatly alleviates Lambada's bottleneck that is mainly caused by function bandwidth limit. As a result, Lambada' shuffle time only accounts for 14.83% of the overall job execution time, largely neutralizing the great shuffle time improvement of MinFlow over Lambada.

Last, since theoretically the compute and input/output speed are proportional to function number, increasing the parallelism can easily slash both of 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, compared with 400-function parallelism in which

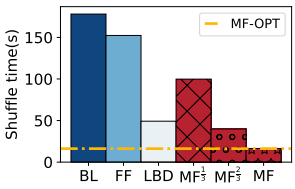


Figure 14: Breakdown.

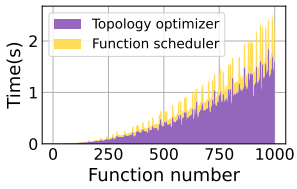


Figure 15: Scalability

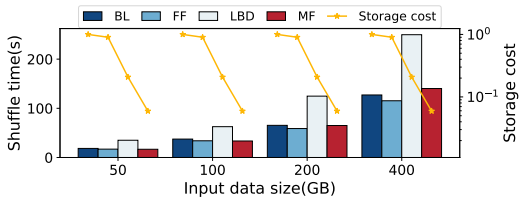
MinFlow achieves 26.49% to 53.38% overall time reduction, in 600-function parallelism the values increase to 41.35% to 77.98% respectively. To conclude, these results demonstrate MinFlow could improve the overall time considerably compared to existing works throughout all three workloads.

4.4 Breakdown and overhead

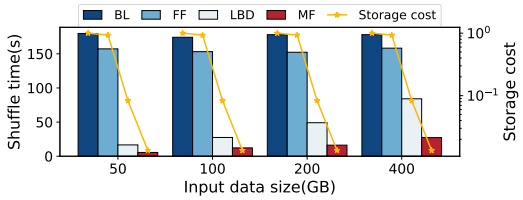
Performance breakdown. We progressively integrate the three components to show their respective contribution to MinFlow's shuffle time reduction. Figure 14 show the results of Terasort under 600 function and 200G input data, where the MinFlow with only Topology Optimizer is referred to as $MF^{\frac{1}{3}}$, 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 Baseline and FaaSFlow, but is slower than Lambada. This is because compared to Lambada that offers a two-level shuffle, by default Topology Optimizer of MinFlow choose the highest level number it can generates, to decrease entailed PUT/GETs maximally. Yet this often incur 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 Lambada, by 19% in the figure. Last, the full version MinFlow further integrate Configuration Modeler to judiciously select the optimal level number and corresponding suitable function scheduling plan, instead of just gluing the topology optimizer and function scheduler. As result, the full version MinFlow could outperform other approaches by 66.62% to 90.77%.

Besides, we further compare with MinFlow-OPT, whose configuration is get by iteratively running all configurations and picking the one with the shortest shuffle time. As the blue line in the Figure 14 shows, MinFlow achieves basically the same shuffle with MinFlow-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 Sec.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 functions number goes up. Under 600-function parallelism the time is not above 0.5s, 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 Sec.3.3). This part of time cost is close 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 cost 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 [22].

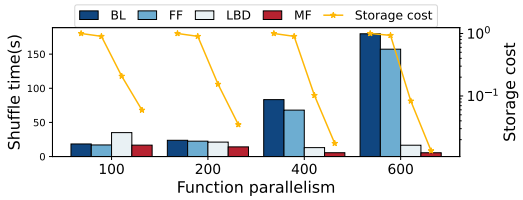


(a) #Func=100

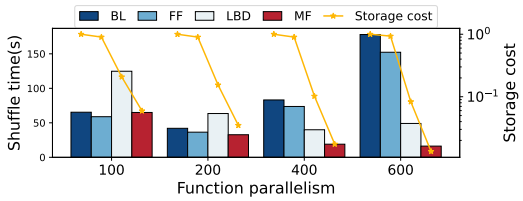


(b) #Func=600

Figure 16: Various input size.



(a) Input data=50G



(b) Input data=200G

Figure 17: Various input size.

4.5 Impact of configuration

As we mentioned previously, 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 trend.

Various input size. First we fix the parallelism as 100-function and tune the input size from 50G to 100G, 200G, and 400G. As we can see in Figure 16, across all approaches the shuffle time increases proportionally with the input size. The trend can be easily explained - Due to the number of GET/PUTs to S3 is not greater than $100\times100\times2 = 20000$, S3's speed of thousands of request per second is enough to rapid 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 low-parallelism setting, which is not MinFlow's target scenario, MinFlow could achieves near-optimal performance compared to others. By contrast, in the 600-function group (Figure 16), though Lambada and MinFlow still exhibit similar trend, the shuffle time of Baseline and FaaSFlow keeps consistent across all input sizes. Such difference stems from their distinct bottleneck. Specifically, for Baseline and FaaSFlow the 600-function parallelism setting would incur $600\times600\times$=720000 GET/PUTs, making S3's speed the main bottleneck. As a result their shuffle time is insensitive to the changing input size. However, due to Lambada and MinFlow's great effectiveness in reducing the number of GET/PUTs, their bottleneck still lies in functions' aggregated bandwidth sending/receiving intermediate data, leading to the shuffle time proportional to input size. Note though it seems that under high-parallelism, say 600-function, the performance advantage of MinFlow over Baseline and FaaSFlow 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 PUT/GETs as the new bottleneck.

Tunable parallelism. Figure 17 shows the shuffle time results under parallelism of 100, 200, 400, and 600 functions, with 50G input size. As we can see, for low efficiency of performing shuffle, i.e., the huge number of PUT/GETs to S3, both Baseline and FaaSFlow's shuffle time keeps getting worse with the parallelism increasing, greatly impeding the critical scalability advantage of the serverless paradigm. By contrast, Lambada 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 time. For example, under low-parallelism, say 100-function, it performs even worse than Baseline. In comparison, MinFlow not only preserves the continuously decreasing shuffle time but also avoid such degradation.

5 Related Work

Serverless DAGs optimization. Several recent proposals have aimed to decrease job complete time by optimizing the performance of serverless DAGs. Orion [25] first propose the idea of bundling multiple parallel invocations to mitigate execution skew and finds the best bundle size through trial and error. WiseFuse [26] 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 [10] and FaaSFlow [22] 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 Section 2.4. Overall, no prior work in this category can effectively reduce the data movement overhead of serverless data analytics.

Serverless intermediate data store optimization. Besides DAGs optimization, recent work also reduces job completion time by optimizing the intermediate data store. Pocket [19] and Locus [32] show that current options for remote storage are either slow disk-based (e.g., S3) or expensive memory-based (e.g., ElastiCache Redis). Therefore, To balance performance and cost, Pocket combines different storage media (e.g., DRAM, SSD, NVMe) that users can choose to conform to their application needs. But this approach only makes economic sense when running many different applications,e.g., when exclusively executing tasks like Terasort, Pocket consistently selects the costly NVMe storage as the intermediate data repository. Faasm [34] and Cloudburst [35] accelerates data movement between function instances, through a distributed shared memory across worker nodes. they relies on specific assumptions regarding the sandboxes 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 service. In contrast to existing efforts, MinFlow uses only cheap S3 and over-provisioned 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 compact bipartite sub-graphs. This optimizes function scheduling and cuts data transmission to remote storage by over 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 of the job completion time and storage cost.

References

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

[2] Tpc-ds, 2023. https://www.tpc.org/tpcds/#, Last accessed on 2023-6-29.

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

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

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

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

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

[8] AMAZON. S3 qps limit, 2023. https://aws.amazon.com/cn/about-aws/whats-new/2018/07/amazon-s3-announces-increased-request-rate-performance/,.

[9] 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.

[10] 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.

[11] 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.

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

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

[14] 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.

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

[16] 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.

[17] 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.

[18] 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.

[19] 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.

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

[21] 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.

[22] 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.

[23] 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.

[24] 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.

[25] 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.

[26] 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.

[27] 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.

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

[29] 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.

[30] 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.

[31] 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.

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

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

[34] 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.

[35] 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.

[36] 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.

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

[38] 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.

[39] 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.

[40] 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.

[41] 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.

[42] 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.